Discovering Cybersecurity

A Technical Introduction for the Absolute Beginner

—

Seth James Nielson

Apress®

Seth James Nielson

Discovering Cybersecurity
A Technical Introduction for the Absolute Beginner

Apress®

Seth James Nielson
Austin, TX, USA

ISBN 978-1-4842-9559-5 e-ISBN 978-1-4842-9560-1
https://doi.org/10.1007/978-1-4842-9560-1

Introduction

For most of its history, cybersecurity has largely been the domain of technical professionals. Professionals with this kind of background typically had technical certifications or college degrees. It was assumed that if you were talking about things like access controls, you also understood, to some degree or another, how passwords were stored and secured.

In recent years, however, there has been a growing demand for professionals with limited technical training to also be involved in cybersecurity. Executives and other leaders need to understand the ever-increasing threats cyberspace poses to their organizations. A demand for improved policy across private and public sectors is creating a need for policy makers that know how cybersecurity technologies work and what they can do. Legal teams and compliance teams are often challenged with cybersecurity responsibilities without necessarily having deep technical backgrounds.

Thus, cybersecurity is no longer just the purview of the engineers, researchers, and scientists that are specialists in that field. It is not even limited to persons with technical backgrounds. To the contrary, there are an ever-increasing number of nontechnical professionals that need more in-depth understanding of how cybersecurity works.

There have been many approaches to teaching cybersecurity to those without a technical background. It has been my experience, however, that these approaches have largely left out most of the technical details of the subject. The reason is typically a belief that nontechnical professionals do not *need* to know these details, or *cannot* understand them, or both.

I do not accept this view. For several years, I have had the privilege of teaching graduate students in law, policy, and business about exactly this. My classes have pushed nontechnical students to learn technical concepts. It has certainly been challenging, but the students have done extremely well every year and have, by large majorities, felt that the class was exceptionally beneficial. This book follows the curriculum of the class and teaches the concepts that are covered therein.

This book, as the title states, is intended for the absolute beginner. It assumes no technical training of any kind. At the same time, it is a *technical* book. It gets as deeply into the selected topics as it can, and it will most likely be a challenging read for the beginner. Challenging, but doable.

The book covers enough technical material that it is also useful for those that are *not* absolute beginners. For those with technical degrees but not cybersecurity training, this book can provide a gentle introduction to the topic.

With all of that said, let me be clear that this book is not perfect. Finding the best way to explain a technical concept to beginners is extremely challenging. During the time that I have been teaching these materials, I have extensively revised the explanations and approach every semester. Each semester, student feedback guided what stayed and what changed. As the materials have stabilized, I felt that the material could be captured in a book form. But even then, there was some extremely good feedback from my most recent semester’s students about how to teach the cryptography chapters that could not make it into this edition of the book. I have also had early reviewers correctly suggest that in the next edition I should incorporate the NIST Cybersecurity Framework in much greater detail.

If you find topics or areas of the book that you struggle with, I would appreciate the feedback. It will be useful for improving my teaching materials now and perhaps a second edition in the future.

In terms of organization, the first chapter is generally a very easy read and does not have a significant amount of technical content. After that, the chapters get more challenging and more technical, especially the cryptography chapters in the middle of the book. To help with some of the core concepts of computing that are used in these chapters, I have provided appendixes at the end that provide a short overview. If you are one of the readers that is an "absolute beginner," you might find it useful to read the appendixes at the end of the book *first*.

Above all, I hope that you can really enjoy discovering cybersecurity. There is a certain irony to the fact that cybersecurity is *deadly* serious but can also be fun and enjoyable to be a part of. What you are about to read is *important*, but it is also *exciting*. At least, I think it is and I hope I can share some of that with you.

Contents

About the Author

Seth James Nielson

, PhD, is the founder and chief scientist of Crimson Vista, a cybersecurity engineering company. He advises clients from startups to Fortune 50 companies on security matters. Dr. Nielson also teaches cybersecurity courses at the University of Texas at Austin. He has authored or coauthored papers on topics such as IoT security, hacking portable chemical manufacturing systems, and methods for teaching computer security to students. Dr. Nielson also coauthored the Apress book *Practical Cryptography in Python*.

About the Technical Reviewer

Massimo Nardone

has more than 22 years of experience in security, web and mobile development, cloud, and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. Massimo also holds a master of science degree in computing science from the University of Salerno, Italy. He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years. His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc. He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj. He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_1

1. The Psychology of Cybersecurity

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide

This chapter is focused on the technology of the human brain and how that technology interfaces with other devices and operations associated with cybersecurity. Much of the *human-made* technology in this area does not interface with *humans* very well at all. The systems and methods used to protect humans need to accept and account for human *errors* and *manipulation*.

Key Concepts

1. Cybersecurity is largely a battle of *wits* where the best thinker wins.
2. There will never be a “perfect” defensive technology that attackers cannot think around.
3. Security professionals and average users alike have natural limitations related to *error* and *manipulation* that attackers exploit.
4. Four sources of human error: *mental automation*, *complex rules*, *meta-ignorance*, *wrong model*

stubbornness.

5. Four sources of human manipulation: *action bias*, *emotional fallback*, *deference to authority*, *visual-emotional responses*.

6. Six suggestions for psychologically aware design: *affordances*, *irrational modes*, *rational centering*, *error robustness*, *failure robustness*, *manageable decisions*.

Common Pitfalls and Misunderstandings

1. “Smart” humans cannot be manipulated and are always rational; only stupid or weak people are vulnerable.
2. Cybersecurity technology does not need to account for irrational humans; mistakes they make are their own problem.
3. Training, or otherwise explaining things to people, will or should “fix” all sources of error and manipulation.

Useful Vocabulary

- **Bias**: A built-in and necessary part of the human mind that leans toward certain default preferences, especially in the absence of information
- **Social Engineering**: Obtaining unauthorized information or services through misrepresentations and manipulations of people
- **Affordances**: A configuration or design that induces a specific human behavior, including desirable behaviors

This is, in fact, a book about *technology*. You might, therefore, be forgiven for thinking a chapter about psychology is from the wrong book. Be assured, dear reader, that you are in the right place. This chapter is about the technology of cybersecurity like all the others. The technology of this chapter is *the human brain*.

It is essential to begin our investigation into the technologies that (nominally) protect the cybersecurity world with a discussion about the brain. Too often, people, both professional and lay users, hope to find technologies that can replace rather than enhance human thinking.

Years ago, I was wrapping up a routine dental cleaning. While chatting with the dental hygienist during checkout and payment, the hygienist found out that I worked in computer security. “Oh,” she said with some energy, “I am really worried about protecting myself online. What program should I install to protect myself?”

Then, as now, the answer to such a question is that there is no such technology. As will be discussed in later chapters, “security” without any additional context is a meaningless word and requires some *thinking* to even clearly state what needs to be protected and how. Nevertheless, even with better-defined objectives, “security” cannot generally be achieved with technology alone. Good security requires a thoughtful human brain to be engaged somewhere.

> The real question is not whether computers think but whether people do.
>
> — *B. F. Skinner*

The hygienist was very disappointed. She had been hoping that no thinking would need to be involved—that some technology could be used that would permit her to escape engaging her brain. Why? Is it laziness? Not for most users and certainly not for my excellent and hardworking hygienist. The problem, at least in part, is the ever-increasing complexity of modern society.

The hygienist knows *dental hygiene*. From personal experience, I can attest to her proficiency and professionalism in her chosen field. If she is like many modern individuals, she also has to worry about social commitments, political issues, children's education, home maintenance, car maintenance, diet, exercise, preventative medical care, and a host of other matters unrelated to her primary expertise. For the vast majority of these matters, most modern persons rely overwhelmingly on other experts to help them manage their lives. Public education, auto mechanics, home repair professionals, doctors, and others are often expected to really know and manage these facets of modern life with generally minimal involvement from the non-expert. For example, the auto mechanic generally suggests bringing in your vehicle for service every 15,000 miles rather than trying to make the owner of the vehicle more capable in the field of automobile maintenance.

Because we cannot possibly have expertise in all areas of our modern life, we all have to outsource our "thinking" to other people and/or automation at some level or another.

Most people, even most highly technical people, would prefer to do the same with computer security. Most people are looking for solutions that don't require them to think about it, keep current in the ever-changing landscape, or spend time and energy solving crises. Surely, one might ask, such off-loading must be possible? Can we not have security mechanics the same way we have auto mechanics? Can we not have security "doctors" that prescribe a medication and tell us to come back for a check-up in six months?

The answer *must* be yes because *most of the users that are not security professionals cannot be mentally and psychologically ready to face their cyberspace adversaries*.

I want to emphasize that the issue under discussion is *thinking*, not *knowledge* or even *education*. As I will discuss in this chapter, the human mind is actually configured *not to think* under a wide variety of conditions. Our evolutionary makeup has created a brain that *shuts down* higher-level thinking for many reasons. Each of these conditions can either create poor responses to cybersecurity incidents or leave people vulnerable to active manipulation by attackers.

Some of these psychological issues cannot be overcome with training of any amount. Others can only be overcome with professional-level training. If professional-level security training and thinking for the average user are necessary to be secure, our current system is doomed for obvious reasons of scope and scale. It is also worth noting that our interconnected world also means that when even a small number of individuals are vulnerable to attack, everyone’s risks increase.

Accordingly, cybersecurity *must* eventually become more automatic and transparent for the average person. But it cannot be just a stand-alone tool that can run in a “fire-and-forget” mode. It can never be “take two of these and call me in the morning.” The best cybersecurity solutions available now and in the future require technologies that push the thinking to professionals and experts and technologies that enhance, encourage, and enable the professionals and experts to think effectively. *There will have to be effective thinking happening somewhere.*

It is crucial to understand the intrinsic, inescapable difference between computer security and all the other life problems off-loaded onto other experts and automation. This difference is literally a “game changer.” It seems obvious to everyone, and yet very few individuals really focus on how this one fundamental truth about cybersecurity must be the foundation upon which we shape the entire approach and solution.

The difference is that in cybersecurity, *highly motivated* human beings, who are literally connected to potential victims at effectively instantaneous speeds, are actively *strategizing* to subvert our data and systems. They are not random mutations of cells that occur with predictable frequencies. They are not parts breaking down in a machine with expected failure rates. They are not static obstacles that can be sidestepped. They are *adversaries*: “one[s] that contend[] with, oppose[], or resist[]” [180]. They have goals and objectives and will stop at nothing to achieve them.

There are cancer[1] survivor t-shirts that say things like “my body tried to kill me,” but imagine how much worse cancer would be if it *strategized*. If cancer could understand cancer treatments and consciously modify its behavior to circumvent such treatments, the danger would be astronomically greater. It would no longer make sense to describe cancer outcomes statistically. A treatment that was 90% effective today could be irrelevant tomorrow if cancer could plan new attacks.

As an alternative mental exercise, imagine if cancer were self-aware and concerned with its own survival. Many cybersecurity attackers engage in cyber warfare for personal survival. This is literally how they make enough money to eat. In their minds, *failure is not an option*. They *will* break into systems and steal something *or they will* *starve*. How long could any cancer patient last if cancer was *motivated*?

Our cybersecurity adversaries will always need to be defeated by human minds.[2] Technology can only enhance, but not replace, *thinking* when dealing with a *thinking* adversary.

The warfare-like nature of cybersecurity means that the winner is the best thinker (or strategist if you prefer). No static defense will last indefinitely against the carefully planned attacks of a motivated, intelligent adversary. Herodotus, an ancient Greek historian, tells us that the city of Babylon had walls that could not be breached and supplies to withstand any siege. However, the Euphrates river ran through Babylon and the highly motivated Cyrus, "drain[ed] the Euphrates into the artificial lake...and invad[ed] the city through the riverbed..." [190].

Fixed fortifications are a monument to the stupidity of man

—US General George Patton

Ideally, cybersecurity would permit many people to be protected by a relatively small number of active, motivated, and thinking brains. A science fiction–based video game from the early 2000s included a military leader that was dealing with aliens that delivered devastating psychic attacks directly into her people's minds. In response, she proposed:

> Against such abominations, we organize our defenses on the principle that one strong and able mind can shield the many.
>
> *—From Sid Meier's Alpha Centauri video game*

This quote, although fictional, perfectly captures how cybersecurity can, and eventually must, be. And, as already stated, the professionals must also be secured against the various sources of errors and manipulations to which all of humankind is vulnerable.

In this chapter, I will discuss two major psychological-driven issues: *error* and *manipulation*. Because all humans invariably struggle to deal with both categories, the final section investigates how cybersecurity defenses must be built with these limitations in mind, as opposed to many of our current technologies that fail to do so.

The Human Brain As Security Technology

In this book, I will review many technologies and technology families. The analysis reviews each technology across the following characteristics:

- Forces behind design and development
- Intended purposes
- Feature set
- Strengths
- Weaknesses
- Contextual requirements
- Deployment in practice (intentional or not)
- Lessons learned and future directions

The human brain, as a cybersecurity technology, benefits from a similar analysis. Our working models for the brain are based, at least in part, on our current understanding of the evolution of our species. According to these theories, much of the behavior humans exhibit in our modern, technologically advanced society are rooted in our desire for survival in primitive ages. Even terrible and reprehensible behaviors and beliefs, such as racism and tribalism, may have roots in defensive mechanisms. Primal recognition of groups that look and think alike represent an unsophisticated version of “friend-or-foe” recognition [234].

What kind of “security” issues plagued our early ancestors? We can guess that their threats included wild animals that were faster, stronger, and more heavily armed than humans (at least more heavily armed than humans without tools). They certainly included other groups of humans anxious to control limited natural resources of food, water, and other materials.

The intellectual tools such as reasoning, planning, and strategizing available to prehistoric humanity could not enable them to survive all of the threats they faced. If a predator appeared unexpectedly, there was no time to *think*. There had to be automated or semi-automated *reactions*. Higher-level brain functions, such as planning, strategy, and reasoning, were best suited to long-term planning. These capabilities were extremely powerful and have led to humanity’s dominance as a life form on earth. Nevertheless, high-level thinking, for all of its power, could not eliminate the need for low-level, *immediate*-term survival responses.

There are other evolutionary reasons why brains retained low-level responses that are not based on logic or reasoning. For example, human reasoning is sometimes unable to reach a confident conclusion when critical information is missing. Perhaps, in many cases, it was better to do *something* rather than nothing, and low-level responses did the job. Or perhaps there were advantages to cohesive social groups over individuality. It is conceivable that *not* thinking, and simply following a leader, worked out better for primitive humans over having every individual think for themselves.

Of course, we do not want to be guilty of just examining human behavior and making up an evolutionary story that seems to fit [234]. Researchers in the field attempt to model evolutionary influences with testable hypotheses. Obviously, a deep exploration into this research is not going to fit into this book, nor am I a sufficient expert to opine deeply on the topic.

Nevertheless, what is clear is that human brains do not operate as logically and rationally as we tend to imagine they do. Rather, a significant portion of the human experience is low level and irrational. These brain systems appear to have been shaped as a product of human evolution.

This, however, introduces a very serious problem. As we will discuss repeatedly throughout this book, security technologies generally have a *context* in which they are effective. It is very rare for a system to provide useful security outside of the context for which it was designed. However, to the extent that evolutionary forces "designed" (or shaped) our brain as a security technology, *most of those forces no longer exist*.

The average human has very little to fear, for example, from wild animals. Certainly, being eaten alive is not the leading cause of death or even in the top 100.[3] In fact, we have so little to fear from being chased down and devoured that some of our leading causes of death are from sitting around with minimal physical activity [219]. Even living in the concrete jungles of urban environments is different from the habitations human brains "grew up" in.

Moreover, new threats have been emerging in recent human history that our brains were not designed to deal with and for which evolution does not have time to respond to.[4]

Story Time: Negativity Bias and Time Horizons

It is well understood that the human brain has a "negativity bias"—we put more emotional and cognitive emphasis on negative events and circumstances than positive or neutral ones. This bias is well adapted to survival in the settings where our species evolved (the vigilant and attentive humans who feared each sudden movement behind a nearby bush survived longer than the overly optimistic and curious humans who tried to befriend the crouching lion waiting to make a meal of them). But in the modern world, threats to our existence, livelihood, and life satisfaction take on vastly different forms, and our brains often misinterpret the dangers [127].

This is especially true when the delay is long between detecting a threat and realizing its consequences, a phenomenon known as "delay discounting" [211]. We understand well the benefits of healthy eating and exercise, but some of the most dire consequences of neglecting these habits are so far removed from us, temporally, that we don't intuitively associate our actions with the outcomes when it's time to make those small, incremental decisions.

Cybersecurity threats are often of this character: consequences far in the future, of uncertain magnitude, and possessing only tenuous intuitive connection to our daily business actions. For the sake of the humans on the front lines, whose job is ostensibly not anticipating such worst-case scenarios, it's crucial that we design our processes and systems to expect the aspects of human psychology that might otherwise undermine our collective efforts.

In summary, in terms of being a security technology to keep us safe, human brains have some weaknesses inherent in their "design." At least some of their operations were designed for security problems that no longer exist and cannot be “redesigned” quickly enough to deal with the explosion of new threats, such as those found in cyberspace.

Correctly Understanding Human Cognition

In my experience, most people in general, and too many computer security professionals specifically, think of humans as if they are logical. This is always a problem for cybersecurity systems for two reasons.

First, the security “system” relies on human input and decision making. Systems can be more automated or less automated, they can have many interaction points or few, but no system is completely independent of humans and completely autonomous in its decision making. In other words, *the effectiveness of any cybersecurity system depends at least in part on humans making correct, or “good,” decisions.* If a deployed system does not work because one or more humans made “bad” decisions, the designers often respond by blaming the human or humans that made the “mistake.” From some designers’ point of view, the humans are not part of the system, and failures happen in the technology [165]. Even designers that view humans as part of the system see the human failure as an indication that the errant human needs to be “fixed.” This often takes the form of “the user just needs to be educated” solutions. More training. Better documentation. While these are not necessarily bad ideas, they often do not address human psychological limitations.

Second, and even worse, the cyber criminals *do understand* human psychology. Either consciously or instinctively, many of the attacks in cyberspace directly target humans using psychological tricks, maneuvers, and ruses. Given that there are conscious, thinking adversaries launching attacks using effective psychological methods, cybersecurity professionals should *not* assume “normal” thinking on the part of the defenders. Defensive technologies must be designed to operate effectively when the humans in the system are being assaulted with sophisticated, “anti-thinking” tactics. Sometimes, these kinds of solutions need to push decision making away from average users that are not equipped to do the thinking of cybersecurity. Other times, solutions need to assist the professionals effectively apply their training and thinking to the problem at hand.

I will get deeper into how security technology *should* be designed in the next section. First, I will walk through a sampling of some well-known psychology issues that present cybersecurity challenges.

The Psychology of Human Error

Some cybersecurity failures are the result of an error on the part of a human. These errors can range from giving access to an unauthorized individual to not updating vulnerable software on a server. Because there is an identifiable mistake, the assessment of what went wrong often stops with the individual or individuals that actually committed the error. But understanding *why* humans make errors can lead to better designs. In this section, I will discuss four common sources of human error: *mental automation*, *complex rules*, *meta-ignorance*, and *wrong model stubbornness*. Obviously, this is not a comprehensive list but should provide a helpful introduction. Some of this content is based in part on Ross Anderson’s discussion of Cognitive Psychology [40, Chapter 3].

After discussing each of these sources of error, I introduce a few mechanisms by which attackers exploit errors.

Mental Automation

One very common reason for human error is when the brain goes into automatic mode. When someone first begins to learn a skill, there is a lot of conscious thinking. The initial learning process involves close attention to detail, critical analysis, and discovery. After a relatively short period of repetition, however, these high-level thinking processes diminish in the execution of the skill. Automation replaces conscious thought for even relatively complex tasks. An example of this is driving. It involves a combination of tasks such as route planning, spatial organization, risk analysis, and processing signals from signs and other drivers. And those tasks do not even include the mechanical components of controlling the gas, brake, and steering wheel. Yet, despite the complexity, most drivers do many of these elements automatically.

In some cases, they may do them more automatically than they intended and find themselves driving to the wrong location. Their automatic processing takes over, and they travel the route they take most often or most closely associated with an initial thought process. The execution of an incorrect automated task (usually a higher repetition task) over another is called “capture” error or sometimes “slip and capture” error [195].

This type of error illustrates the danger in assuming that the human in the loop is *thinking*. If you think about all of the things that you do in a given day, you might be surprised to see how many of them you did *not* think about. Systems that assume conscious thought or that require conscious thought may be vulnerable to exploitation of these kinds of errors.

Complex Rules

Other errors happen even when thinking is engaged. A second source of error stems from the challenges of decision making when many competing rules may or may not apply. Human decision making is guided by rules, but rarely does one rule override all others. Using the example of driving a car again, consider the many rules that go into choosing the car's speed. Obviously, the posted speed limit must typically be followed for a non-emergency vehicle. But weather factors are also rules that must be considered. Or when visiting a new location for the first time, a driver may consider slowing down somewhat in order to take the correct turn or stop at the right address. At the same time, if the road carries a lot of high speed traffic, going too slowly might be a safety hazard. Balancing all of these rules requires making reasonable decisions about which rule is the most important under the current circumstances.

There are many situations in computer security where figuring out which rule applies to a given situation is not immediately obvious. The user may make an incorrect choice and not realize it because they are consciously following rules. They just happen to be the wrong rules for the current scenario. They might be the correct rules at other times or even most times.

But I hasten to emphasize that the biggest problem with complex rules is not that the rules are not known. You could provide a user with a codex of all the rules and that would not address the underlying problem. The primary issue with *complex rules* is that *the user will generally think they are doing the right thing because they are following some rule*. In other words, because of the complexity of the rule set, it is easy for the user to pick a rule to follow and *believe* they are doing the right thing because they are following a rule. When confronted with the erroneous behavior, the person will generally protest that they were justified in what they did and even cite the very rule that they chose to follow. The psychology lesson is that the complexity of rules can actually impede a user from doing the right thing because doing the wrong thing is now justified and supported by a misapplied or misunderstood rule. The user persists in the erroneous behavior because they were “doing everything they were supposed to.”

Accordingly, it is crucial for designers to be aware that “good” decisions or rules can become bad when misapplied. It should not be assumed that users will always be able to tell the difference, especially when there are many rules and lesser-used rules.

Meta-ignorance

The third source of error is simply not understanding the problem and *either not being aware of this lack of understanding or feeling pressured to act anyway*. In other words, the problem is not ignorance. It is an inability to understand *how to respond to ignorance*. In other words, this is a form of *meta-ignorance*.

The critical psychological insight for this particular class of error is how easy it is to think that a problem is under control when it is not. This class of error generally covers the problems that occur when people should get help from experts or better trained individuals but do not. These kinds of errors are common when things can appear to be working correctly and perhaps are working correctly *under normal circumstances*. In cybersecurity, for example, it is often difficult to know that the security is not working correctly before an adversary attacks. And the attack scenario is the *abnormal* one. Up until that moment, everything seems fine. Because of this, users may not reach out to get guidance from better trained persons because nothing appears to be wrong. Or, in other situations, users may be aware that the situation is not ideal but choose not to consult with experts because of pressures of time, budget, or politics.

In my classes, lectures, or books, I often emphasize the importance of knowing when to get help from subject matter experts (SMEs). One very difficult area where this applies is *cryptography*, which is the subject of Chapters 5 and 6. I often tell students to remember YANAC: *You Are Not A Cryptographer*. Most decisions about cryptography require an SME. I will discuss some of the reasons why cryptography is so easy to get wrong in the aforementioned chapters. But even beyond that particular subject, many cybersecurity technologies require cybersecurity expertise to fully understand the context of the problem they solve, the trade-offs of one configuration over another, and pitfalls associated with their deployment. An important lesson for the non-expert user is to know when experts are needed.

Nevertheless, *relying* on users having this kind of understanding, or pretending that there are no pressures against this kind of behavior, is not ideal. It is certainly not facing reality.

In teaching about this concept to my students, a large portion of any given class will focus on the term “ignorance” and immediately assume that this is solved with “training.” This response, however, ignores that different kinds of ignorance require different solutions. A paper entitled “The Five Orders of Ignorance” by Phillip G. Armour identified that dealing with “known unknowns” was different from dealing with “unknown unknowns.” The next level up was an even higher order of ignorance in which there was no process for exploring or uncovering or discovering unknown unknowns [44]. Armour was writing from a software engineering context, but the ideas apply to computer security as well. Meta-ignorance cannot generally be solved with training because that is generally only suited for dealing with known unknowns. Meta-ignorance requires developing a security process within the individuals in an organization that enable them to figure out what kind of security circumstance they are in, know when and where to go to get help, and incentivize these activities.

Story Time: Pay No Attention to the Man Behind the Curtain

An example can illustrate how a cybersecurity incident can be made worse by failing to investigate warning signs or actively instructing users to ignore them.

In March of 2023, 3CX's softphone application software was compromised by a supply chain attack, in which a software vendor is compromised to subsequently push malicious updates to a large number of organizations [86, 123, 261, 276]. Users first reported seeing SentinelOne (antivirus software) flagging it as malware on March 22 on the 3CX forum, but lack of acknowledgment and communication from the vendor caused confusion about how to handle it and whether it was even a threat at all. Some people on the forum claimed to have been advised to whitelist the compromised software [80], others reported being banned for reporting issues [254], and others claimed that 3CX was attempting to shift the blame for the incident to the SentinelOne antivirus software [254]. CrowdStrike performed an analysis that confirmed the malware and provided information on the likely threat actors behind the attack [86]. Eventually, on March 30, 3CX finally admitted that their software was indeed infected with malware and provided some guidance on how to address it [79].

Wrong Model Stubbornness

Wrapping up, the fourth and final source of error I discuss in this brief survey is what I call *wrong model* *stubbornness*. For obvious reasons, humans are incapable of perceiving their environment completely or unfiltered. Our inability to observe things completely is manifested in many ways. We cannot see behind us, we cannot see through opaque materials (e.g., around corners), and we see less detail as distance increases. Even with all scientific advancement, there is much that is not known about the human mind, the deep ocean, and subatomic particles. We are also limited by time: even if every question could be answered and every part of the universe explored, nobody would have time to do it all.

We also cannot experience our environment unfiltered. Our minds are not designed, and indeed it may be impossible, to comprehend our environment particle by particle. Even if it were possible, our only knowledge of our environment, our reality if you will, is determined through five very limited senses. The data is both too much and not enough at the same time.

The human mind solves these limitations through the use of *models*. We model everything. Physically, particles that we can perceive are abstracted into materials, and materials are abstracted into objects. Take a very simple concept like “a window.” What is it? Can you define it concretely? Whatever definition you have forming in your head as you read this, I can almost guarantee there is an exception. Because a window is a *concept*; a model. And we match the model to the environment in ways that make sense to us. The architect Christopher Alexander described these models in buildings as *patterns*, and his work became the basis for describing certain models in computer programming called software *design patterns*.

These kinds of models enable us to quickly abstract our environment. We can think in terms of “office” instead of desk, chairs, computer, papers, folders, and books. We can think of “car” instead of four wheels, five seats, engine, mirrors, windshield, etc. We can think of “book” instead of hard cover, 300 pieces of paper, and symbols printed in the form of words.

We even model people. We have *friend* or *enemy* models, *spouse* or *partner* models, and *teacher* or *student* models. We have models for politics, religion, profession, recreation, and sports. Literally everything we *think* is a model of some form or another.

Because we cannot know everything, we rely on models to predict the future and decide courses of action. If you have modeled one person as a *coworker* and another as a *spouse*, you can predict that the coworker will not have a negative emotional response to you leaving for a different company, but you might predict that the spouse will have a very negative reaction to you leaving the marriage. Models can be wrong of course. The coworker may have expected some loyalty that you did not include in your model. And you may not have modeled that your spouse was also interested in leaving the relationship. But we all have to model nonetheless. There simply is no other alternative.

Wrong model error occurs when we apply the wrong model to the data that we have available. Specifically, this term does not include errors associated with a model being incomplete or inaccurate, but is meant to identify the errors of *applying the wrong model altogether*. A wrong model error would be applying the coworker model to a spouse or vice versa. Typically, a wrong model is selected either because some initial, more limited information available suggested it or because the correct model is simply not known at all. Selecting the wrong model can cause subsequent information to be misunderstood, miscontextualized, or ignored. It will almost always result in incorrect responses and decision making.

You may have experienced wrong model error when you try to discuss problems or concerns with others. Have you ever been talking to someone about a problem and had them suggest solutions or ask questions that seem completely unrelated? Quite often, you have selected one model, and the other person you are talking to has selected another. When the models are different, the words said are almost always interpreted incorrectly, and very little communication is taking place.[5]

But as problematic as wrong model error is, it is also *normal* and *expected*. After all, we have to create models based on the best information that we have. Applying the wrong model because it seems to fit the data is good thinking even if it is wrong. Given that we do not have perfect information, applying a wrong model is inevitable.

However, good thinking also means *abandoning the wrong model when new evidence or information becomes available that invalidates it*. Unfortunately, for whatever reasons, human minds seem particularly unwilling to give up a model once we have adopted it. Sticking to the wrong model in this form often involves ignoring or discounting information that does not fit with the existing construction while simultaneously accepting uncritically any and all information that conforms with it. I have termed this problem *wrong model stubbornness*.

Wrong model stubbornness has shown up in some high-profile criminal investigations and prosecutions. In a 2005 paper by Susan Bandes, a scholar on the role of emotion in the legal system, she discusses several cases in which prosecutors became convinced of their theory of the case, refusing to concede they were wrong even after strong evidence emerged to refute them. She writes:

> The recurring theme of these cautionary tales–and the dynamic on which this article will focus–is the prosecutor's tendency to develop a fierce loyalty to a particular version of events; the guilt of a particular

> suspect or group of suspects. This loyalty is so deep it abides even when the version of events is thoroughly discredited, or the suspect exculpated. It results in a refusal to consider alternative theories or suspects during the initial investigation, or to accept the defendant's exoneration as evidence of wrongful conviction.

Bandes uses the term "theory," and I will more or less equate that with model. One reason I prefer model is because investigators are often attempting to fit the current case to other types or classes of cases experienced in the past. One such model is what I will generically call an *insider* model. Whenever there is an abduction, for example, the police look closely at the family to figure out if one of them, the insider, did it. If police, prosecutors, or others in the justice system become convinced that a model (such as "insider") applies, it appears that they can experience *wrong model stubbornness*. As already discussed, it can lead to the ignoring of data that does not fit the model and overemphasis on data that does.

The same thing can happen when a cybersecurity professional is investigating intrusions and security incidents within their organization. A professional has experience, which is a good thing. But if the professional attempts to force a new situation or scenario to fit with previous experience, they may experience *wrong model stubbornness*. To reiterate, it is good thinking to apply old models to new problems even if it is the wrong model. However, the professional must also be ready to abandon the model when the evidence is sufficiently against it.

The lesson for the cybersecurity technologists is understanding how models impact the user's ability to understand, process, and use the information given to them. Just because information is available, it does not mean that the human is *interpreting it correctly* and, in fact, may be *ignoring or discounting* it because of a model that they are struggling to give up. The designer should not necessarily assume that the humans in the system have adopted the correct model even when the information is sufficient to do so.

Errors and Cybersecurity

To reiterate, errors by humans can be and are exploited by attackers to bypass defenses, steal information, or compromise systems. The cyber criminals analyze systems for error-inducing components and then search for ways to use potential errors to their advantage. Errors might be induced directly or abused opportunistically.

Direct manipulation usually involves sending some kind of input to the system such as an email, phone call, or other system access that causes someone within the system to have to react to the input. The input is chosen to specifically stimulate the error-precipitating characteristics of the system and push, if at all possible, the victim user to a wrong decision of the attacker’s choosing. A very simple example is an attacker redirecting a user’s browser from the real website to a fake one. It used to be common for browsers to simply *warn* the user that something looked wrong. But these warnings were common even for legitimate websites, and users often just clicked “OK” and went to the fake website anyway. This was an automated behavior learned during normal operations. Users were so accustomed to having to “click through” these errors that they rarely even read the error. Modern browsers have improved security by making many of these warnings hard blocks; you simply cannot visit the page with the invalid credentials. But this kind of error is still possible under certain circumstances.

On the other hand, because the users of a poorly designed system are often making mistakes even without malicious input, attackers can also look for already-vulnerable systems. Many systems ship, for example, with default passwords that are supposed to be changed before real deployment. This is sometimes not understood by nonspecialists and overlooked. Attackers can easily find such systems and subvert them. In 2016, a massive number of IoT devices, such as cameras, were taken over on a global scale creating what is called a *botnet* (subverted machines are often called "bots" because they are controlled by the attacker and do their commands). This botnet was so large it could take down websites by just sending traffic from the millions of devices it controlled to a single target. The website would be unable to process the unending requests from the millions of devices rendering it unusable [41].

The Psychology of Manipulation

Beyond erroneous behavior, the human mind is also manipulatable by others. As with the preceding section on psychological sources of error, in this section I will survey some of the limitations of the human mind that enable manipulation. As before, this is not a comprehensive list but is meant as an introduction to the problem.

For each of the issues discussed in this section, the common theme is the reduction of critical thought in exchange for some kind of automatic or reactive response. From the attacker's perspective, these kinds of behaviors are desirable because such responses are *predictable*! Attackers want predictable because they can plan for and exploit predictable behavior. On the other hand, people that exercise critical thinking and methodical evaluation often evade the manipulation.

So, the problem of manipulation in many cases reduces down to triggering an automatic or reactive response. Unfortunately for the victims, the human mind is designed to act without thinking (critically) in many circumstances. As I discussed in the overview of the brain, these features of the mind were driven by evolution. They were most likely an advantage for primitive humanity. In many cases, they are probably still necessary today.

Action Bias

The first limitation to discuss is perhaps the most direct embodiment of not thinking. Called either *action bias* or *bias toward action*, humans in crisis situations have an impulse to act even if they lack the information necessary to make a good decision. Drawing again from the earlier background, early humans needed to run from a predator whether or not they had a good plan for it. It may not be a perfect solution, but it was better than no solution.

Even in modern humans, a bias toward action is sometimes, maybe even often, a valuable bias. Early actions, even without very much information, can help to discover and generate additional data. Sometimes, imperfect actions are a necessary step at getting to better choices and outcomes. Because it is impossible to think through every single decision, biases, such as a bias toward action, are essential for basic function.

Before proceeding, I also need to talk through the word *bias*. Bias in modern usage has an extremely negative connotation because it is associated with socially unacceptable biases such as racial bias, gender bias, and other group biases. However, the word bias refers to *preferences* that are preconceived, automatic, or otherwise not based on a rational basis. A bias toward action, therefore, is a preference to act rather than to not act, without any particular reason for doing so. Or, in other words, without strong reasons to act or strong reasons not to act, a bias toward action leads a human to act. Everyone has these kinds of biases to some extent or another, and no negative judgment is meant by referring to humans as "biased."

In a computer security context, however, attackers will use humanity's biases against them, even if under "normal" circumstances they would be generally helpful. It does not matter if a bias leads people to make the right decision 99% of the time. If the attacker can figure out how to reliably trigger the bias for the 1% of the time it is problematic, they will use it.

In the case of action bias, the problem is that if an attacker can suggest a course of action to a victim, especially with some sense of urgency, the victim will often act *unless they specifically know they should not*. This bias is frustratingly easy to exploit and shows up in attacks like *phishing*. I will dig into this topic in more detail in Chapter 10.

Emotional Fallback

The next limitation of human psychology that is useful to manipulative attackers is what I generically call *emotional fallback*. The basic idea here is, what does a human do to make decisions when they cannot make the decisions rationally? In the previous section, we discussed that humans tend to act even when they do not have enough information to believe action is necessary. However, *emotional fallback* can drive human thinking when they *know they need to act* but do not have sufficient information. If someone knows they must make a decision but they do not have enough information or experience, how do they choose their course of action? The answer appears to be that they rely on emotional responses. For example, have you ever been in a debate with someone and experienced feelings of anger when they brought up new information that you hadn’t heard before? Many people experience this, and, when it happens, disagreements that are constructive and respectful turn combative very quickly. What seems to be happening is that when logic runs out, emotion takes over.

One of the harsh truths of emotional fallback is that *educating users may have limited value*. Ross Anderson explains the problem this way: “If the emotional is programmed to take over whenever the rational runs out, then engaging in a war of technical instruction and counter-instruction with cyber attackers is unsound, as they’ll be better at it” [40, Chapter 3]. The problem is this. There will *always* be something the average user does not know that an attacker does. Unless the user had expertise on par with the bad guys, they will have some kind of information shortfall somewhere. Whatever education or instruction is provided to the user, the attacker will simply find a new place of ignorance to which they push the user. Wherever that place is, the user will switch to emotional processing, and the attacker will be able to manipulate.

Attackers also increase their odds of success by using emotional weaponry when triggering the victim. Pretexting is the practice of calling up a target and making requests, often for information, that are not authorized. Private investigators (PI) will sometimes attempt to obtain confidential medical information about a target by calling a medical records office and pretending to be a doctor providing emergency care to the target [40, Chapter 3]. In this scenario, one can imagine all of the various emotional statements the PI could make:

1. This man is dying on my table! Get me his chart!
2. Fine, then you be the one to explain to this woman’s children why they no longer have a mother.

3. I hope you are ready for the wrongful death lawsuit! I cannot wait to testify against you.

The key idea here is not that these statements make any sense. Rationally, the person in the records office on the phone with the attacker should know that these statements are false. But if the pretexter manages to trigger an emotional response, the logical side may simply not function.

Pretexting is just one type of *social engineering*. This broader term describes any techniques for stealing information, planting false information, and obtaining unauthorized resources from convincing people that you are somebody you are not. The term is better than just calling it “manipulation” because the techniques can be both complex and sophisticated.

Using either calls (pretexting) or emails, social engineering is often about an attacker convincing the victim to trust them *just a little bit*. Once a small amount of trust is achieved, it can be leveraged. In many cases, the emotional manipulation is not the life-or-death, urgent approach I described for medical records. Instead, it is often calm, reassuring, and even jocular. One emotional weakness that can be exploited is that humans tend to trust people they perceive as being like themselves. If the attacker can trigger an emotional response from the victim that causes them to identify the attacker as “part of the group,” they are more likely to provide information [165].

So, for example, a social engineer may call up an administrative assistant and engage in small talk. Not only does the small talk put the assistant at ease, but it introduces opportunities to convince them that they are already bonded in some kind of community. Gossip is a great way to do this. The attacker may have information about an incident at the office from a news story or from some other employee. Using this information to gossip, the attacker establishes themselves as belonging. You can imagine conversation like this one:

> So, I heard what happened to Bob and Sally...I know, right? What were they thinking!...I'm sure it was a mess!...You said that to them?...That is hilarious!...

This kind of banter pushes the target into the emotional fallback. The target feels no need to ask for authorization (which is the rational decision) because the target already believes they know to whom they are talking (which is the emotional decision). Of course, if the attacker asked for something big, it might raise too many red flags, even with the emotional connection established. Often, the attacker asks for something small, perhaps just additional office information. They might try to get the name of another assistant, or the name of a computer in the building, or "unimportant" information such as the schedule for the soda delivery vendor.

Once the attacker gets information, they will typically call or email someone else and use the information they gained to be even more convincing. The more details they know about the internals of the organization's operations, the better they can pass themselves for appearing to belong when they talk to someone. Eventually, they get to the person they really want to talk to and go after what they really want. It might be a password (or a password reset), sensitive information, or even confidential company design data. Kevin Mitnick was a famous hacker that was arrested for computer crimes. He now consults as a legitimate security researcher and tells people about the tricks he used when he was the bad guy. In one incident in 1992, he stole source code from Motorola's top-of-the-line cellphone through social engineering techniques that began simply with getting Motorola's main number. He slowly built a rapport with each individual, using information from the previous encounter to make himself seem legitimate [227]. This would be equivalent to someone today phoning up Apple and stealing all the design plans to the iPhone.

Deference to Authority

The third psychological limitation discussed here is *deferring to perceived authority*. Humans will often subvert critical thinking and rational analysis if told to do some by someone perceived to be in authority. As with the other "limitations," there are probably some very good evolutionary reasons for this. This one, however, has been one of the most dangerous in human history. It has been pointed out that if one compares the private, individual violence committed throughout history to the violence committed under the direction of organized groups (such as nation states), the pain, death, and destruction from crimes like murder pale in comparison to forced relocation, war, and genocide. Oddly, murder is rejected by most societies as horrific, while destructions under orders of authority are seen as noble and honorable [271].

In cybersecurity, attackers attempt to exploit the appearance of authority at just about any level including corporate, academic, or governmental. I have received many calls from the "IRS" telling me that I have serious issues and I need to speak with an agent to discuss my case. The weird thing is that even though I *know* these are scams, I can still *feel* an urge to accept it. This partially stems from dealing with the complexities of running a small business and constantly worrying about doing the taxes correctly. My emotional response to getting one of these scam IRS calls is to worry about whether or not I did the taxes correctly. But it is enough that I can feel the urge to talk to the agent and make sure I am not in trouble. If this sounds irrational, then I am making my point. Emotional responses are not rational, and we all have them. This is one of the reasons cybersecurity is hard.

Story Time: Phishing and Impersonation
In March of 2016, a phishing scam affected Seagate Technology. The scammers were able to trick an employee at Seagate using a phishing email requesting the information which the employee believed was a legitimate internal company request. Using this, the scammers were able to obtain the W-2 information for thousands of employees, for use in filing fraudulent tax returns [155]. This incident shows how it is possible to trick people into believing a phishing email even if they have gone through training.

Scammers also play on emotional responses by claiming fraudulent authority in order to steal sensitive personal or financial information. A 2020 IRS advisory warned about scams related to Covid-19, describing schemes about fake charities, economic impact payment theft, fraudulent treatments, fraudulent investment opportunities, and so on [141]. In another example, during the Australian bushfires of 2019–2020, there was an increase in scammer activity such as fake charity scams, impersonating government entities, and relief scams [102].

Visual Emotional Responses

The last limitation I will discuss is the emotional response we feel to visual images. Neuroscientists have identified a strange case of a man with some brain damage after a traffic accident. This man recovered but experienced *Capgras syndrome*, the delusion that people are *impostors*. In particular, the man was convinced that his parents were impostors. He said they looked like his parents, but they were not his parents. This delusion was purely visual. He had no trouble accepting them as his parents when he talked to them on the phone. He was also eventually able to say he "intellectually" accepted his parents as not being impostors he had not done so emotionally [133].

The neuroscientists studying this individual have hypothesized that humans have *two* pathways through the brain for visual recognition. One is cognitive and is basically pattern matching, and the other is for our emotional response to the visual pattern. The takeaway lesson is that *humans respond emotionally to visual images*. So much so that if the emotional connection is severed or damaged, it can lead to a "split" representation in the mind. Interestingly, although the delusion of the brain-damaged man was primarily about his parents, he also occasionally split other things including countries on a map.

This kind of emotional connection to what we see might go somewhat to explaining to why fake emails are so effective. When a human sees an email that purports to be from their bank and it includes all of the graphics and symbols associated with it, it very likely is generating an emotional response of acceptance.

Story Time: Feelings Despite Knowledge

In a personal example, I ask my students every semester to create fake emails (phishing emails) for a competition. One semester, a student created a fake email purporting to be from one of my then employees in my consulting company. The email was very well done, including the employee's picture in the signature block and also including the company logo exactly in the right place. I *knew* the email was fake when I opened it, but I still had an *emotional* response to it anyway. For the briefest time, I was convinced it was real.

It should be noted that the effectiveness of visual forgeries also highlights our weakness in detecting such forgeries. This may also have an explanation in our evolutionary development, as the ability to quickly and perfectly replicate visuals, symbols, and pictures has only existed for less than a century. These simply were not threats during the thousands of years during which our ability to detect deception was developing.

Psychology-Aware Design Considerations

Security professionals must not fall into the trap of simply being dismissive of and condescending toward the cyber victims that have experienced cybersecurity failures for psychology reasons like those described in the previous sections. Unfortunately, I have known some professionals that seem to think that only an unintelligent or weak-willed human being would fall for a social engineering, fake emails, and the like. Arrogant attitudes like these fail to recognize that *everyone* is, in fact, human! All of us are vulnerable to psychological manipulation because all of us are dealing with the same inherent limitations. While it is true that with education and training we can learn to *mitigate* psychological limitations, they cannot be eliminated completely. We owe it to our fellow humans to be understanding of this reality.

Not only are condescending attitudes not very charitable, they are unhelpful at best. At worst, they are failures on the part of the security professional and designers of cybersecurity technologies to create technologies that realistically match the limitations of humanity. By way of analogy, imagine if automobile manufacturers instead of installing seat belts simply demanded that drivers did not get into accidents. After all, the vehicle is *perfectly safe* if you do not get into an accident.

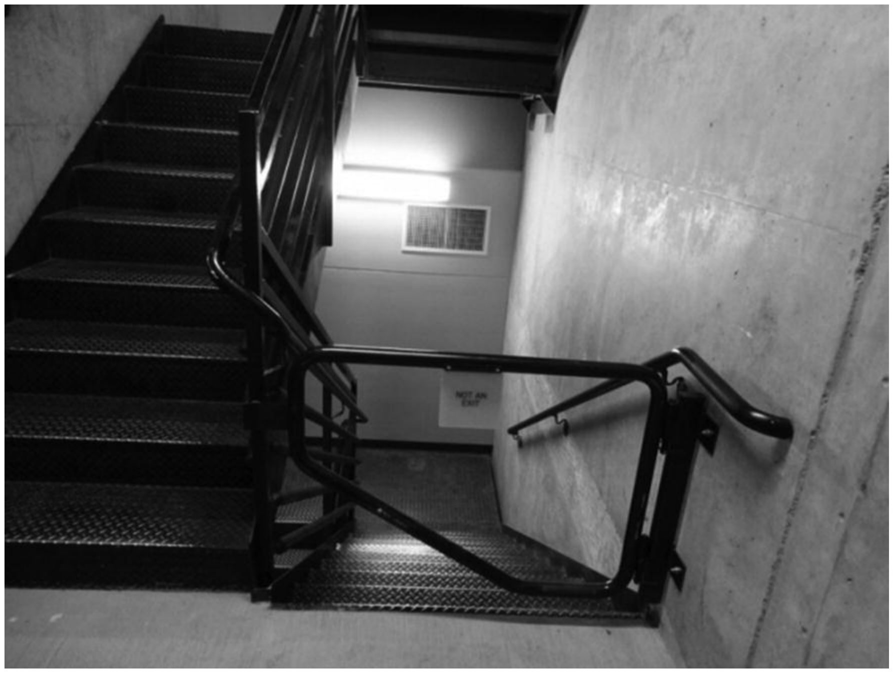


Figure 1-1 A closed gate. It does not look difficult to climb over. What is its purpose? This figure was used with permission from Pacific Stair Corporation: www.pacificstair.com

To illustrate what our cybersecurity technology needs to do, there is a very good example in the physical world of a safety device that *does* take into account human psychology. If you walk into a stairwell on the ground floor in many commercial multistory buildings, you can find a special kind of gate up against the wall (Figure 1-1).

These gates are so unobtrusive that many people are not even aware of them. In my many years of teaching, very few students have known of their existence or their purpose. These gates are *fire safety devices*. What do they do? Do they block off areas that are burning? Do they prevent people from running back in to get possessions? Do they protect sensitive firefighting controls and equipment? *No.*

Moreover, these simple gates generally do not even lock. They can usually be pulled open if you wanted to. Even if they were not locked, they can be very easily climbed over. These gates seem to be very weak from a “security” perspective.

The actual purpose of these gates is to prevent hysterical, panicking humans from running from high floors *into the basement*. In a situation that someone perceives as being immediately life threatening, the typical response is to *stop thinking* and *and start reacting*. Unless the individual has rigorous training for dealing with the life-or-death situation, the mind will shut down debate, thinking, reasoning, and planning. The lower-level instincts activate what we often call *fight or flight*. In the case of fire, flight is the only option, and the human on a higher-level floor will start running. Running *down*. The low-level brain processing apparently does not have a plan beyond running *down*. People, in this kind of panic mode, will run down the stairs and not stop until they cannot run down any further. They can run right past the ground floor level where the exit to the outside is visible from the stairwell. Obviously, this is a bad decision, to the extent a panicking human is making decisions at all.

What is the solution? Training? Should we try to train every human that will ever walk into a multistory building? Should we hand them a manual that explains to them how to react in a fire? Hopefully, it should be obvious that these are not “solutions.” Human psychology simply does not support them.

But the automatic fire gate is an amazingly simple and amazingly effective solution. The gates are normally up against the wall, but when triggered (e.g., by a fire alarm) they swing shut “blocking” access to the basement. The word *blocking* is in quotes because, as I explained, it is fairly simple to circumvent them. However, when a human is on a panic-induced run down the stairs, the gate will generally direct them out of the stairwell and out to safety.

The gate does not need to lock. It does not need to be insurmountable. It simply needs to interrupt and redirect a human survival instinct. It *assumes* the human is acting irrationally. It *assumes* that the human is not thinking. It works with this reality rather than fighting against it.

Unfortunately, cybersecurity technology has not yet caught up to this kind of psychological accommodation. It is not yet common for a security device, program, or tool to take into consideration the reality of human thinking effectively. In fairness, cybersecurity is much, much harder for the very reason I introduced at the beginning of the chapter: the bad guys are conscious, strategizing humans. A fire, as dangerous as it is, follows rules and laws. It can be predicted and modeled.

Still, the fire gate to the basement is a good model for what we want and need. We must accept that humans are human. Our technology needs to be designed with those limitations in mind. The following design principles and concepts are useful in creating such systems.

Design Principles

Affordances First, it is important to recognize just how much a design can influence human behavior and thinking. Wrong model error, discussed earlier, is related to the concept of *affordances* proposed by James Gibson, although from the positive direction. An affordance can be configured or designed to induce certain behaviors. If, for example, we build stairs into a building, people will likely use them. At the same time, affordances will impact the way the user *perceives* the system. Ross Anderson suggests that, “we design [systems] to train and condition our users’ choices...” [40, Chapter 3]. In other words, systems can be designed to help users model the system correctly.

Irrational Modes Second, like the fire gate, cybersecurity systems need to have *irrational modes* that assume users are behaving irrationally. Modern face recognition can recognize emotion. It would not be unreasonable to have a system try to identify when humans are under duress and respond accordingly. Even if the system were not automated, there should be a trigger that flips into panic mode or the equivalent. This is a significant challenge. How, exactly, do we flip a switch before someone responds to a fake email? Even if the system is put into panic mode, how should it behave? What is the right “gating” behavior to a phishing email? Although we may not have good answers for this question yet, it does not mean we should stop asking it.

Rational Centering Third, systems need to have *rational centering* means for bringing users out of emotional responses and back into rational thinking. There is some psychology research that suggests the *anterior cingulate cortex* (ACC) of the brain has components for both cognitive (logical) thinking and emotional responses. According to the *reciprocal suppression model* when the cognitive part of the ACC picks up, the emotional activity diminishes and vice versa [67]. This is consistent with observed human behavior. Humans that get whipped up into an emotional frenzy have trouble thinking rationally. On the flip side, humans experiencing an emotional overload can tone down the intensity of the feelings with walking through times tables or other rational exercises. Well-designed systems that can take into account human irrationality should also help to push them back into the thinking realm.

Error Robustness Fourth, systems must be designed with *error robustness* and assume that *humans will make* *mistakes*. Of course, this means that the system should be designed to minimize mistakes, but it should also be able to tolerate a human making an error. There are many ways to achieve this. The *principle of least privilege* teaches that a user should have the minimum amount of power in the system that is necessary for them to do their job. Minimizing what a user can do also minimizes the impact a mistake can have. A similar idea is the concept of *separating duties and concerns*. If more critical operations require two or more users, all of them must make mistakes for the operation to be exploited. It may also be necessary to design the system with layers of protection, sometimes called *defense in depth*. If done correctly, a single failure in the system will not compromise the overall security. Ideally, the failure is detected so that it can be corrected before additional failures collectively cause a compromise.

Failure Robustness Fifth, systems must be designed with *failure robustness*. Even with all of the other good design principles in place, the attackers will win sometimes. Computer security is *combat*, and sometimes the other side is going to be better, and sometimes the defenders are going to be worse. If the defenders are not prepared for losses, they will respond with bias and emotion rather than critical and effective thinking. Having effective systems for failure detection, and well-trained plans for recovery, is essential.

Manageable Decisions Sixth, and finally, security systems must be designed with *manageable decisions*. Keeping decisions manageable probably means pushing decision making to experts. This also means that managed services (security run by specialized security companies) are likely to be the future. As hammered repeatedly in this chapter, most users will not be capable of doing the security thinking required nor ready for the psychology of decision making in a security context. Or, put another way, it is hardly a fair fight to force the average user to be better at security than the average attacker. However, it is not enough to simply centralize. Centralized services have also been guilty of not thinking, and that makes things just as bad if not worse. The managed services must be able to think on behalf of the clients and not just run on autopilot.

As I have mentioned several times in this chapter, *training will almost never be the right solution to any of the problems discussed*. Generally speaking, the problems addressed in this chapter cannot be addressed with training or can only be addressed with professional levels of training. For example, automation is simply how our minds work and operate. *It cannot be trained out of us.* On the other hand, as mentioned in the section on *emotional fallback*, this happens when someone reaches the end of their knowledge or training. *The attacker will almost always know more than the average user, which means that the average user will almost always be at risk of emotional fallback.*

Interestingly, training always seems to be the fallback solution. In teaching my course to students, at least half the class will always bring up training if asked about these psychology issues on an exam. It seems to be part of human nature to believe that if we just explain something to someone enough, they will then act in a logical manner consistent with what has been taught. Accepting that humans will act irrationally really appears to be difficult for most students. However, accepting that reality is essential for developing effective cybersecurity systems.

Summary

Humans are part of cybersecurity systems. Until we develop true artificial intelligence, if we can and should, there will always be humans involved. Accordingly, humans must be understood just like any other piece of technology used in the system. It is important to understand how the human mind developed, what it can do well, and what its limitations are.

One way in which humans are different than the rest of the technology in a system is that the humans are unlikely to *change* much. We design new technologies with new features and new architectures all the time. This is not possible, of course, for our brains. What this means is that to make a system more effective, it is largely the other technologies that have to *change* to match the brain, not the other way around.

The psychology of error and the psychology of manipulation are two areas that need to be of interest to cybersecurity professionals. This chapter covered four sources of errors: *mental automation*, *complex rules*, *meta-ignorance*, and *wrong model stubbornness*. It also covered four sources of manipulation: *action bias*, *emotional fallback*, *deference to authority*, and *visual-emotional responses*.

Creating technology that effectively complements the human brain is still very challenging, and it is not done very well yet. I covered six different ideas that can be useful in developing psychology-aware cybersecurity technology. These included *affordances*, *irrational modes*, *rational centering*, *error robustness*, *failure robustness*, and *manageable decisions*.

In my opinion, creating psychology-aware systems is the greatest challenge for computer security technology in the foreseeable future. As you will see in the other chapters of this book, there are technology problems unrelated to human thinking that are sources of security vulnerabilities, but many, if not most, are human driven. It would be a wonderful new dawn for the security of cyberspace if the technology issues were the only ones, or the primary ones, that had to be dealt with.

Further Reading

This chapter touched on just a few topics of psychology related to computer security. Several portions were inspired by and based in part on Ross Anderson's chapter on Psychology in his book *Security Engineering* [40, Chapter 3]. For an overview of human error issues, Ahmed et al. have an excellent summary in "Human Errors in Information Security" [34]. I only lightly touched on action bias, but The Decision Lab has a good article about it [157]. More recently, Josiah Dykstra and Douglas Hough presented at Black Hat about how action bias causes cybersecurity problems [99].

For exploring issues of deception and social engineering, Kevin Mitnick's book is an excellent starting point [186]. Perhaps surprisingly (or perhaps not), deception is emerging as a *defense* which I touch on a little in Chapter 8. However, there is an excellent chapter on "Psychology of Cyberdeception" in Rowe and Rrushi's book on the deceptive techniques that can be used against attackers entitled *Introduction to Cyberdeception* [225, Chapter 2].

Finally, although not directly related to security, if you are interested in more about how scientists study the brain (and the mind to the extent they are different), I recommend looking into the field of cognitive neuroscience. The study cited in the chapter by Ramachandran and his colleagues is an example of work in this field. You might find Ramachandran's TED talk to be worthwhile [212].

References

34. Ahmed, M., L. Sharif, M. Kabir, and M. Al-Maimani. 2012. Human errors in information security. *International Journal of Advanced Trends in Computer Science and Engineering* 1(3): 82–87.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

41. Antonakakis, M., T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J.A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. 2017. Understanding the Mirai botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*, 1093–1110. USENIX Association.

44. Armour, P. 2000. The five orders of ignorance. *Communications of the ACM* 43(10): 17–20. [Crossref]

67. Bush, G., P. Luu, and M.I. Posner. 2000. Cognitive and emotional influences in anterior cingulate cortex. *Trends in Cognitive Sciences* 4(2): 215–222. [Crossref]

79. Community, C. 2023. 3cx desktopapp security alert.

80. Community, C. 2023. Threat alerts from sentinelone for desktop update initiated from desktop client.

86. CrowdStrike. 2023. Crowdstrike prevents 3cxdesktopapp intrusion campaign.

99. Dyskstra, J., and D. Hough. 2021. Action bias and the two most dangerous words in cybersecurity.

102. Elsworthy, E. 2020. Australia fires see spike in fraudster behaviour.

123. Guerrero-Saade, J.A. 2023. Smoothoperator: Ongoing campaign trojanizes 3cxdesktopapp in supply chain attack.

127. Hanson, R. 2010. Confronting the negativity bias.

133. Hirstein, W., and V.S. Ramachandran. 1997. Capgras syndrome: A novel probe for understanding the neural representation of the identity and familiarity of persons. *Proceedings. Biological Sciences* 264: 437–444. [Crossref]

141. Internal Revenue Service. 2020. Irs warns against covid-19 fraud; other financial schemes.

155. Krebs, B. 2016. Seagate phish exposes all employee W-2s.

157. Lab, T.D. Why do we prefer doing something to doing nothing? The action bias, explained.

165. Luo, X., R. Brody, A. Seazzu, and S. Burd. 2011. Social engineering: The neglected human factor for information security management. *Information Resources Management Journal (IRMJ)* 24(3): 1–8. [Crossref]

180. Merriam-Webster. Adversary.

186. Mitnick, K.D., and W.L. Simon. 2003. *The Art of Deception: Controlling the Human Element of Security*. Wiley.

190. Munson, R.V. 2001. Telling wonders: Ethnographic and political discourse in the work of herodotus. [Crossref]

195. Norman, D. 1983. Design rules based on analyses of human error. *Communications of the ACM* 26: 254–258. [Crossref]

211. Rachlin, H., and B.A. Jones. 2008. Social discounting and delay discounting. *Journal of Behavioral Decision Making* 21(1): 29–43. [Crossref]

212. Ramachandran, V.S. 2007. VS Ramachandran: 3 clues to understanding your brain.

219. Ritchie, H., and M. Roser. 2019. Causes of death.

225. Rowe, N.C., and J. Rrushi. 2016. *Introduction to Cyberdeception*, 1 ed. Springer International Publishing Switzerland. [Crossref]

227. Sagarin, B.J., and K.D. Mitnick. 2012. The path of least resistance. In *Six Degrees of Social Influence: Science, Application, and the Psychology of Robert Cialdini*, ed. D.T. Kenrick, N.J. Goldstein, and S.L. Braver, chapter 3, 27–38. Oxford University Press. [Crossref]

234. Schaller, M., J. Park, and J. Faulkner. 2003. Prehistoric dangers and contemporary prejudices. *European Review of Social Psychology* 14(1): 105–137.

[Crossref]

254. Stueh. 2023. Comment on: [3cx breach update].

261. The PC Security Channel. 2023. 3cx: How this malware almost hacked every business.

271. van der Dennen, J., K. Thienpont, and R.L. Cliquet. 2000. Of badges, bonds and boundaries: Ingroup/outgroup differentiation and ethnocentrism revisited. In *In-group/out-group behaviour in modern societies: An evolutionary perspective*, NIDI/CBGS Publications, ed. Robert Cliquet and Kristiaan Thienpont, 37–74. Nederlands Interdisciplinair Demografisch Instituut (NIDI); Centrum voor Bevolkings- en Gezinsstudien (CBGS).

276. Vijayan, J. 2023. 3cx breach widens as cyberattackers drop second-stage backdoor.

Footnotes

1 I recognize that cancer is lethal and claims millions of lives every single year. None of the comparison to cancer is meant to minimize the suffering or loss experienced by anyone that has been impacted by its terrible destruction. In fact, I explicitly chose cancer as the comparison subject precisely because it is so damaging and destructive. Cancer is destructive *without* being conscious, self-aware, or intelligent. The comparison here is to help the reader understand just how much more terrifying the disease would be if it were and to illustrate just how dangerous our cybersecurity adversaries are.

2 Unless, of course, we really do develop some kind of independent artificial intelligence...but that might come with its own problems.

3 Being killed by animals is still a relatively large risk, but this includes, for example, mosquitoes. But prehistoric humans could not run away from them either. On the other hand, the data shows that approximately 100 people were killed by lions in 2017 [219].

4 Although the discussion thus far has referred to evolution in the biological sense, the slowness of “evolution” also includes the evolution of society. Biological evolution moves so slowly that no significant changes to the human brain can be expected for thousands of years. The evolution of social interactions moves faster as societies learn, experiment, and adapt to enable

better cooperation and also to protect themselves from threats. However, the speed of human technological progress has been moving so rapidly that even social evolution cannot keep up.

5 One of the real challenges with healthy, respectful discourse in politics, religion, and other sensitive subjects is an inability to figure out “the other side’s” modeling. If you find there are a large number of people that seem to take a point of view you just cannot understand, it may be worth exploring the models that you and they are using. Working to understand and explain your model, and figure out theirs, can lead to better mutual understanding and an improvement in working with others.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_2

2. Authentication Technology

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
In this chapter, you will learn about how parties (typically human parties) can be identified to a computer system. This process is called *authentication*. For identifying a human party, there are three common approaches: *something you know* (like a password), *something you have* (like your phone), or *something you are* (like a fingerprint). Although it is popular to talk about passwords as "weak" and other mechanisms like biometrics as "strong," every approach has pros and cons, strengths and weaknesses.

Key Concepts

1. In cybersecurity, *identity* is typically just a unique sequence of characters assigned to a party.
2. Authentication by *something you know* requires the secret be known by the proper party and *only* the proper party.
3. Authentication by *something you have* requires a unique token be held by the proper party and *only* the proper party; many something-you-have approaches can only simulate a unique token.

approaches can only simulate a unique token.

4. Authentication by *something you are* has a wide range of requirements including statistics requirements, privacy requirements, false positive/false negative requirements, and others.
5. Multifactor authentication, when done properly, combines two or more of the authentication approaches.

Common Pitfalls and Misunderstandings

1. It is challenging to create passwords that are both memorable to the authorized party and hard to guess by unauthorized parties.
2. Many of the practical approaches for something-you-have authentication can only *simulate* a unique device.
3. Many organizations that deploy biometrics operate with overconfidence creating a culture that is vulnerable to attack.
4. All authentication approaches can break down if edge cases, such as password resets, are not carefully designed and controlled.

Useful Vocabulary

- **Principal**: Any entity that is unique within the context of the system
- **Hashing Function**: A function that produces a small, unpredictable mathematical code for any amount of data

- **Hash**: The output of a hashing function that can be thought of as a kind of “fingerprint” for the data that was hashed
- **Brute Force**: Any approach to cracking cybersecurity systems that entails trying all possible combinations
- **Entropy**: A measure of the unpredictability of a key or password, or the amount of unique information contained in it, as a proxy for how difficult it will be for an attacker to guess it
- **False Positive**: Incorrectly confirming invalid data is valid
- **False Negative**: Incorrectly rejecting valid data as invalid

---

With a few notable exceptions, *authentication* is an essential component in almost all computer security systems. Authentication is part of the larger problem of *identity management*. Identity management relates to “storage, processing, disclosure and disposal of users’ identities, their profiles and related sensitive information” [55]. There is no one definition of identity management, but over the decades, there have been a range of approaches to the broad concept that have been variously described as *Authentication, Authorization, and Accounting* (AAA) [182]; *Authentication, Authorization, and Audit* [166];[1] *Identity and Access Management* (IAM) [260]; or *Identity, Credential, and Access Management* (ICAM) [251]. These are not synonyms or even necessarily alternatives. Rather, they illustrate how authentication is almost always part of a larger combination of access technologies. In this book, I primarily focus on just authentication and authorization. This chapter covers some of the foundational ideas behind authentication. Authentication is defined by the National

Institute of Standards and Technology (NIST) as, “Verifying the identity of a user, process, or device, often as a prerequisite to allowing access to resources in an information system” [223, 257]. In other words, authentication is how a system “knows” who it is dealing with. This is in contrast to authorization which deals with deciding whether this person (identified through authentication) should be granted access. We will discuss this more in the next chapter.

Hundreds of millions of people are being authenticated every day. Many of them are being authenticated repeatedly throughout the day. It is not uncommon for a person living in the developed world to authenticate before accessing their bank on the Internet, when they use their work computers during the day, and in order to play a movie or video game in the evening. Add to this the repeated uses of a personal device, such as a phone, and authentication is almost constant throughout the day.

Despite its ubiquity, authentication can be quite problematic. Because it is used everywhere, it is incorporated into almost all modern computers and computer systems. Nevertheless, it is often misconfigured, mismanaged, or not appropriate for the context in which it operates. Even when the technology is correctly used, more fundamental problems drive problems that cannot be solved with technology alone.

The goal of this chapter is to provide you with a starting point for understanding core concepts that shape all authentication technologies, proper deployment in correct contexts, and inherent limitations that cannot be eliminated (despite vendor claims) and must be dealt with by the users.

Foundations of Authentication

A good starting point for learning about authentication is "identity." Since the goal of authentication is to verify identity, it is essential to understand what identity is.

It is also important to understand what it is *not*. Identity in terms of authentication technology is not necessarily the same as identity as it is used in common parlance (at least in the English language). The word, as defined in the dictionary, means "the condition of being oneself or itself, and not another," or "condition or character as to who a person or what a thing is; the qualities, beliefs, etc., that distinguish or identify a person or thing" [181]. Even in terms of cybersecurity, the average user probably associates identity with crimes like "identity theft," wherein a person's identity within society is taken or used by someone else.

In contrast, identity in authentication does not necessarily have anything to do with who or what a person (or thing) is. It may not be related to their real name or real self. It may not even be related to their nature as a *person* at all. In fact, an identity may not be representative of anything in the physical world at all.

What *is* an identity then? Usually, it is just a sequence of symbols, such as letters and numbers. In computer terms, this is called a "string" (originally, "a string of characters"). So, for example, an identity might be a username, an email address, a URL, or a phone number. These strings may, in fact, be related to some real-world counterpart, but from the perspective of the authentication system, it is just a unique sequence of symbols.

What is important about an identity is whether or not a party may *claim* it. Consider, for example, your username on a computer you use. There was some process by which you received permission to use that name. It may have been assigned to you, or you may have registered it, but you had that identity associated with you. Nobody else, even if they know your identity, should be permitted to claim it.

An identity can be claimed by a flesh-and-blood human, of course. But an identity can also be claimed by an organization, a group of people, someone acting in a particular role, a computer, a program, a system, and so forth. The term *principal* is used to describe any entity that is unique within the context of the system. Principals may claim an identity. Clearly, there should be a unique identity for a unique principal within a given system.

With that said, the three approaches discussed herein are widely used for humans to identify themselves to a system. While passwords can be used by machines to identify themselves, tokens and biometrics really cannot. Moreover, there are other approaches for machines to identify themselves that are discussed in other chapters. For the purposes of this chapter, you may assume that a *principal* refers to a human principal. For this reason, when speaking informally, I will sometimes use the term “user” as a generic synonym for principal.

Accordingly, the general approach to authentication follows a very basic formula:

- The principal requesting authentication claims to have or control an identity string.
- The system performing authentication demands proofs of the claim.
- The principal requesting authentication provides the demanded proofs.

Figure 2-1 shows a generic illustration of this process.

In practice, the types of proofs used to validate an identity claim fall into one of three categories:

- Something you know
- Something you have
- Something you are

Authentication succeeds when the principal with the valid identity claim is correctly validated and when any other principals (which may include attackers) with an invalid claim are not validated. A *false positive* occurs when an invalid claim is accepted, and a *false negative* occurs when a valid claim is rejected. For example, if you try to access your phone using your fingerprint scanner and it cannot recognize you, that is a false negative. If your friend is able to access your phone using their fingerprint, that is a false positive.

Authentication also fails when the wrong party gets a hold of proofs used for validating identity. The proofs provided to prove identity are called *credentials*, and certain types of credentials can be stolen. Protecting a principal's credentials often requires a certain amount of correct use on the part of the authentication system and the principal seeking authentication.

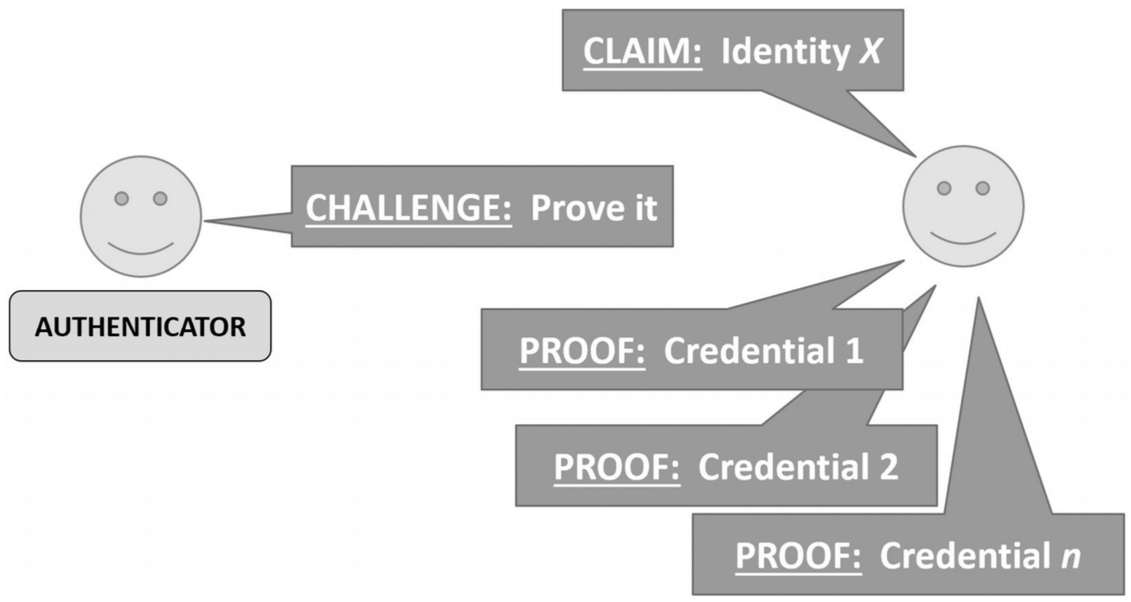


Figure 2-1 A general depiction of authentication. This process is not that different from physical-world authentication. When you arrive at a hotel, for example, you tell them your name. They ask to see your ID which you present as your credential. Sometimes, you need to have more than one form of ID (credential)

As you will learn throughout this book, security components, such as authentication, do not exist in a vacuum. Authentication, for example, will generally be part of a system and must be integrated into the system correctly. One simple example is that authentication controls must be present on *all* the different entrances into the system. It is not good to require identification at one gate, while another gate is left completely open. This concept is sometimes captured as the principle of *Complete Mediation*.

Something You Know

Authentication using "something you know" requires that the principal seeking authentication prove to the authenticating system that they know a secret. The most common form of this authentication approach is passwords.

Passwords have been used for centuries and long before computers were around. A popular story from the *Arabian Nights*[2] is "Ali Baba and the Forty Thieves"[3]. Ali Baba overhears thieves entering their secret treasure room with the magic words "Open Sesame." After they leave, Ali Baba is able to enter the room using the same *password*. Authentication, in this case, failed. Ali Baba was able to enter an unauthorized location using a stolen password.[4] Stolen passwords are a good segue into the three requirements necessary for passwords to be an effective authentication technology.

Password Requirement 1: Exclusive Knowledge. The password must be known *only* by the party with the valid identity claim. The password is meant to prove that the knowledgeable party, and only the knowledgeable party, has claim on the identity. As soon as the password is known by multiple parties, it is no longer a viable authentication mechanism. In an ideal world, passwords should not be shared with coworkers, close friends, or even domestic partners. There may be circumstances where there is no other choice but to share, but this should be the exception, and not the rule, and it should only be for relatively low security systems. Passwords should also not be shared because your friend or coworker “promises to keep it secret.” This is well illustrated in the story of Ali Baba; as soon as “Open Sesame” was known by Ali Baba, it was no longer an effective way of authenticating the leader of the thieves.

Three can keep a secret, if two of them are dead.

— *Benjamin Franklin* [110]

Password Requirement 2: Unguessable. The password cannot be “easily” guessed by humans or computers. This requirement is one of the most misunderstood by the average user and the biggest problem for password-based systems. According to a 2017 report by Verizon, more than 80% of all cybersecurity breaches were due to stolen *or weak* passwords [275]. Although they did not break it down further, weak passwords (i.e., passwords that can be guessed) are undoubtedly a significant portion.

Password Requirement 3: Unforgettable. The password must not be forgotten by the principal seeking authentication. Forgetting the password obviously results in being unable to authenticate. The “solutions” commonly used, such as writing the password down or using a password reset, introduce significant security issues that I will detail later in this chapter. As a fun illustration of the importance of an unforgettable password, in the movie version *Arabian Nights* that I like so much, Ali's brother Kasim goes back to the cave. To help him remember the password, Ali has given him sesame biscuits. But he loses the biscuits after getting into the cave; he forgets the password and can no longer get out. Unfortunately for Kasim, he is trapped there until the bandits find (and kill) him.

Passwords are widely used partially because of how easy they are to set up. No special infrastructure is required. A user thinks up a password and that is all there is to it. Something-you-have and something-you-are authentication always requires more and more expensive configuration and setup. Passwords are also intuitive for users and require almost no training to use, even if used incorrectly.

But this easy deployment is probably the only reason it is still used. Passwords are a terrible form of security in almost every way. Every security expert and a pretty good number of non-experts hate passwords. Some of the many password problems will be detailed in the following subsections.

Password Verification and Storage

For a user to authenticate using a password, the system performing the authentication must know the user's password. This is a security risk that should not be overlooked. Although it is generally necessary,[5] the risks must be understood and managed.

Typically, a user has some form of a registration phase. A typical design for the registration process is depicted in Figure 2-2. During this phase, the user submits their password along with whatever identity they have chosen or been assigned. The password is received at the authentication system where it is processed and stored along with the identity string. For security reasons, the password should not be stored "raw." Instead, a special derivative of the password, called a "hash," is stored.[6]

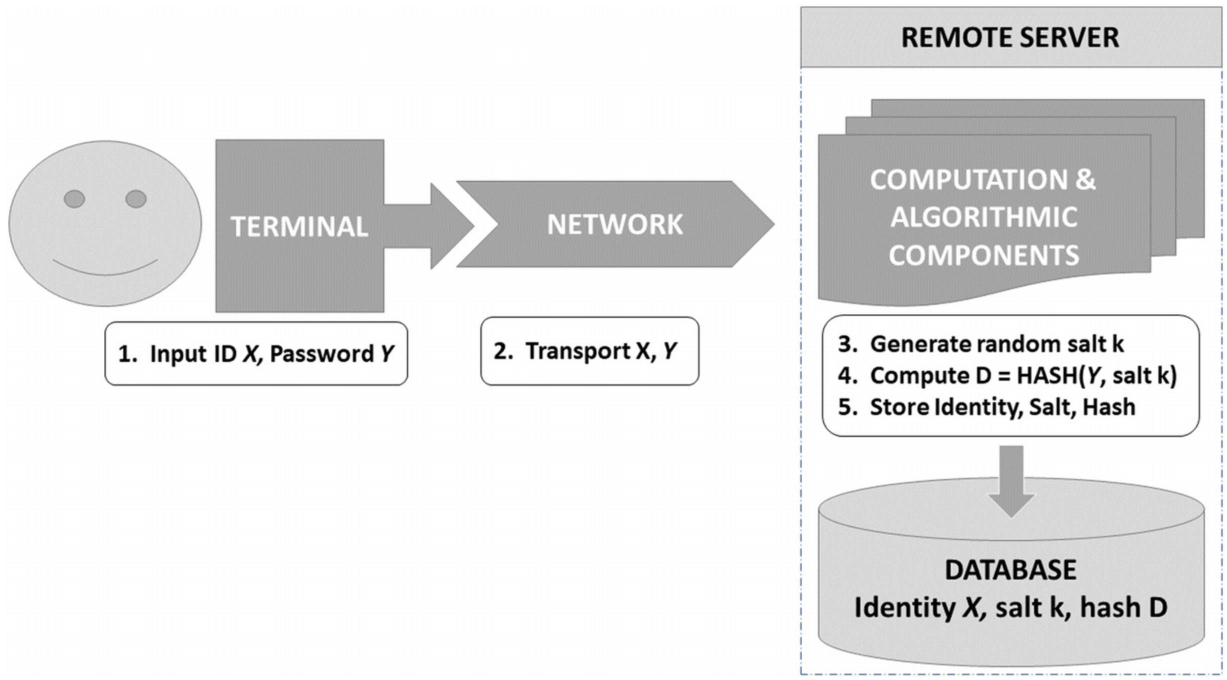
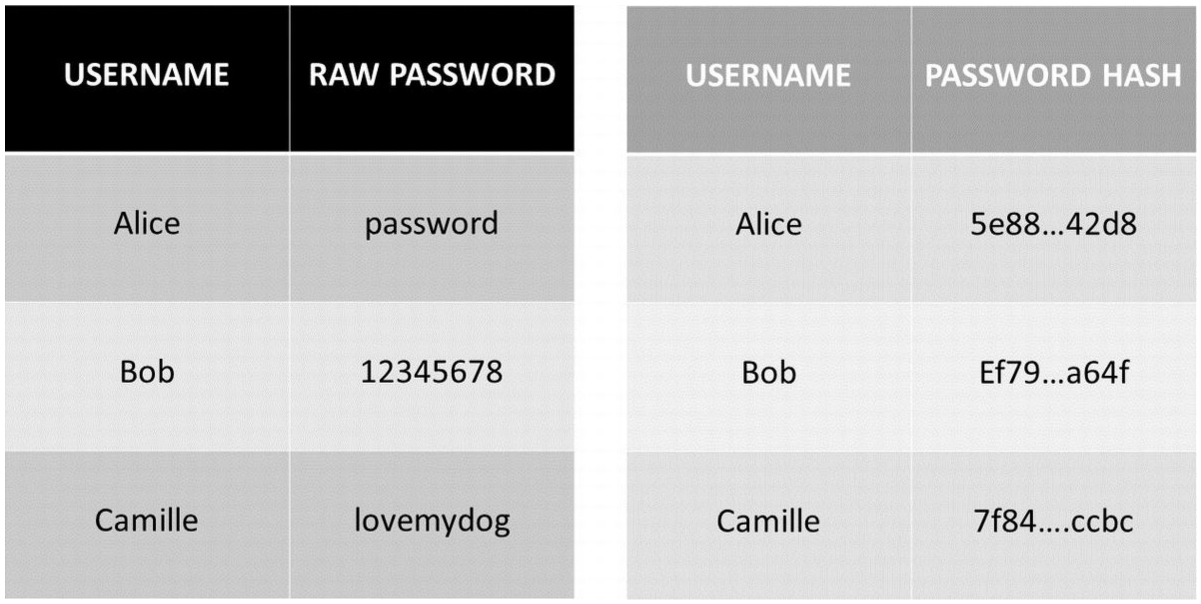

REMOTE SERVER
TERMINAL
NETWORK
COMPUTATION & ALGORITHMIC COMPONENTS
1. Input ID *X*, Password *Y*
2. Transport X, *Y*
3. Generate random salt k
4. Compute D = HASH(*Y*, salt k)
5. Store Identity, Salt, Hash
DATABASE
Identity *X*, salt k, hash D

Figure 2-2 Password registration starts with a user submitting an identity (e.g., username) and a password. The system that receives the data generates a random value called "a salt" and generates a hash of the password and salt. The identity, salt, and hash of the password are stored in a database. The important point is that the password itself is not stored



USERNAME	RAW PASSWORD
Alice	password
Bob	12345678
Camille	lovemydog

USERNAME	PASSWORD HASH
Alice	5e88...42d8
Bob	Ef79...a64f
Camille	7f84....ccbc

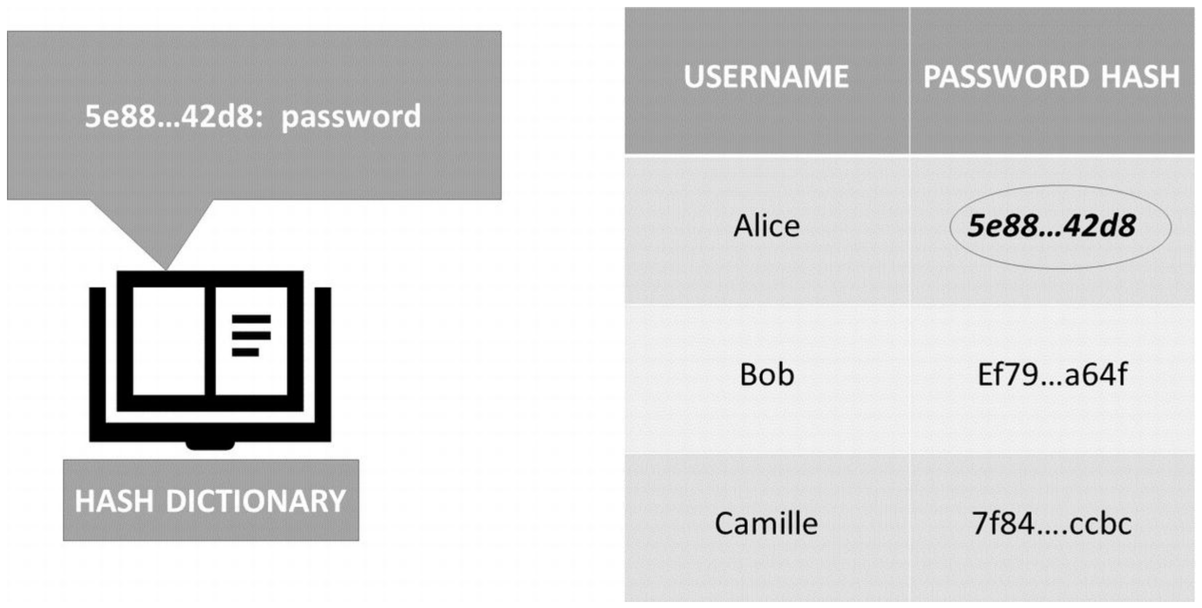
Figure 2-3 Storing a username with the raw password, as shown on the left, is not good security. If an attacker breaks into the server and steals the password database, they immediately have all of the usernames and passwords. On the other hand, if passwords are hashed, as shown on the right, they have what looks like completely random data. These password hashes can still be “cracked,” but it can take time and, for good passwords, it will probably take too long to be of much use to the intruder

You will learn much more about hashes in Chapter 5. For the purposes of this section, you only need to know that a hash is a “one-way function.” The hash function takes the password as an input and spits out what appears to be a random sequence of data. It *is not* random, of course. The same password input to a hash function always results in the same output every time. But the output itself looks random and has nothing of the original password in it. It is effectively impossible to examine the hash output and know what the original password was that generated it. This is why it is called a “one-way function.” It is easy to take the password and get the hash, but it is difficult to take the hash and figure out the password.

Because the authentication system stores the hash of the password, rather than the password itself, the user has some protection should an attacker break into the authentication system and try to steal the password, as illustrated in Figure 2-3. The would-be Ali Babas of cyberspace will only find the hash of “Open Sesame”[7] stored in the system, making it harder for them to steal.

Additionally, the password is hashed with randomly generated data called a *salt*. The salt is not secret and is stored in the database with the username. By mixing in this random data, the hash output will always be unique. This is important for two reasons. First, if two users accidentally (or on purpose) chose the same password, the hashes would be the same for each user. If an attacker broke into the system and stole the password file, they could instantly determine that the two users share the same password.

Now, social engineering attacks like those described in Chapter 1 can be used against either target to steal the password of both.



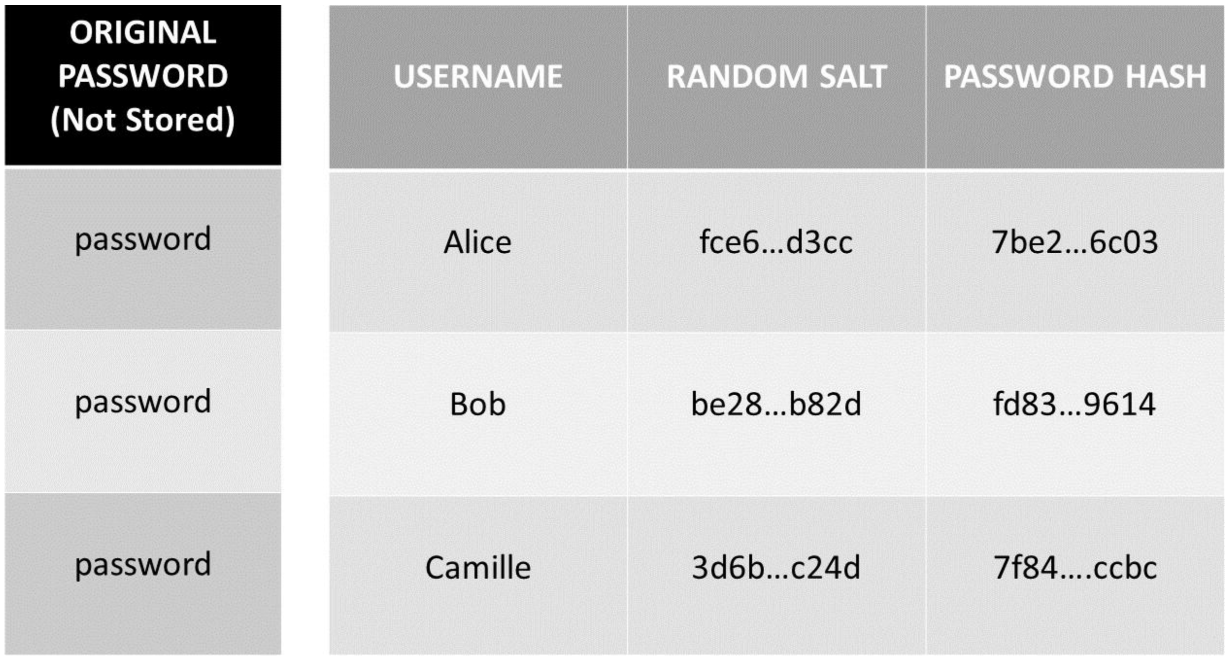
USERNAME	PASSWORD HASH
Alice	*5e88...42d8*
Bob	Ef79...a64f
Camille	7f84....ccbc

Figure 2-4 In this example, passwords are not hashed using random salts. An attacker can easily crack common passwords, such as dictionary words, by taking a dictionary and hashing each word ahead of time. This creates a dictionary of hashes to words that can be easily consulted to crack the password

A more important reason for using salts is to try and protect users with weak passwords. Remember that the hash of a password is always the same for a given hash algorithm. If salts were not used, attackers could just have entire dictionaries of hashed words. Every word would be prehashed. If the user's password is "password" (still commonly used!), the hash stored in the database would be immediately found in the prehashed dictionary, as illustrated in Figure 2-4. On the other hand, Figure 2-5 shows how storing the hash of a password mixed with a salt is unique for each user.

Once the user has registered, they can authenticate by a similar process as shown in Figure 2-6. The user submits the claimed identity (i.e., username) and the password. As during registration, this data is transmitted to the authentication system. When it arrives, the authentication system looks up the username in the database and extracts the stored password hash and salt. Remember: The password itself is not stored, only the hash of the password. To authenticate the user, the transmitted password is hashed (with the salt), and this value is compared to the hash stored in the database.

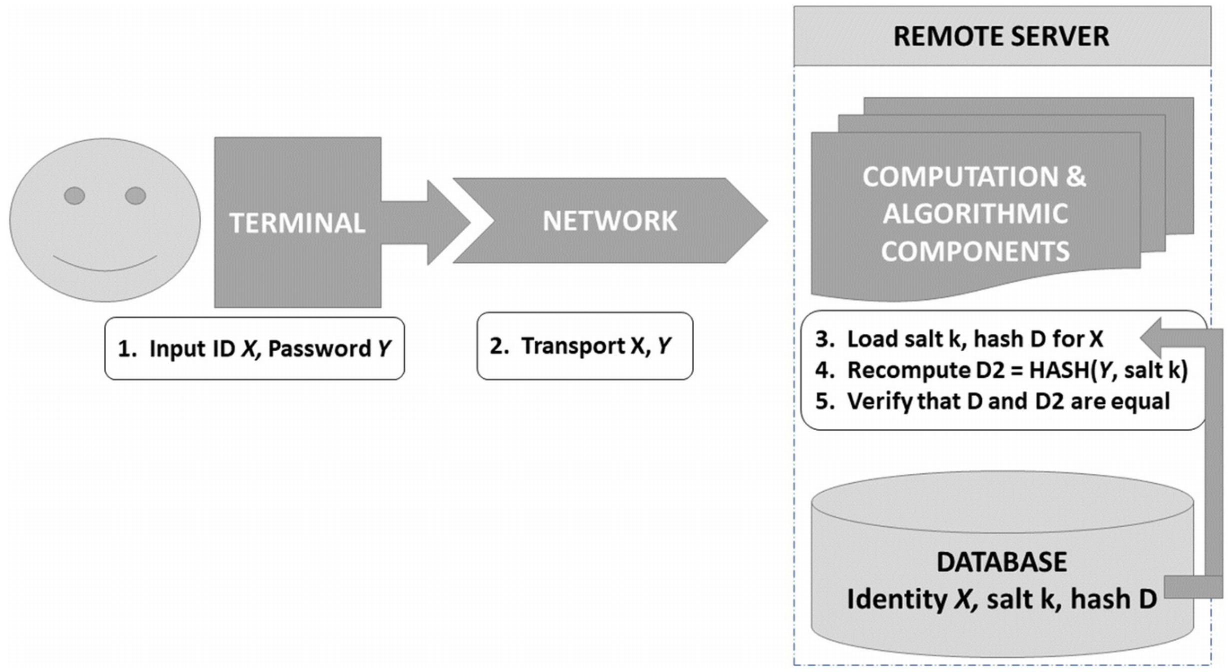
Attackers can try to steal passwords by compromising the system at any location. For example, if the attacker has compromised the user's machine with evil software ("malware"), they may be able to capture the password as it is being typed. "Keyloggers" are a type of malware that record every key pressed on the keyboard and then exfiltrate this information to the attacker. Malware will be discussed in greater detail in Chapter 7.



ORIGINAL PASSWORD (Not Stored)	USERNAME	RANDOM SALT	PASSWORD HASH
password	Alice	fce6...d3cc	7be2...6c03
password	Bob	be28...b82d	fd83...9614
password	Camille	3d6b...c24d	7f84....ccbc

Figure 2-5 The hash of a password might look completely random, but it is not. The hash of "password" is always the same for a given hashing function. If multiple users all decided to use the same (bad) password, their hashes would all be the same. If an attacker stole the password file, they would immediately know that the users have the same password, which is a problem all by itself. But worse, it permits the precomputed hash dictionaries like the one depicted

in Figure 2-4. In this example, however, the three users, even though they choose the same bad password, all have different hashes because a different random value (salt) was mixed in with each one. Not only does the attacker not instantly know the passwords are the same but each password must be cracked separately using the individual random salt



REMOTE SERVER
TERMINAL
NETWORK
COMPUTATION & ALGORITHMIC COMPONENTS
1. Input ID *X*, Password *Y*
2. Transport X, *Y*
3. Load salt k, hash D for X
4. Recompute D2 = HASH(*Y*, salt k)
5. Verify that D and D2 are equal
DATABASE
Identity *X*, salt k, hash D

Figure 2-6 When a user logs in to a system, they transmit their identity (e.g., username) and password, just like they did during the registration phase. But when the username and password are received, the server takes the identity and loads the original salt and hash out of the database. To verify that the user entered the right password, the transmitted password is hashed with the salt to produce a new hash. The new hash is compared to the stored hash from the database. If they are the same, the user entered the correct password

Passwords that are transmitted over the Internet can be intercepted en route unless they are transmitted over a secure channel. Much of the data sent to websites is transmitted using the HyperText Transfer Protocol (HTTP). HTTP can be used in an unprotected form and a secure form, the latter of which is identified by HTTPS. Fortunately, most websites have been switching over to HTTPS and do not even provide an unprotected version. Nevertheless, some websites still use the unprotected HTTP, and no sensitive data should be submitted to such sites. Other mechanisms for securing data transmissions include Virtual Private Networks (VPNs). I will discuss these issues in greater detail in Chapters 8 and 9. To illustrate, again I suggest watching *Arabian Nights*. Ali Baba is shown overhearing the password while hiding behind a rock. Stealing the password is easily achieved by simply being in the right place and listening in. The same thing can be done on modern networks. But, fortunately in the modern world, the password is transmitted through the computer networks over an encrypted channel. Attackers that are listening in cannot tell anything about the data, and, in fact, it will look random. It would be equivalent to Ali Baba overhearing static.

Finally, the attacker *can be the authentication system itself*! I hinted at this earlier when I mentioned the security issues of sharing a password with an authentication system in the first place. All of the security described so far *presumes that the authentication system is honest.* If the authentication system is malicious, there is nothing to stop it from revealing the user's password to other parties. Many users choose the same password for many of the websites and systems that they use. A malicious authentication system could use whatever username and password were submitted to itself and try those same credentials on another system, such as a bank. In the United States, three banks hold about 30% of all deposits. Suppose that just 100 Americans register with an evil website. Statistically, about one in three of these users will have an account with one of these three banks. The evil website can take each user's username and password and test it out on each one of them. There is a good chance that, for at least a few of the users, they will have reused the same password. The username might also be the same, can be guessed, or otherwise determined. This attack becomes even easier if the user is willing to give the evil website credit card info, because it reveals at least one bank with whom the user has an account.

Story Time: Three Times Is a Conspiracy
Vox writer Sara Morrison discusses how her financial life was upended when she was hacked at Grubhub first, then banks and credit cards later. In total, hackers got away with $13,000 before she realized how serious the problem had become. Although she is not absolutely certain of it, she believes the reason she suffered breaches going over many months across multiple accounts was because she had been using the same password for most of them. This is very likely. Learning from her mistakes, she installed a password manager and created a new password for every account [187]. It would be a good idea to follow her example.

For this reason, passwords should not be reused between different systems. Most users find it difficult, if not impossible, to remember a different password for every system. One solution to this problem is to use a password manager. Because password managers also solve other problems addressed later in the chapter, I will defer discussing them in detail for now.

Speaking of other problems, attackers can also attempt to “crack” a password hash. This is the topic of the next section.

Cracking Stored Passwords

Even if passwords are not stored in their raw form, weak passwords can still be cracked. Let us assume that an attacker has penetrated a website and stolen its password database. Inside the database, the attacker finds the usernames and the password hashes. Given that hashes are one-way functions, the attacker should not be able to figure out the original passwords, right?

Not exactly. The attacker can *guess*. The attacker can hash any given guess and see if it matches what is in the stolen database as illustrated in Figure 2-7.

As I discussed earlier, passwords should not be easily guessable by either humans or computers. The type of guessing that each can do is different and based on different strengths. Humans' guessing is good at using semantic information. Humans can figure out the names of family members, friends, pets, important dates, sports teams, colleges, high school, and so forth. Users creating passwords based on this information seem to assume that "bad guys" will not be able to find out personal information even if friends, family, and perhaps even coworkers know all of these details. These assumptions are all wrong as friends, family, and coworkers can *be* the attackers. More importantly, personal details are extremely easy to find even for a stranger. Personal information should not be used as a password.

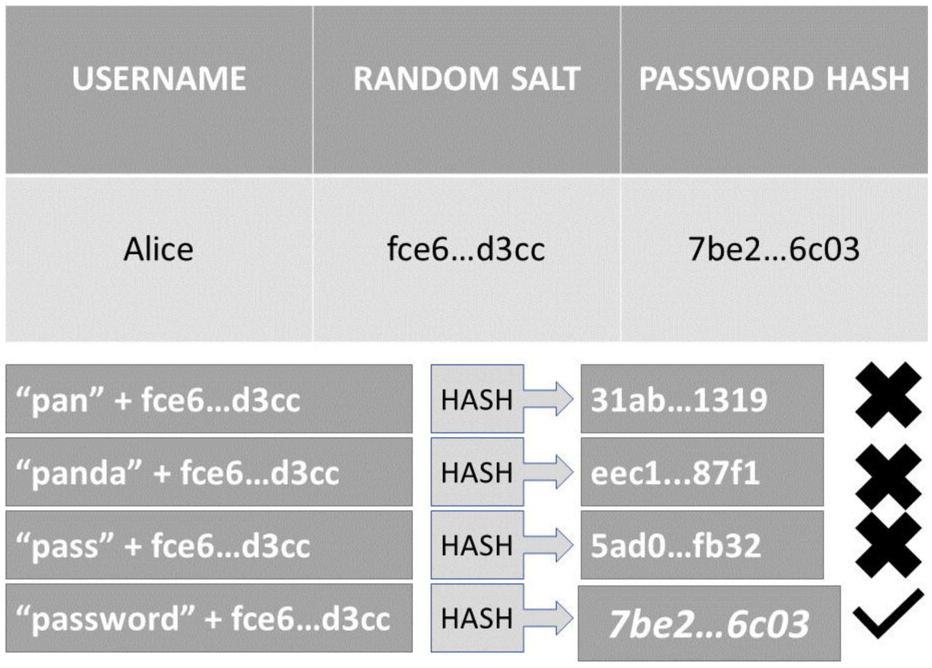


Figure 2-7 An attacker cracks passwords by guessing a possible password, mixing it with the random salt, and hashing the guess and the salt together. If the output of the hash matches the value in the password database, the attacker has found a correct password

Humans are also good at guessing phrases that a user might like such as “to be or not to be!” Not only are these phrases easily guessed but so are derivatives such as using the first letter of each word (e.g., “tbontb”) or replacing a letter or word with a number (e.g., “2b0ntb!”). Computers can guess these too, but the difference here is that a human can guess phrases based on their knowledge of the victim’s preferences.

Computer-based guessing on the other hand is based on huge amounts of trial and error. This is what a computer is good at. Computers do not get tired or bored, do not need to stop for food or rest, and can operate in extremely methodical ways. Humans without training in computer security often have poor intuition as to just how capable computers can be at trying millions of combinations.

Two of the most common ways that computers guess passwords is *brute-force* and *dictionary* based.

Brute-force guessing involves trying every single combination of every single symbol that could be in the password. This is hard for humans (perhaps impossible in any reasonable time) but easy for computers. I talked about why *salting* the password was so important in the previous section. Were it not for salting, attackers would only have to precompute all possible dictionary words once. By using a salt, the attacker must try hashing each word in the dictionary with the random salt mixed in. And the attacker must do this *for every user* as every user has a unique salt. This is the starting point for our discussion.

Even still, making a lot of guesses, even for each individual user, is straightforward for a computer. A very simple program can systematically try hashing every possible combination of letters, numbers, and special characters in every possible length. It can start, for example, with all combinations of one-letter passwords, then advance to two-letter passwords, and so forth. The only limitation is *time*. No matter how fast computers get, there will always be a limit to how many guesses can be tried in “reasonable” time. Passwords should be long enough that a computer cannot try all possible combinations of that length without running out of time.

It is probably intuitive that with more symbols there can be more passwords. But to make it more clear, let us walk through some calculations.

Suppose that a user only uses lowercase letters from the English alphabet in their password. How many possible *one-letter* passwords are there? Twenty-six because there are twenty-six lowercase letters in the English alphabet. If a user’s password was just one letter, it would take, at most, twenty-six guesses to figure it out. The attacker would hash each one of the twenty-six lowercase English letters and see if the hash of the letter matched the hash in the database.

As a side note, the output of a hash is always a fixed size no matter how big the input is. A twenty-character password and a one-character password have the same size hash for the same hash function. It is impossible for the attacker to look at the hash and know (or even guess) how big the original password is.

Now imagine that the user’s password is two lowercase letters. How many possible passwords are there now? There are still 26 possibilities for the first letter, but for each first letter, there are 26 possibilities for the second letter. That is, if the first letter is “a,” then there are 26 possible passwords that can be chosen as there are 26 possible choices for the second letter. If the first letter is “b,” there are *another* 26 possible passwords. And so on. What this means is that there are 26 times 26 possible passwords for a two-letter password. That is 676 passwords.

Thinking it through, you can see that for an all lowercase password using the English alphabet, the number of possible passwords is 26^n where *n* is the number of letters in the password. The following table illustrates how the number of passwords increases with length:

$$26^1 = 26$$
$$26^2 = 676$$
$$26^3 = 17{,}576$$
$$26^4 = 456{,}976$$
$$26^5 = 11{,}881{,}376$$
$$26^6 = 308{,}915{,}776$$
$$26^7 = 8{,}031{,}810{,}176$$
$$26^8 = 208{,}827{,}064{,}576$$
$$26^9 = 5{,}429{,}503{,}678{,}976$$
$$26^{10} = 141{,}167{,}095{,}653{,}376$$
$$26^{11} = 3{,}670{,}344{,}486{,}987{,}776$$
$$26^{12} = 95{,}428{,}956{,}661{,}682{,}176$$
$$26^{13} = 2{,}481{,}152{,}873{,}203{,}736{,}576$$
$$26^{14} = 64{,}509{,}974{,}703{,}297{,}150{,}976$$
$$26^{15} = 1{,}677{,}259{,}342{,}285{,}725{,}925{,}376$$

This illustration shows why the length of a password is so important. For every one character added to a password, even just a lowercase letter, the attacker has to try *26 times* more guesses.

You might have already figured out why it is so important to use more than just lowercase letters. Look how everything changes with lowercase and uppercase letters. Again, using the English alphabet, there are now 52 possible symbols for each spot in the password. The math now looks like this:

$$52^1 = 52$$
$$52^2 = 2{,}704$$
$$52^3 = 140{,}608$$
$$52^4 = 7{,}311{,}616$$
$$52^5 = 380{,}204{,}032$$
$$52^6 = 19{,}770{,}609{,}664$$
$$52^7 = 1{,}028{,}071{,}702{,}528$$
$$52^8 = 53{,}459{,}728{,}531{,}456$$
$$52^9 = 2{,}779{,}905{,}883{,}635{,}712$$
$$52^{10} = 144{,}555{,}105{,}949{,}057{,}024$$
$$52^{11} = 7{,}516{,}865{,}509{,}350{,}965{,}248$$
$$52^{12} = 390{,}877{,}006{,}486{,}250{,}192{,}896$$
$$52^{13} = 20{,}325{,}604{,}337{,}285{,}010{,}030{,}592$$
$$52^{14} = 1{,}056{,}931{,}425{,}538{,}820{,}521{,}590{,}784$$
$$52^{15} = 54{,}960{,}434{,}128{,}018{,}667{,}122{,}720{,}768$$

By using both uppercase and lowercase numbers, the user has increased the number of guesses the attacker has to make significantly. For a ten-length password, the number of guesses increases from 141,167,095,653,376 to 144,555,105,949,057,024. That’s an increase from a 15-digit number of guesses to an 18-digit number of guesses. It is 1024 times bigger.

To complete the series, let us look at what happens when we include all ten digits (0 through 9) and symbols such as !, (, and .. For simplicity, we will just use 16 symbols, even though there are more, so that combined with the digits we have added another 26 symbols to our possible passwords:

$$78^1 = 78$$
$$78^2 = 6{,}084$$
$$78^3 = 474{,}552$$
$$78^4 = 37{,}015{,}056$$
$$78^5 = 2{,}887{,}174{,}368$$
$$78^6 = 225{,}199{,}600{,}704$$
$$78^7 = 17{,}565{,}568{,}854{,}912$$
$$78^8 = 1{,}370{,}114{,}370{,}683{,}136$$
$$78^9 = 106{,}868{,}920{,}913{,}284{,}608$$
$$78^{10} = 8{,}335{,}775{,}831{,}236{,}199{,}424$$
$$78^{11} = 650{,}190{,}514{,}836{,}423{,}555{,}072$$
$$78^{12} = 50{,}714{,}860{,}157{,}241{,}037{,}295{,}616$$
$$78^{13} = 3{,}955{,}759{,}092{,}264{,}800{,}909{,}058{,}048$$
$$78^{14} = 308{,}549{,}209{,}196{,}654{,}470{,}906{,}527{,}744$$
$$78^{15} = 24{,}066{,}838{,}317{,}339{,}048{,}730{,}709{,}164{,}032$$

How many guesses are needed to make things difficult for a computer to brute-force in reasonable time? It depends on the hashing algorithm. There are more than one, and some take longer than others. The hash that takes the least amount of time that is still in widespread use is called MD5. Modern hardware for a single machine can compute more than *20 million* MD5 hashes *per second*! That sounds like a lot, right?

Even at 40 million MD5 hashes per second, it takes a long time once passwords get around ten symbols long. Even with just uppercase and lowercase passwords (no numbers or special characters), there are 144,555,105,949,057,024 possible passwords of length ten.

Dividing that number by 40 million per second is 3,613,877,648 seconds to complete all the guessing. That works out to about 120 years!

Of course, attackers can speed up the process by using multiple machines. Two machines would cut the time in half, four machines in a quarter, and so forth. Still, it is important to see how increasing the length by one can make such a big difference. Passwords less than size eight are almost always worthless. Ten is a better starting point.

The other widely used approach to guessing passwords is to use a dictionary. Many users choose passwords based on dictionary words. It might have been difficult for Ali Baba to guess that the password was “Open Sesame,” but modern computers do not have this limitation. A computer can iterate through a dictionary of words, hashing each one and looking for a match. Users often think they are making their password stronger by using a misspelling, replacing letters with numbers or other symbols, or throwing in some numbers at the end.

Unfortunately for the users that rely on such tricks, the reality is that computers can figure these substitutions and minor additions with relative ease. Suppose that the user is creating a password out of his head (i.e., the user is not consulting an actual dictionary). Americans have a vocabulary of between 20,000 and 35,000 words. When picking a password, a user will almost certainly pick from a much smaller list of relatively familiar words. Nevertheless, even if they could pick any random word from their mental dictionary, it would not be difficult for a computer to try out 35,000 words.

Computers can also apply rules to a word to create variations. A computer can have a set of rules for replacing letters with numbers and other common substitutions. Some password crackers use a set of rules called the “Best of 64,” which are 64 of the most common changes people make to their passwords. If each word is transformed into

64 variants, the number of words to test only increases from 35,000 to 2,240,000. That is still significantly less than an eight-character password of even just lowercase letters. There are not enough transformations that a human can do in their head that will significantly challenge the capacities of a computer.

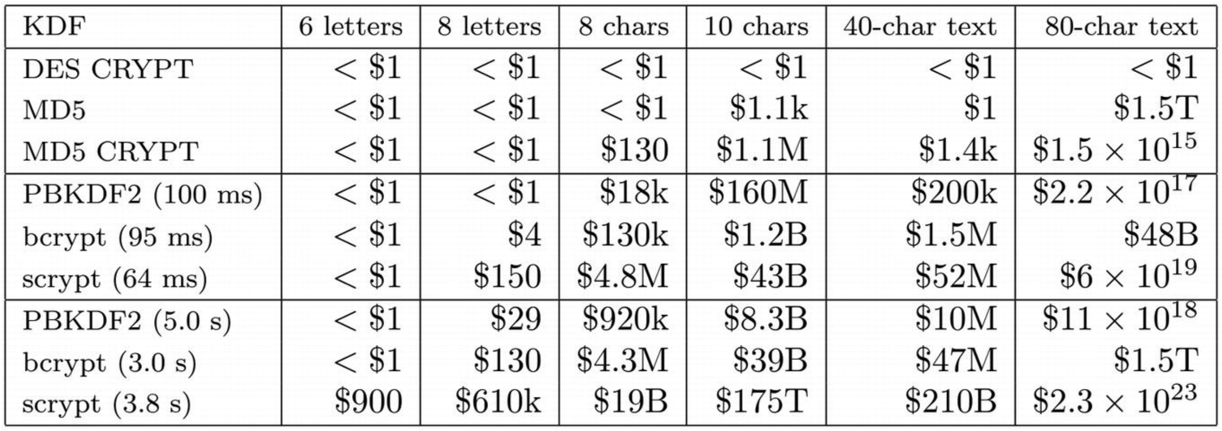
The exception, however, is a combination of random words. Even if the user is picking from a significantly smaller list of words, perhaps just 2000 words total, a combination of random words quickly becomes impossible for a computer to do quickly.

To figure out the number of guesses a computer would have to do for this type of password, you need to calculate combinations of words instead of combinations of symbols. So, instead of starting with one-symbol passwords of which there are 26 (or 52, or 78) possibilities, you start with a single dictionary word of which we assume there are 2000 possibilities. A two-word password will have 2000 times 2000 possibilities, or 4,000,000. The following table shows the growth in possibilities:

$$2000^1 = 2{,}000$$
$$2000^2 = 4{,}000{,}000$$
$$2000^3 = 8{,}000{,}000{,}000$$
$$2000^4 = 16{,}000{,}000{,}000{,}000$$
$$2000^5 = 32{,}000{,}000{,}000{,}000{,}000$$

When creating a multiple-word password, four words should be the minimum.

Whether a computer is doing brute-force attacks, dictionary attacks, or a combination, each guess requires performing a hash operation. The attacker can be slowed down significantly by using hash algorithms that are very slow. MD5, as we have seen, is one of the fastest and modern computers can do tens of thousands of MD5 hashes in a second. Authentication systems should use slower algorithms like SHA-256 or, even better, an algorithm designed to be slow such as PBKDF2. Computers can run the latter algorithm at about 100,000–200,000 per second. This can delay the attacker by a factor of 100 or more. So if it would have taken 7 days to crack the same set of passwords hashed using PBKDF2. Figure 2-8 includes the increased effort required for cracking passwords hashed by different hashing algorithms in 2002.



KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	$< \$1$	$< \$1$	$< \$1$	$< \$1$	$< \$1$	$< \$1$
MD5	$< \$1$	$< \$1$	$< \$1$	\$1.1k	\$1	\$1.5T
MD5 CRYPT	$< \$1$	$< \$1$	\$130	\$1.1M	\$1.4k	$\$1.5 \times 10^{15}$
PBKDF2 (100 ms)	$< \$1$	$< \$1$	\$18k	\$160M	\$200k	$\$2.2 \times 10^{17}$
bcrypt (95 ms)	$< \$1$	\$4	\$130k	\$1.2B	\$1.5M	\$48B
scrypt (64 ms)	$< \$1$	\$150	\$4.8M	\$43B	\$52M	$\$6 \times 10^{19}$
PBKDF2 (5.0 s)	$< \$1$	\$29	\$920k	\$8.3B	\$10M	$\$11 \times 10^{18}$
bcrypt (3.0 s)	$< \$1$	\$130	\$4.3M	\$39B	\$47M	\$1.5T
scrypt (3.8 s)	\$900	\$610k	\$19B	\$175T	\$210B	$\$2.3 \times 10^{23}$

Figure 2-8 This table, by Colin Percival, shows the approximate dollar cost to break passwords of different sizes with different password hashing functions in 2002. I have not been able to find a more recent comparison, so these numbers are definitely out of date. However, it is a useful comparison of the *relative* speed of the different hashing algorithms. Remember in this case that *slower and more expensive is better for the defender*

Story Time: Legacy Systems Strike Back

I was once asked to evaluate the security of a compromised password file. That is, attackers had broken into a cloud system (an interesting story in and of itself) and stolen the file of password hashes for a web application. I was asked to evaluate the password file and see how easily it would be compromised.

Unfortunately, the file was nearly instantly crackable. The password hashes were created with MD5 and *were* *not even salted*. Just using a very basic cracking tool, I cracked most of the passwords in about two minutes.

I was also asked to review the source code for the password system and figure out how this had happened because it was *supposed* to be using a better hashing algorithm. What I found was that the MD5 was a legacy password hash from more than a decade previous. The system was "upgraded" to use the better hash with salting. However, the upgrade had been designed to support *both* hashes during a transition period. Remember, the system does not have the unhashed password, so there is no way to just convert the MD5 hashes into the newer hashes. The transition code worked by having people log in to the system, confirming the password is correct by checking against MD5, and then storing it in the newer format. Clearly, at some point, the MD5 values were supposed to be stripped out. But somewhere along the way, it was forgotten about. Either they kept putting it off until they forgot about it or a developer left the company or some other event distracted engineering from removing the vestigial data.

Unfortunately for the company, this meant that when the attackers stole the file, the data was protected with a really strong door and a really weak door. Effectively, this meant that the data was easily compromised.

Putting it all together, a good password of random letters might look like this:

```
>^KdDW+(x.
```

Or, alternatively, a good password might be four random words:

```
WhatJuggleChinRed
```

Note that the words *must be random*. “Mary had a little lamb” is not very secure, because it consists of common words that are readily associated together. Characters can be inserted into the middle to increase the password’s strength:

```
WhatJuggle8;1ChinRed
```

The main point here is that passwords with higher *entropy* are more secure. That is, the more surprising, unique, or difficult-to-guess the content, the harder it will be for an attacker to crack the password. Passwords with low entropy are more vulnerable to common attacks than passwords with high entropy. Password rules are all attempts at getting humans to create sufficiently complex passwords while working with the limitations of our psychology.

On that note, one of the arguments for using a password made from multiple words rather than from random symbols is that it is supposed to be easier for the user to remember.

The problem with this belief is that it is not supported by research [85]. In addition, even if a single password is more memorable, a user should have a different password for every system. Users might have hundreds of systems they use and would need to remember hundreds of passwords. This is not feasible for most users.

The best recommendation for password security is to use a password manager. A password manager is a program or service that stores all of a user’s passwords in an encrypted container. Only a master password decrypts the passwords. Plug-ins can be used to enable passwords to be entered automatically at websites from the decrypted storage. Most provide tools for automatically generating new passwords and managing password rotation.[8]

In general, passwords should not be written down outside of a secure application such as a password manager. But correct use of a password manager helps users to meet password security requirement numbers 2 and 3.

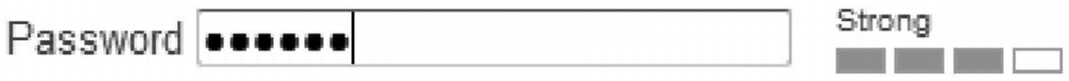
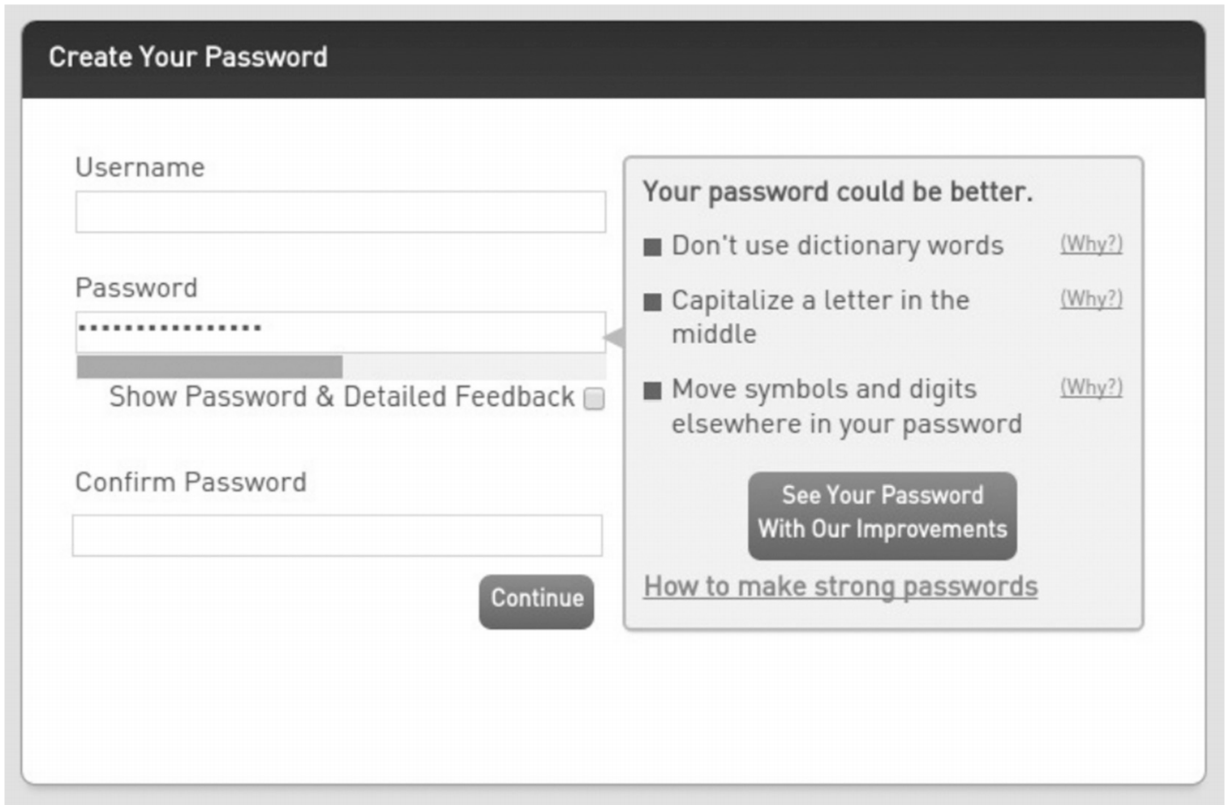


Figure 2-9 An example password meter from the 2013 time frame. More recent password meters typically provide some feedback on what is wrong with the password


Create Your Password
Username
Password
Show Password & Detailed Feedback
Confirm Password
Continue

Your password could be better.

- Don't use dictionary words (Why?)
- Capitalize a letter in the middle (Why?)
- Move symbols and digits elsewhere in your password (Why?)

See Your Password With Our Improvements
How to make strong passwords

Figure 2-10 Researchers at CMU developed a password meter with password advice based on quantifiable research [269]

Earlier in this section, I hinted that technology can also help users to choose a good password (or ensure that the password manager generated a good password). During the registration phase when a user is choosing a password, authentication systems can analyze the user's proposed password and alert them if it is weak. This technology is commonly called a “password meter.” Figure 2-9 shows an example password meter from 2013.

Password meters should *not* be confused with systems that enforce certain rules on a password. For example, you may have had to create a password for a system that required the password to have at least one uppercase letter, at least one number, and at least one special character. Unfortunately, many users simply add these to the end of a bad password:

```
Password123$
```

A good password meter can recognize this kind of bad password and communicate to the user what is weak about it. More importantly, a good password meter can help the user to make a better password choice with specific suggestions [269]. Figure 2-10 shows an example.

Dr. Cranor’s work is important because it is based on empirical research. One of the problems in security is designers sometimes create a system based on what they *believe* is the problem or what they *believe* is the solution. The best security technologies, and the best uses of technologies, are based on scientific study.

For example, many voices in the cybersecurity world have promoted the regular changing of passwords. “Passwords should be like underwear,” the saying goes, that is, they should be changed regularly [130]. But this assertion was based on educated guesses and reasoning based on assumptions. Dr. Cranor assembled empirical evidence that determined there was no advantage to regular password changes. If a user is using a password poorly, such as weak passwords, reused passwords, or writing them down in insecure places, password changes are unlikely to help. If a user is using a strong password, with a different password for each system, then password changing does not add any benefits. Not only do frequent password changes not add security but they can actually make it worse as users subjected to such requirements are more likely to store their passwords insecurely or to use weaker but easier-to-remember passwords [85].

Password Reset Challenges

Password resets are almost certainly a necessity in some form or another. Users *will* forget passwords. I, myself, use a password manager and have still managed to not enter a new password when I created it or entered it incorrectly. When I went to look for it, it was not there and I obviously could not remember it. Lost passwords will always happen.

Regardless of how necessary they are, password resets are very, very dangerous for many reasons. The biggest risk, of course, is an attacker triggering the password reset to let themselves into the system!

The starting point to understanding the risk of password resets is to see them as an alternative version of the password itself. You do not just have a password that lets you into the system, you have a password and a password reset that let you into the system. The attacker is happy to attack either one. It is all the same to them.

The sad truth is that password resets are often not as well defended or protected as the password itself. Take, for example, security questions. What is your mother’s maiden name? That question is protecting your account? Even questions that might appear to be more safe, such as the name of your first pet, are not any better. How many of your friends and family know that information? If you are young enough, it might be available on social media. Your parents may have posted about it.

My recommendation is to never answer security questions with real answers. I recommend using real passwords that are stored in a password manager, just like any other password. I have seen this advice repeated in security trainings I have attended. I find it amusing that we have to be advised by security *not* to use widely deployed security features, but that is a good representation of the overall state of cybersecurity today.

Other password reset solutions are also problematic. IT help desks that provide password resets often rely on even weaker security questions over the phone. They often ask for billing zip code or other account information that can be easily obtained from a statement or bill. Some companies are finally requiring a PIN for phone verification, basically introducing a verbal password. The problem is that users often choose very weak security pins (such as their social security number, phone number, or anniversary date), or they forget them and the company has to fall back to much weaker security measures.

There is not yet a good solution to this problem. For the time being, companies have to rely on fraud detection algorithms, and users have to monitor their accounts for unusual activity. Most systems will email the user if the password is reset, but there are a number of ways attackers can deal with that. The easiest way around is to simply get the password reset at night while the user sleeps. They can do all the mischief they want for a good number of hours before the user will even be awake.

Story Time: Destroying Someone to Steal a Twitter Handle

In 2012, Mat Honan, a writer for *Wired*, had his digital life destroyed. Attackers managed to take over and destroy data in his Gmail account, his personal MacBook, and other accounts. The worst part? He lost images forever of his child’s first year. That entire part of his life and the child’s life gone. Forever.

The attackers also took over his Twitter account and used it to distribute offensive messages. Why did they do this, you might ask? Was it because he was a writer for a tech magazine? Nope. The attackers just wanted his three-letter Twitter handle. Memories, emails, and digital life gone. For a Twitter handle.

How did the attackers get in? At least partially through password resets. You can read the entire tragic story in an article he wrote about it [136]. This is a risk we all run on the Internet today. It is true that companies have improved some aspects of this security a little, but it remains a significant problem.

Another defensive solution employed by a few institutions (such as some banks) is to require the user to receive a code on their mobile device that they repeat back to the service professional they are speaking with. This approach is probably the best technology currently available but is not yet widely deployed.

For the time being, password resets are an Achilles heel for systems that rely on something-you-know authentication.

Something You Have

Although passwords are by far the most common authentication technology, authentication using “something you have” has grown in popularity. Much of this has to do with the ubiquity of mobile phones.

Before explaining the modern design of a something-you-have authentication system, I will start by describing how something-you-have has been used since long before computers. That is, people have authenticated themselves for centuries based on holding some kind of physical proof of identity. A fictional, but illustrative, example is found in the movie *Annie*. In this story, Annie is an orphan living in a US orphanage during the Great Depression of the 1930s. At this time period, it was not uncommon for parents to leave their children at orphanages if they could not feed or care for them. Many hoped to return and be reunited with their children once they had found work. Annie, about eight years old, still believed her parents were alive and would come back for her someday. She had been in the orphanage since being a small child, however, and had little idea of what they looked like.

Unfortunately for Annie, her parents had already died. When it looked like Annie was to be adopted by a rich patron, Oliver Warbucks, a conspiracy was formed to pose as her parents in order to swindle money out of said patron. The fraudsters included the head of the orphanage who had in her possession a number of personal items of both Annie's and the parents. The faux parents showed up at the Warbucks house with a number of "proofs" of their identity: forged identities and Annie's birth certificate.

More personally, they had in their possession (taken from the orphanage) half of a locket. Annie wore the other half and hoped that, one day, her parents would come bringing the other half of the locket as proof of their identity. When these fraudulent people arrived bearing the matching other half, it was seen as positive proof that they were Annie's real parents.[9] The locket was seen as a stronger evidence of identity in some ways even than the forged government identities. Because the locket had been broken, it had created an item so unique as to be almost impossible to forge, especially while Annie was in possession of the other half.

But the unforgeable (in practice) nature of the locket did not prevent it from being stolen. Herein is another example of a failed authentication and an excellent introduction to the security requirements for authentication using something you have. The “something” in “something you have” is often called a *token*, and I will use that term hereafter.

Token Requirement 1: Exclusive Possession. The token is only in the possession of a principal with a valid identity claim. Annie’s story of the stolen locket is fictional but real. That is, stolen tokens are a very real and very serious problem. For many people, their mobile phone is used as a token. A report from 2013 from Lookout noted that 3.1 million Americans had been victims of phone theft. On a daily basis, that is a staggering rate of 8493. The same report concluded that 44% of the victims had left their phones behind in a public place. One should find these statistics sobering in light of how many authentication purposes they are used for.

Token Requirement 2: Unforgeable. The token cannot be easily forged or duplicated. This is another very serious problem. Annie’s locket was valuable for identification because of how unique it was. Almost all tokens are mass manufactured devices that are individualized with numbers and codes that should be unique. But if those codes can be stolen and copied into another device, both devices will act in exactly the same way. Authentication using this device is broken.

Token Requirement 3: Secure Protocol. There exists a secure protocol for proving possession of the token. Not only must the principal with the valid identity claim *have* the token, they must be able to *prove* they have the token in a secure way. Technically, this is a requirement for passwords as well, but in practice, it can be a bit more difficult for tokens. Passwords are more conducive to transmission, and they are more easily changed.

Before the advent of mobile phones and devices, tokens were often fobs that could be fitted on key chains. These devices still see some use and work on similar principles used in mobile phones, so they make an excellent illustration of how these kinds of systems work in practice.

The fob pictured in Figure 2-11 is a product of the RSA corporation called SecurID. These devices work by having secret data, known as a *seed*, embedded within the fob. The fob is designed to be tamper-resistant such that extracting the seed from the fob is not feasible.

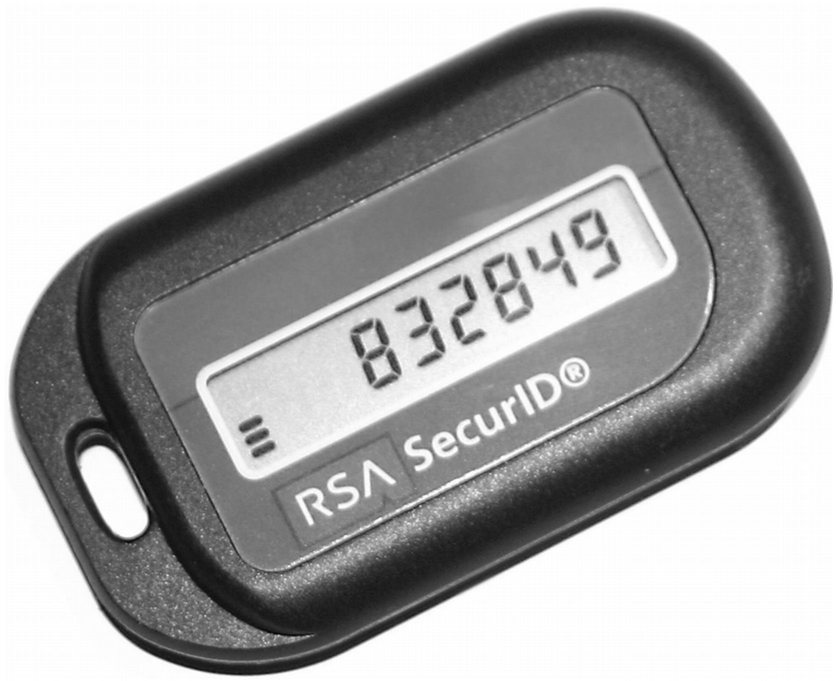


Figure 2-11 An RSA SecurID Token. It displays a code on the screen that is unique to the device and changes on a regular interval. Image downloaded from Wikimedia Commons (commons.wikimedia.com) and licensed under Creative Commons Attribution-Share Alike 1.0 Generic license (https://creativecommons.org/licenses/by-sa/1.0/deed.en)

When the fob is purchased, however, the seed is provided to the purchaser in a data form. The seed must be uploaded to an authentication server. The other requirement is that the fob and the server have synchronized clocks.

Once running, the fob will display a number that changes at fixed intervals such as 30 seconds. When authenticating, the user transmits the number displayed on the screen to the authentication server. The authentication server can verify that the number is correct to verify that the principal is in possession of the fob. Each fob generates a unique sequence of numbers based on the internal seed, so no two fobs can predictably generate the same number at any given point in time.

How does the authentication server know that the number transmitted for authentication is correct? The server and the fob both share the secret seed. This is why the seed must be uploaded to the server. The seed is used to generate the sequence of numbers. So long as both the authentication server and the fobs have the same secret seed, they can both generate identical numbers.

The numbers change based on a time interval because the time interval itself is used to calculate the number displayed on the fob’s screen. The process is somewhat similar to hashing with a salt. If we take the seed as input and the time interval as a salt, the hash output will be unique for each seed. The actual process is more complicated, and the output has to be scaled down to a smaller number, but the idea is similar.

So, assuming that

1. The user’s fob and the authentication server share a secret seed.
2. No other parties know the seed and no other fobs have it.
3. The user’s fob and the authentication server have synchronized clocks.

Then when the user transmits the number displayed by the fob to the authentication server, the authentication server can use the seed and time interval to compute an identical number. If all the preceding assumptions are true, this constitutes a proof that the principal requesting authentication is in possession of the fob. The vast majority of something-you-have authentication systems work on similar principles, including Google Authenticator, Duo, and others. Figure 2-12 illustrates these concepts as a generic process.

There are some potential problems with this approach. First, it is hard to argue that this is really something-you-have authentication when, in fact, authentication comes down to possession of the *seed*. The seed is, more or less, a password. Anybody with the seed can generate the appropriate numbers for the fob. In fact, software programs for RSA SecurID exist. These so-called *soft tokens* do exactly the same thing only with the seed programmed into memory rather than imprinted into the hardware fob.

The counterargument is that the *user* does not need to know the secret; the user only needs to be in possession of the fob. The problem with this argument is that the seed has to be transmitted to the authentication server. An intrusion that compromises this server could steal all registered seeds. All the fobs that authenticate with the server are now worthless as tokens. As soon as the *secret* is disclosed, the fobs cannot function.

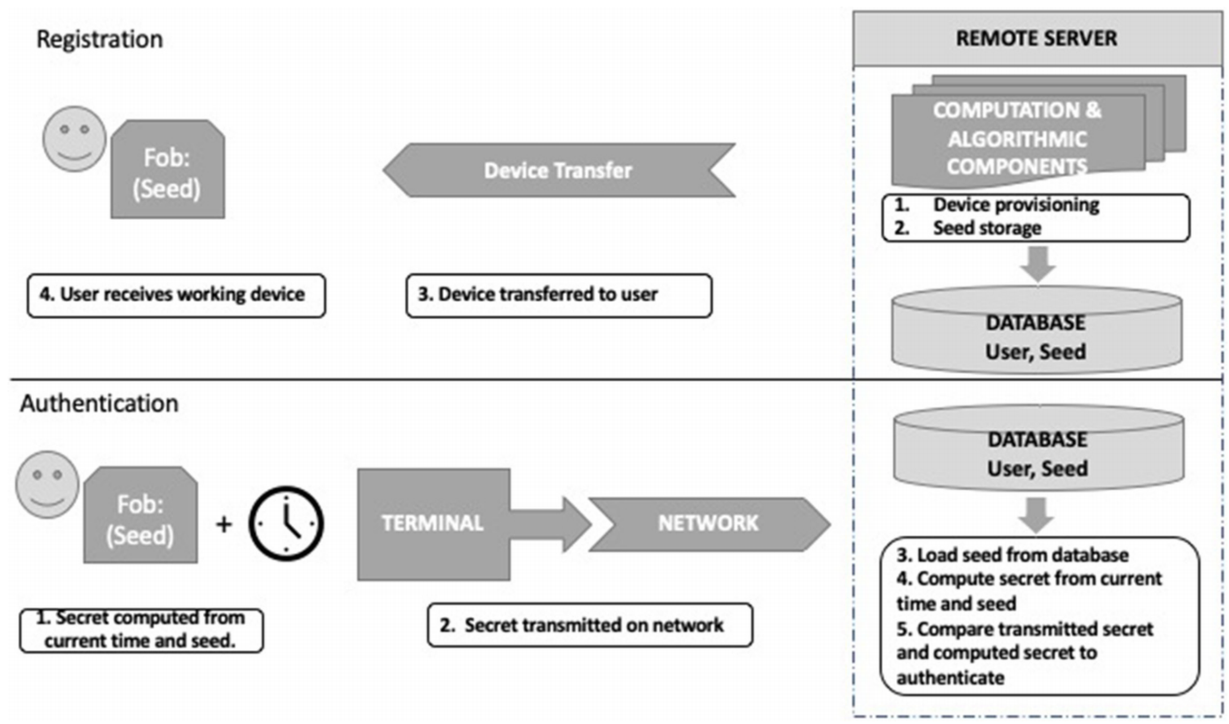

Registration
REMOTE SERVER
COMPUTATION & ALGORITHMIC COMPONENTS
Fob: (Seed)
Device Transfer
1. Device provisioning
2. Seed storage
4. User receives working device
3. Device transferred to user
DATABASE User, Seed
Authentication
DATABASE User, Seed
Fob: (Seed)
+
TERMINAL
NETWORK
3. Load seed from database
4. Compute secret from current time and seed
5. Compare transmitted secret and computed secret to authenticate
1. Secret computed from current time and seed.
2. Secret transmitted on network

Figure 2-12 In a something-you-have authentication, the registration process typically involves issuing a token from an issuer. In the case of a fob based on a seed, the seed will be recorded in a database when the token is issued. During authentication, the output of the token is compared with values computed at the server

Soft tokens are even more easily broken into. They do not have the tamper resistance of the hardware fobs.

Story Time: RSA Data Breach
As a matter of fact, this risk is not just hypothetical. In 2011, RSA was hacked and a trove of SecurID seeds were exfiltrated. This enabled hackers to clone any SecurID token built with one of the stolen seeds (effectively creating false “soft tokens”), breaking the two-factor authentication users, organizations, and governments relied on [117].

Codes from these kinds of tokens can also be stolen using network interception or social engineering. A combination of both network interception and social engineering could be a fake website that convinces a user to “reuse” their token for signing in. When the user signs in at the fake website, they submit the number from their token. The evil website now signs in as the user to the website for which the token was actually registered.

Nevertheless, this type of something-you-have authentication is much better than certain other alternatives. For example, sending a confirmation number by email is a much worse choice. In actuality, an email account is protected by a password. Even more than the fob, an email account is really just a form of something you know [114].

SMS messages used to be popular for something-you-have authentication. The idea was that an SMS code is sent only to the unique device, and anyone in possession of the code must be in possession of the unique device. The problem with the SMS messages was manifold. Some users have their SMS messages transmitted to their email. Possession of the code may have nothing to do with possession of the device.

Another problem with SMS codes is that there were many social engineering scams that extracted them. An attacker might try to sign in as a user, triggering an SMS code. The attacker calls the user claiming to be a technician and asking for the SMS code. This was harder to do with fobs because the timeouts were so much shorter (e.g., 30 seconds). By the time someone called you, it was just about time for the fob to display a new number.

For these and other reasons, SMS messages are discouraged for something-you-have authentication.

There is another kind of authentication that does not really fit in either something-you-have or something-you-know. In this kind of authentication, an entity, often a computer program, proves its identity by proving ownership of a digital certificate issued by a third party. There is no token. Ownership of the certificate is proved through asymmetric cryptography. This technology will be discussed at length in Chapters 6 and 9. I mention it here for completeness.

Something You Are

The final authentication proof commonly used in contemporary systems is something you are. This generally refers to authentication of a human through some unique characteristic or trait they possess. I have refrained from using the word *human* in reference to the previous authentication mechanisms because they can also be used for programs, machines, and even groups. Something you are is almost always for identifying specific and individual humans. The identification of, and verification of, these traits is generally called *biometrics*.

Having already given examples of the previous two approaches to authentication using examples from movies and literature, I think it makes sense to do the same thing within this section as well. This example comes from the story of Cinderella.[10] In this fairy tale, the young and abused girl of the same name is enabled by her fairy godmother to attend a ball. The prince of the kingdom falls in love with her during the dance, but she flees from him at midnight when the spell is wearing off. Somehow, the glass slippers she was wearing do not disappear when all of her other magically created clothing fades away. In her haste to escape, she leaves one of the glass slippers behind.

Determined to know the identity of the mystery woman, the prince has the glass slipper fitted to each unmarried lady in the kingdom. When the slipper is to be fitted to Cinderella, the stepmother interferes and causes the slipper to be shattered in an “accident.” Fortunately, Cinderella produces the other glass slipper that, of course, fits her perfectly.

I like this illustration of a biometric (a 3D footprint?) because it includes hints at many of the requirements necessary to make biometrics work as an authentication technology.

Biometric Requirement 1: Unique Characteristic. The biometric characteristic is effectively unique. What is “effectively” unique depends a great deal upon context, and I will discuss this in some detail later in this section. But it is crucial that no two people have identical characteristics. In the story of Cinderella, her feet were magically unique, and the glass slipper fit on nobody else.

Biometric Requirement 2: Measurable Characteristic. Not only must the characteristic be (effectively) unique but it must be possible to measure it. This requirement must be understood in light of the first requirement as well. It must be measurable with sufficient resolution to distinguish between two people. This is typically the harder problem. Many biometric characteristics are unique by nature, but detecting the distinct features is more challenging. If you have watched the Disney animated Cinderella, you may recall that the Wicked Stepmother tried to thwart Cinderella by tripping the man carrying the glass slipper, causing it to shatter into pieces. Effectively, she tried to make Cinderella’s magically biometric feet unmeasurable by destroying what she thought was the only “scanner” that could measure them. The necessary quality of the measurement also varies depending on whether the biometric is used for verification or identification. Verification is a simpler check to see if the test characteristic is registered and outputs a yes or no (see Figure 2-13). On the other hand, identification involves comparing the test characteristic against a database and must output which stored identity it is most likely to match (see Figure 2-14).

Biometric Requirement 3: Unforgeable Characteristic. The characteristic cannot be forged, replicated, or otherwise duplicated. Biometrics and tokens are similar in this regard. In fact, a biometric is a biological "token." However, biometrics are more fragile to the extent that *they cannot be replaced*. If someone steals the seed of an RSA SecurID, the fob can be discarded and a new one purchased for relatively little money. However, if someone is able to steal a fingerprint, there is little that can be done to replace it.

Biometric Requirement 4: Stable Characteristic. The characteristic will not change too much over time or otherwise be "lost." Biometrics can and do change. How much change depends greatly on both the characteristic and lifestyle. For example, manual laborers tend to have weaker fingerprint matching [40, Chapter 17]. And, of course, accidents can cause the characteristic to be destroyed through amputation, scarring, or other transformations. It would have been a very sad day for Cinderella if when she needed to try on the glass slipper she found that her feet had swollen. After all, her Wicked Stepmother was having her do a lot of physical labor that might cause bruising and swelling. It is a good thing that her magical feet were always the same size and would always fit the slippers!

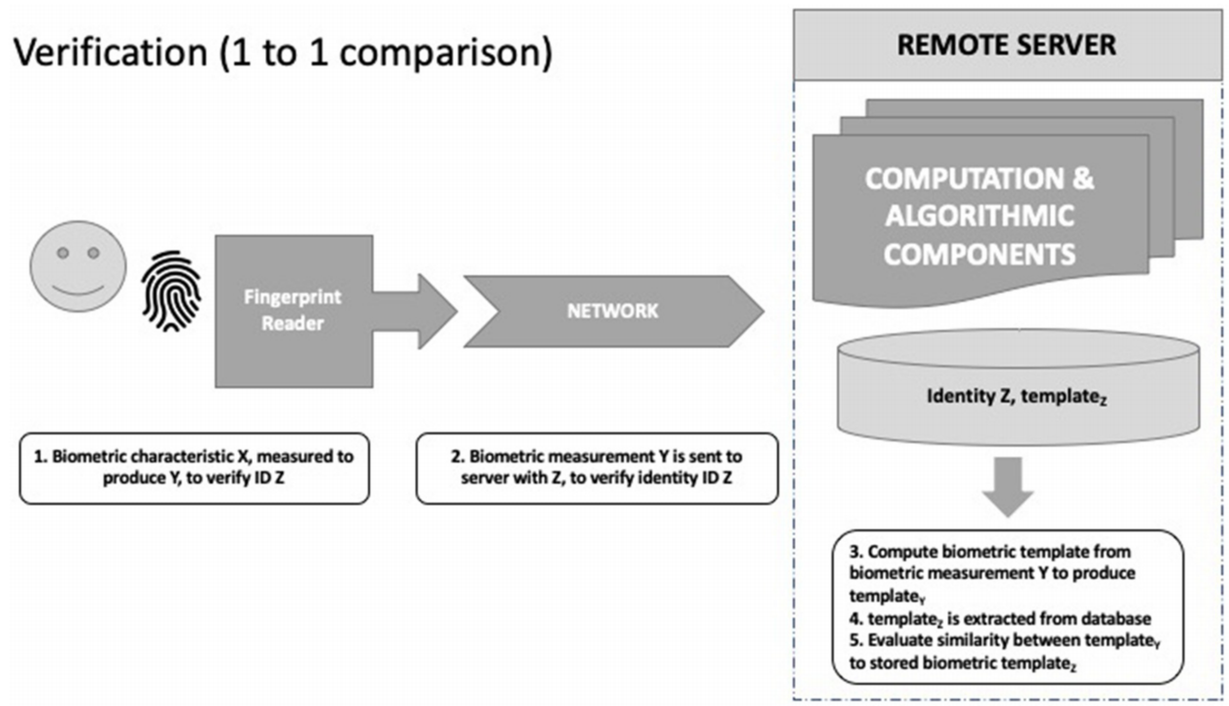

Verification (1 to 1 comparison)
REMOTE SERVER
COMPUTATION & ALGORITHMIC COMPONENTS
Fingerprint Reader
NETWORK
Identity Z, template$_Z$
1. Biometric characteristic X, measured to produce Y, to verify ID Z
2. Biometric measurement Y is sent to server with Z, to verify identity ID Z
3. Compute biometric template from biometric measurement Y to produce template$_Y$
4. template$_Z$ is extracted from database
5. Evaluate similarity between template$_Y$ to stored biometric template$_Z$

Figure 2-13 For a biometric verification, there is a specific biometric template that the detected characteristic is compared to

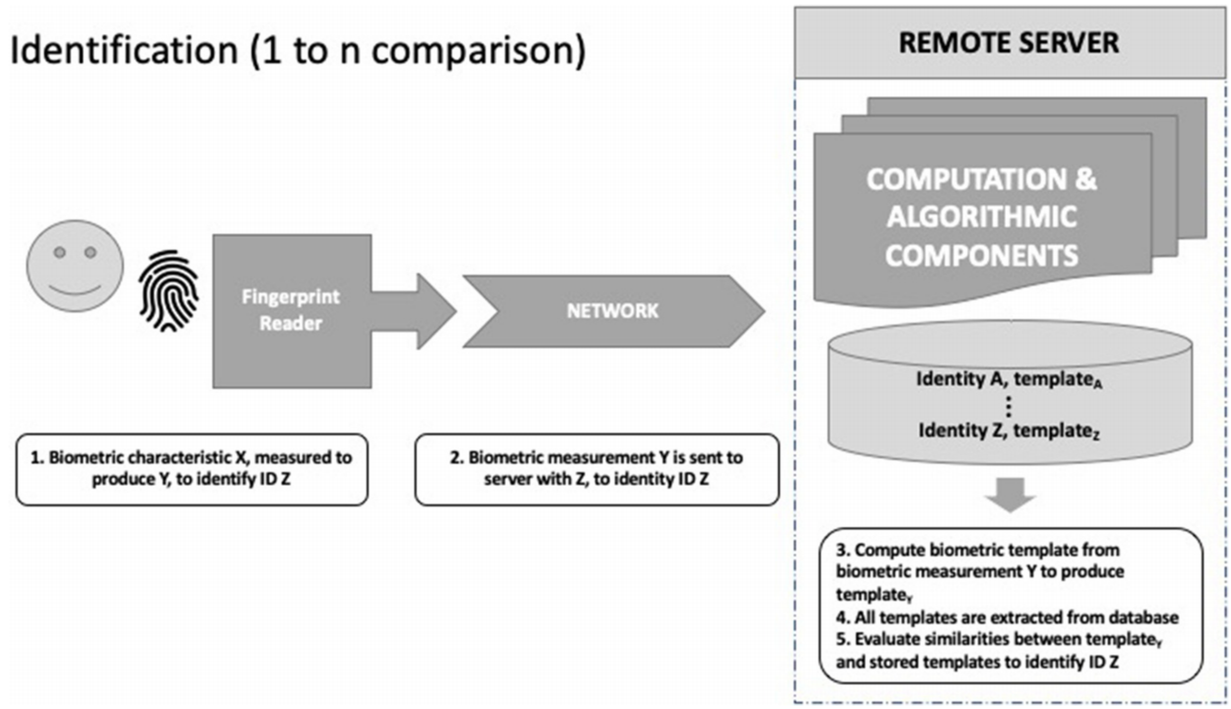

Identification (1 to n comparison)
REMOTE SERVER
COMPUTATION & ALGORITHMIC COMPONENTS
Fingerprint Reader
NETWORK
Identity A, template$_A$
⋮
Identity Z, template$_Z$
1. Biometric characteristic X, measured to produce Y, to identify ID Z
2. Biometric measurement Y is sent to server with Z, to identity ID Z
3. Compute biometric template from biometric measurement Y to produce template$_Y$
4. All templates are extracted from database
5. Evaluate similarities between template$_Y$ and stored templates to identify ID Z

Figure 2-14 Unlike biometric verification, biometric identification must *search* for the best matching template that will match the detected characteristic

Biometric Requirement 5: No Revocation. The characteristic will never need to be revoked. I alluded to in Requirement 3. Generally, users cannot change their biometrics. If a biometric is duplicated, stolen, or otherwise compromised, there is no way to issue a new biometric to the victim.

Biometric Requirement 6: Secure Protocol. There exists a secure protocol for proving possession of the biometric. Not only must the principal with the valid identity claim *have* the biometric, they must be able to *prove* they have the token in a secure way. This requirement is similar to Token Requirement 3. To repeat, this is technically also a requirement for passwords. But the nature of biometrics makes it more difficult in practice. As just discussed in Biometric Requirement 5, it is a really bad thing for biometric data to be lost or stolen. Replacement is simply not an option. So the protocol for transmitting biometric data is far more sensitive and brittle.

The use of biometrics has exploded with the use of mobile phones and devices with fingerprint readers and face scanners. These biometric readers not only grant access to the device but apps on the device can rely on those readers as well. Banking applications, password managers, and other critical programs make use of these tools in order to eliminate the need for the user to enter a password. Entering passwords is always slow, but it becomes almost barbaric for some phone keyboards.

But ubiquity does not equal good security, and it is important to understand the limitation of the technology. At the beginning of the chapter, I introduced the idea of false positives and false negatives. Typically, these terms only apply to something-you-are authentication. Nevertheless, I introduced them as a more general issue with all types of authentication on purpose. Although unusual, false positives and false negatives *can* happen with passwords or tokens. But it is so rare as to be vanishingly small and rarely discussed. This is one of their strengths. It is almost never the case that a user entering a correct password will be told it is wrong and almost never the case that a user entering the wrong password will be told it is correct.

Biometrics, on the other hand, experience this problem *by nature*. This is a crucial reason why they are not a panacea for authentication problems nor a straightforward replacement for passwords (which are universally despised). Biometric errors, both false positives and false negatives, will limit their usefulness. They are so intrinsic to these systems that it is the subject of significant research, comparative analysis, and deployment recommendations.

In the biometric world, false positives (accepting the wrong biometric for a given identity) are also called *false accepts* and the rate of these errors the *false accept rate* or FAR. False negatives (rejecting the correct biometric for a given identity) are also called *false rejects* and the rate of these errors the *false reject rate* or FRR.

The problem with false accepts are relatively straightforward to understand. If an unauthorized person gains access to the system, the security is completely bypassed. But perhaps unintuitively, false rejects can be viewed in some systems as the bigger problem. Why? Because in a number of common scenarios, a system with occasional successful intrusions is actually less of a problem than a system that cannot function because authorized people cannot gain access. For example, in the UK, banks have as a target a 1% false accept rate but a much lower .01% rate of false rejects [40, Chapter 17]. On the other hand, high-security systems require that the FAR is the prioritized metric. NIST, for example, in their document on Biometric Specifications for Personal Identity Verification, explicitly specifies that FRR “does not represent a direct security objective” [120].

The nature of biometric systems means that decreasing false positives will often increase false negatives and vice versa. The reason for this is because the system is trying to compare a contemporary measurement of the characteristic to a database of previously recorded characteristics. Because no measurement is perfect, the stored version and the current one will not match exactly. So the system has to decide which imperfect match is the “best.” To reduce the risk of false positive, the system will typically be tuned to accept far more minimal differences. But at the same time, this typically results in more false negatives because the current measurement, even though it is the correct biometric, seems too different from the stored version.

The *receiver operating characteristic* is a system’s trade-off between false positives and false negatives. Operators have to adjust this characteristic until the system operates within acceptable error levels for their organization. If the system is tuned such that the error rates are equal, the system is said to have an *equal error rate*. As stated earlier, this is rarely the right setting for a system, and quite often the system is configured to ensure that authorized users almost always get in (decreased false negatives).

One of the other big problems with biometrics is understanding the nature of statistics and how this impacts results. Because the biometric system basically has to compare measurements against each other, the number of possible measurements significantly impacts the results.

To put it more concretely, consider your mobile phone or device. It probably has a fingerprint reader. It probably is storing only your fingerprint. How many other people is it likely to compare your fingerprint to? There might be more than seven billion people on the planet, but the vast majority of them will never come in physical contact with your fingerprint reader. Your phone's reader only needs to distinguish between you and maybe a few dozen people—a few hundred at most. If the chance of a false positive is 1 in 10,000, there is a very low chance that any of those few dozen or few hundred people will gain access.

But what if the fingerprint reader is going to be used in a national database for accessing social security benefits in the United States? Now there are hundreds of millions of people on file and hundreds of millions of people trying to gain access. If the false positive rate is 1 in 10,000, there are going to be thousands of people gaining access to the wrong accounts.

Unfortunately, biometrics are often seen as far more effective and unerring than they are. This is also a significant problem in security culture. If the mechanism is seen as infallible, the policies, procedures, and operations of the organization are not equipped to deal with the system when it inevitably fails [17, 40]. If using biometrics causes people to *stop thinking* ("I don't have to worry about anything because I use biometrics!"), the individual and the organization they belong to will inevitably get hacked.

One other problem with biometrics worth mentioning is the possibility of social exclusion. There are many groups of people that have harder-to-detect characteristics. In some circumstances, this may include certain subsets of manual laborers, the elderly, the disabled, and other people that are already facing disadvantages. Other biometric systems have been measurably shown to have racial biases or other socially unacceptable operations. Given the sensitive nature of physical features in the modern world, contemporary technologies must be designed and deployed with these issues in mind [17, 40].

Multifactor Authentication

Although something you know, something you have, and something you are have been presented as “equals,” it should be clear that only passwords are in wide use as a *primary* form of authentication. There are exceptions, of course. For example, biometrics in the form of a fingerprint or face scanner have become one of the most common ways to unlock a personal mobile device.

Tokens and biometrics are used for *secondary* authentication. Using a secondary authentication mechanism is called *multifactor authentication* or MFA. The idea is that if a user requires two factors to be successfully authenticated, then it will be harder for an attacker to compromise both authentication methods. The attacker might be able to steal a password, but not the password and the token. Or, the attacker might steal the user’s phone for authenticating with something you have, but will not know their password. In the story of Cinderella, as told by Disney’s animated classic, the Wicked Stepmother succeeds in destroying the first glass slipper. Fortunately for Cinderella, she had the *other* slipper, providing a two-factor authentication of both something you have and something you are.

This is a specific example of the more general computer security design principle of *defense in depth*. This principle posits that an attacker should generally not be able to obtain an unauthorized objective preferably at all, but at least in part, when a single defensive mechanism is compromised. This principle is widely emphasized in modern computer security deployment [121]. Accordingly, most security experts believe that multifactor authentication is generally preferable.

However, when deploying MFA, it is important to think carefully about the goals of the system and the security context. In almost all circumstances, it is expected that *two different* factors will be used for MFA. It is generally not a good idea to use two passwords. As I discussed in the section on something you have, I do not consider codes sent via email to count and should be considered another form of password protection.

But beyond just using two different factors, it is important to understand the context in which they will be used. For example, if a user needed a password and a hardware fob to use an ATM, it might not produce the desired result. Although it would protect against skimmers and over-the-shoulder spying, it might also encourage armed robberies.

It is also important to think through remediation channels. If a user has to have a token, what options do they have to gain access if they lose their phone? Imagine if a user trying to call and lock their account after a robbery was unable to do so because the service professional would not authenticate them. When deploying a system, it is necessary to plan for failures. They will happen. Attackers will take advantage of a system that cannot fail safely and gracefully. It is, however, important to remember that MFA is not infallible just like the other technologies we discussed; I have included some reference for further reading about MFA flaws.

Summary

Authentication is the starting point for many security technologies and systems. Many security goals assume that different principals can be identified. The next chapter dives into assigning *permissions* to a principal once their identity has been verified.

In reading about the three primary mechanisms for authentication (i.e., something you know, have, or are), you may have noticed that there is no perfect solution. In fact, passwords are considered to be one of the worst solutions but are yet the most widely deployed. Despite constant announcements that “the password is dead,” there simply has not been a universal replacement for it. In evaluating authentication technologies, a good starting point is to figure out which one or ones of these approaches it uses and then start to analyze how well it deals with these weaknesses. No vendor will be able to eliminate fundamental and inherent limitations, but the mitigations, trade-off selections, and failure handling components will define how well it will support your security goals and provide a mechanism for comparison to other vendors’ offerings.

This chapter also spent a little bit of time talking about passwords because they are so common. In the first place, it is important to understand how passwords are stored. This is useful not only for understanding solutions being offered to you and your organization but also for how attackers crack passwords. This, in turn, will help you to understand and propose wise password policies for yourself and others. It is probably best to get a password manager for storing your passwords, of which there should be one per website. Most password managers can help you generate random passwords for maximum strength.

Further Reading

The name of the research group at CMU that does password research is CyLab Usable Privacy and Security Laboratory (CUPS). They have a wide range of research and reading materials. Many of these are articles, such as blog posts, that are understandable even to those with nontechnical backgrounds. You can find their password home page at http://cups.cs.cmu.edu/passwords.html. I highly recommend following this group because their conclusions are based on empirical research.

Within the scope of this book, I do not go into more details of how these systems are attacked, especially in MFA contexts. While MFA is seen as the very best way to protect systems and accounts, it can, of course, be defeated. A recently published book, *Hacking Multifactor Authentication*, by Roger Grimes digs into the author's tests of 150 different real systems and all the ways they can be exploited. This would be a good reference on how things go wrong, but it is also a practical book with specific recommendations for purchasing and deploying an MFA solution [119].

References

17. The quantum computer and its implications for public-key crypto systems. Technical report, Entrust Datacard, 2019.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

55. Beres, Y., A. Baldwin, M.C. Mont, and S. Shiu. 2007. On identity assurance in the presence of federated identity management systems. In *Proceedings of the 2007 ACM Workshop on Digital Identity Management (DIM'07)*, New York, 27–35. Association for Computing Machinery.

85. Cranor, L. 2016. Time to rethink mandatory password changes.

110. Franklin, B. Benjamin Franklin quotes.

114. Grassi, P., J. Fenton, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y.-Y. Choong, K. Greene, and M. Theofanos. 2020. Digital identity guidelines: Authentication and lifecycle management. Special Publication (NIST SP) 800-63B, National Institute of Standards and Technology, Gaithersburg.

117. Greenberg, A. 2021. The full story of the stunning RSA hack can finally be told.

119.

Grimes, R.A. 2020. *Hacking Multifactor Authentication*. Wiley. [Crossref]

120. Grother, P., W. Salamon, and R. Chandramouli. 2013. Biometric specifications for personal identity verification. Special Publication (NIST SP) 800-76r2, National Institute of Standards and Technology, Gaithersburg.

121. Group, J.T.F.T.I.I.W. 2020. Security and privacy controls for federal information systems and organizations. Special Publication (NIST SP) 800-53r5, National Institute of Standards and Technology, Gaithersburg.

130. Havenridge, J. 2015. Passwords are like underwear.

136. Honan, M. 2012. How apple and Amazon security flaws led to my epic hacking.

166. Luostarinen, K., A. Naumenko, and M. Pulkkinen. 2006. Identity and access management for remote maintenance services in business networks. In *Project E-Society: Building Bricks*, ed. R. Suomi, R. Cabral, J.F. Hampe, A. Heikkilä, J. Järveläinen, and E. Koskivaara, Boston, 1–12. Springer.

181. Merriam-Webster. Identity.

182. Metz, C. 1999. AAA protocols: Authentication, authorization, and accounting for the internet. *IEEE Internet Computing* 3(6): 75–79. [Crossref]

187. Morrison, S. 2020. Hackers stole $13,103.91 from me. Learn from my mistakes.

223. Ross, R., S. Katzke, and L. Johnson. 2006. FIPS-200. Minimum security requirements for federal information and information systems. Technical report, National Institute of Standards and Technology Federal Information Processing Standards (NIST FIPS), Gaithersburg.

251. Stallings, W. 2016. ICAM: A foundation for trusted identities in cyberspace. *IT Professional* 18(1): 26–33. [Crossref]

257. Swanson, M., J. Hash, and P. Bowen. 2006. Guide for developing security plans for federal information systems. Special Publication (NIST SP) 800-18r1, National Institute of Standards and Technology, Gaithersburg.

260. Thakur, M.A., and R. Gaikwad. 2015. User identity and access management trends in it infrastructure—An overview. In *2015 International Conference on Pervasive Computing (ICPC)*, 1–4.

269.

Ur, B., F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L.F. Cranor, H. Dixon, P. Emami Naeini, H. Habib, N. Johnson, and W. Melicher. 2017. Design and evaluation of a data-driven password meter. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI'17)*, New York, 3775–3786. Association for Computing Machinery.

275. Verizon. 2017. Verizon 2017 data breach investigations report.

Footnotes

1 I have seen this written as AAA and AA&A.

2 More correctly, the work is entitled *One Thousand and One Nights*.

3 This story was added to the collection by a French translator in the 18th century. For an excellent modern rendition, I suggest Hallmark's movie *Arabian Nights* from the year 2000.

4 This primitive form of password had no identity to claim. Possession of the secret word was claim on a *group identity*. The password and the identity claim were one and the same.

5 Zero-Knowledge Password Proofs (ZKPP) do not require the authentication system to know the password. However, these are not commonly used at present.

6 There are other ways to store passwords safely, but hashes are the typical approach.

7 The SHA-256 hash of "Open Sesame" is `0x7cb8 9be2 6325 3e21 9666 1f30 3926 cfb1 bc66 2a60 7220 486a 6926 6dff b737 af02`.

8 Picking the right password management depends on many factors. Please consider consulting with a security professional to determine the best one for

your situation.

9 As a security expert, I find Warbucks's lack of investigation and background checks frustrating, fictional though it may be.

10 There are many versions of Cinderella told in many cultures and in many languages. The retelling I make here is from the animated Disney classic.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_3

3. Authorization Technology

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
Building on top of authentication concepts from last chapter, this chapter presents how authenticated users are given permissions within a system. This is also known as *authorization*. Authorization is also a great starting point for learning about *security policy models*. These models are conceptual structures that provide a framework for understanding how to think about the security of a system. One of the earliest models is known as Bell-LaPadula (BLP). BLP, and a similar model named Biba, provides some good groundwork for authorization and security models. More modern policies tend to fall into a policy family, such as Domain and Type Enforcement (DTE), role-based access controls (RBAC), and attribute-based access controls (ABAC). In most computer systems, authorization policies are implemented using access controls that determine the appropriate permissions for an authenticated user and a given computing resource.

Key Concepts

1. Authorization

Common Pitfalls and Misunderstandings

1. Your concept of policy may not be the same as what is in this chapter; review carefully.
2. There is a temptation to jump to security mechanisms (e.g., a firewall) instead of starting with policy.
3. Policies are often weak at the “edges,” which are exceptional cases, rare events, or elements that simply are not defined.
4. Policies should generally not be created from scratch but be built or repurposed from existing policies.
5. Keeping an authorization policy operationally secure is complicated by many factors including complexity and *data rot*.

Useful Vocabulary

- **Access Controls**: Technologies that enforce or support authorization (permissions) for users
- **BLP**: Bell-LaPadula authorization model
- **Biba**: Biba authorization model
- **DTE**: Domain and Type Enforcement models (note the plural)
- **RBAC**: Role-based access controls
- **ARBAC**: Administrative role-based access controls
- **ABAC**: Attribute-based access controls
- **MAC**: Mandatory access controls[1]
- **DAC**: Discretionary access controls
- **TCB**: Trusted computing base; the part of the system that *must* work for security to be guaranteed

In the previous chapter, you learned about *authentication*: how principals that use a system are identified. But once a principal is authenticated, the system still has to decide *what they are allowed to do*. This is sometimes described as the principal's *permissions*. Authentication almost always has to happen first because the principal's permissions are almost always determined by their identity. Permissions are managed by *access controls*, and many technical references will refer to these concepts as access control rather than authorization. I will continue to use the term "authorization" throughout to describe the more general concept of determining a user's permissions and the term "access controls" to describe the technical components that provide the authorization service and manage a principal's permissions in a given system.

Please note that in the last chapter I discussed the authentication of primarily human principals. In this chapter, principals may refer to human or nonhuman participants. Many of the authorization approaches are still directed to humans or programs that operate as their avatars in the computer system. Some, however, do not and in many cases the system works the same regardless. However, if necessary to distinguish between human and nonhuman, I will do so.

In what might sound like another repeat from the previous chapter, authorization technologies are ubiquitous. They are also problematic and are often the source of many security issues. These problems stem from both inherent technological limitations as well as human errors in configuration, management, and deployment.

In this chapter, I will first introduce the concept of a security policy and why they are so important (and often misunderstood). The next section introduces you to a number of authorization security policies to illustrate the power of conceptual modeling as well as inherent challenges in authorization that decades of modeling have illuminated. Finally, the chapter concludes with an overview of different types of access control mechanisms that can enforce authorization security policies.

Computer Security Policies

What is a security policy? For the purposes of this book, I will use Ross Anderson's definition of *security policy* from his book *Security Engineering* [40]. Anderson helpfully describes what a security policy is *not* despite corporate and political uses to the contrary: a policy is *not* vague platitudes.

Anderson gives the following bad examples of "policy" [40, Chapter 9]:

1. This policy is approved by management.
2. All staff shall obey this security policy.
3. Data shall be available only to those with a "need-to-know."
4. All breaches of this policy shall be reported at once to security.

You may have seen such a policy in your professional experience. You may have *authored* such a policy. These are *not* the type of policies I am referring to.

Anderson more technically calls these security policies *security policy models.* He defines a security policy model as "a succinct statement of the protection properties that a system must have." And specifically, by succinct, he clarifies that such a document is typically "a page or less." The purpose is to have the "protection goals of the system [agreed upon]...” [40, Chapter 9]. One of Anderson’s examples for a security goal or property is “all transactions over $1,000,000 must be authorized by two managers” [40, Chapter 1].

Another way of thinking about security policy is it helps to provide a concrete definition of what security *means* for a given system. A security policy defines when the system is in a secure state or an insecure state in a concrete way [60, Chapter 4]. So, using Anderson’s example in the previous paragraph, if all transactions over $1,000,000 have been authorized by two managers, the system is in a secure state according to that policy. On the other hand, if not all transactions over $1,000,000 have been authorized by two managers, the system is not in a secure state according to that policy.

Because security policy is focused on the goals of the system, it is not an *implementation* document. It does not describe *how* the goals are to be achieved. The enforcement of the policy is the job of specific technologies, operations, and procedures deployed for securing the system. Anderson calls this concept the *mechanism*. A real-world deployed system should have a *security target* that provides a more comprehensive description of the protection mechanisms that the system has implemented. The security target ties security policy goals to concrete enforcement controls.

Often, when working on securing a system, the stakeholders will jump directly to the mechanism. Some new technology will be taking the world by storm, and the powers that be in an organization will be certain that this new technology will “make them more secure by an order of magnitude.”[2] However, if the correct security policies (e.g., security goals) are not used or if they are not understood, the mechanism will typically not secure the right things or will not secure them in the right way [40, Chapter 1].

Policy is important to get right as much as possible before moving on to enforcing it with mechanism. Given that we will *define* our security for a system by the policies we specify, it is critical that we *pick the right policies*. And it is harder than you might think.

A security policy can target just about any property of a system. In this chapter, you will see a number of policies that are primarily authorization policies. Most of these policies have been in use for *decades* and have generated a wealth of knowledge and wisdom for security professionals. In studying these policies, you will learn both about authorization concepts as well as more about security policies in general.

A Survey of Authorization Policies

One of the challenges in coming up with an authorization policy is dealing with all of the different permissions that may need to be assigned. Computer systems and devices are constantly gaining new features and new functionality. Devices are also working together with other devices (including the cloud) at an ever-increasing rate. Increasing functionality and increasing device collaboration drive an exploding number of permissions that have to be assigned and managed.

In cybersecurity, complexity is often a source of system *vulnerabilities*. Complexity increases the risks of errors, inaccuracies, and misuse, all of which are vectors for attackers to gain unauthorized access to systems. It is bad enough when a vulnerability exists anywhere in a system, but it is especially problematic if the vulnerability is part of the *security* component of the system. You will learn about the basics of a common weakness called a *buffer overflow* in Chapter 7. These vulnerabilities are found everywhere in computer systems, and attackers use them to get into systems far too often. But as bad as they are, there are mitigations including additional layers of security that can be wrapped around the system. A vulnerability in a security system, however, is far more severe. In some cases, this kind of vulnerability means that the system security may be completely bypassable, or it may even be effectively disabled. Worse, there may not be any obvious indication that the security system is in such a state.

The complexity issues are exacerbated by the care that must be exercised in the process. The assignment of permissions should be carefully thought out along proven guidelines. The security principle of *least privilege*, for example, teaches that every authorized user should have only the *minimum* amount of permission necessary to do their job. Following this principle reduces the amount of damage one person can cause if they go rogue (choosing to act against the interests of the organization) or if their account is compromised. However, this principle also leads to breaking up large permit-everything permissions into smaller, more granular permissions. Correctly allocating the right permissions, *and only the right permissions*, to every user of the system can be difficult. Worse, maintenance is even more of a nightmare because the system is not static. Users can be promoted, change departments, or leave the company. New systems introduce new permissions, and, in some cases, old permissions might be obsolete *but not removed*.

Many data breaches have been directly caused, or at least exacerbated, because of users having inappropriate permissions [78]. Former employees, angry at termination, have used access that should have been disabled to steal from or harm the organization. Parties without a need-to-know have used access that never should have been granted to look up personal information on employees or clients. Computer systems with misconfigured permissions (remember! they have permissions too!) have enabled high-sensitivity data to “leak” into lower sensitivity environments including publicly available web pages!

Story Time: Blame the Customers
As more and more data moves to the cloud, misconfiguration is a growing problem. Microsoft offers a framework called *Power Apps* that enable organizations to quickly develop “professional” web applications. Users of this framework include state and city government agencies, banks, and airlines. However, in 2021, the security group UpGuard found that many web applications that use Power Apps were incorrectly configured, and sensitive data could be easily extracted from them.

It turns out that Microsoft configured the databases in Power Apps to be *publicly accessible* by default with an expectation that the customer would configure the appropriate security. Many customers had not configured them at all, and the security researchers were able to extract sensitive data such as social security numbers, Covid contact tracing, and email addresses. Microsoft has since modified the framework to *block* all access by default and require users to explicitly allow access [280]. This also highlights another important principle of secure design: fail-safe defaults, which means that systems should be configured to be secure by default; in this case, it means to block access by default.

Authorization models deal with the complexity challenge by attempting to reduce the granularity. Rather than try to deal with every user and every permission individually, the model creates larger groupings that users and permissions are assigned to. The models then reason about how these groups interact with each other in a simpler manner.

The bad news is that *reality* is not simple, and these policy models break down at the edges. A breakdown at an "edge" is not necessarily any better than a breakdown in the middle. Security is not usually measured on a linear scale where securing twice as many entrypoints means the system is twice as secure. Usually, if there is *any* security hole, the attackers will find it. Security holes show up most often at a "boundary" or "edge" of some sort, and attackers have an amazingly good historical track record of searching boundaries and finding the holes there.

In the following sections, I will walk through the good, the bad, and the ugly for a number of authorization policies. Let's find out what they do, where they are good, and where they break down.

Bell-LaPadula

Our first stop on our tour of authorization policies is the *Bell-LaPadula* model, often abbreviated as BLP. BLP was first proposed in 1973 by Dave Bell and Len LaPadula [52]. BLP is a policy designed to support government and military systems that require *multilevel security.* In multilevel secure (MLS) systems, a given computing resource (data, system, program, etc.) is given a security *label*. The resource is called an *object*, and the labels represent an ordering of sensitivity. For example, the labels might be "Unclassified," "Confidential," "Secret," "Top Secret," and so forth. Labels could also be numbers. Labels for objects are also called "classifications." Not only are objects in the system given labels but so are *subjects* (actors) within the system. You might think of *users* as actors, but users are represented within the system by the programs they run. Under most circumstances, when a program is started by a specific user it includes the identity and permissions of the user that started it. From an authorization perspective, a program running on a system is an incarnation of the user that spawned (launched) it.[3] The labels attached to subjects are also called "clearances." Because it may be confusing, let me repeat a key point. When a program is running that was launched by a user, it is *running with the user's identity*. If two users launched the same program on the same computer, each would be separately identified by the initiating user. Different access controls could be applied to each running version of the program. Thus, the user, and not the program, is the subject. It is possible to have programs that are launched automatically and not tied to a specific human user. Access controls apply to these as well and are an example of authorization applied to nonhuman principals. However, for the discussion of BLP, it is easiest to think of the subjects as humans.

To summarize, in MLS systems, both the subjects and the objects have a label. The authorization policy defines what a subject with a given label may do with objects of a given label. For example, can a user with a label of "Secret" read a file with a label marked "Top Secret"? In government contexts, the answer is "no." Information is supposed to be controlled so that nobody can access, or ideally even be aware of, information at a higher security level.

The BLP policy was designed to protect government MLS systems by enforcing the flow of information downward from higher sensitivity to lower. To do this, BLP included three protection properties:

1. **The Simple Security Property**: A subject can only read data from objects with the same security level or lower. That is, they cannot read data that requires higher security than they are authorized to access. This is often called *no-read-up* (NRU).

2. **The *-Property**: A subject can only write data to objects with the same security level or higher. That is, they cannot write data to a lower security level, as this might cause information known from a higher security level to leak. This is often called *no-write-down* (NWD).
3. **Discretionary Security**: A subject can only read data from objects if an access control matrix grants them permission to do so.

The no-read-up and no-write-down properties, illustrated in Figure 3-1, are the key ideas from BLP (I will get back to the discretionary security property later). No-read-up is probably intuitive to you. After all, if you are assigned the label “Secret,” you probably should not be reading from documents marked “Top Secret.” But no-write-down is less obvious. If you have the label “Secret,” why can you not create a document marked “Confidential”?

The answer is that BLP was designed to prevent information leaks. BLP assumes that a user’s account can be compromised, that the user can make mistakes, or that the user can become the attacker! The no-write-down property prevents more sensitive information from leaking downward by either accident or malice.

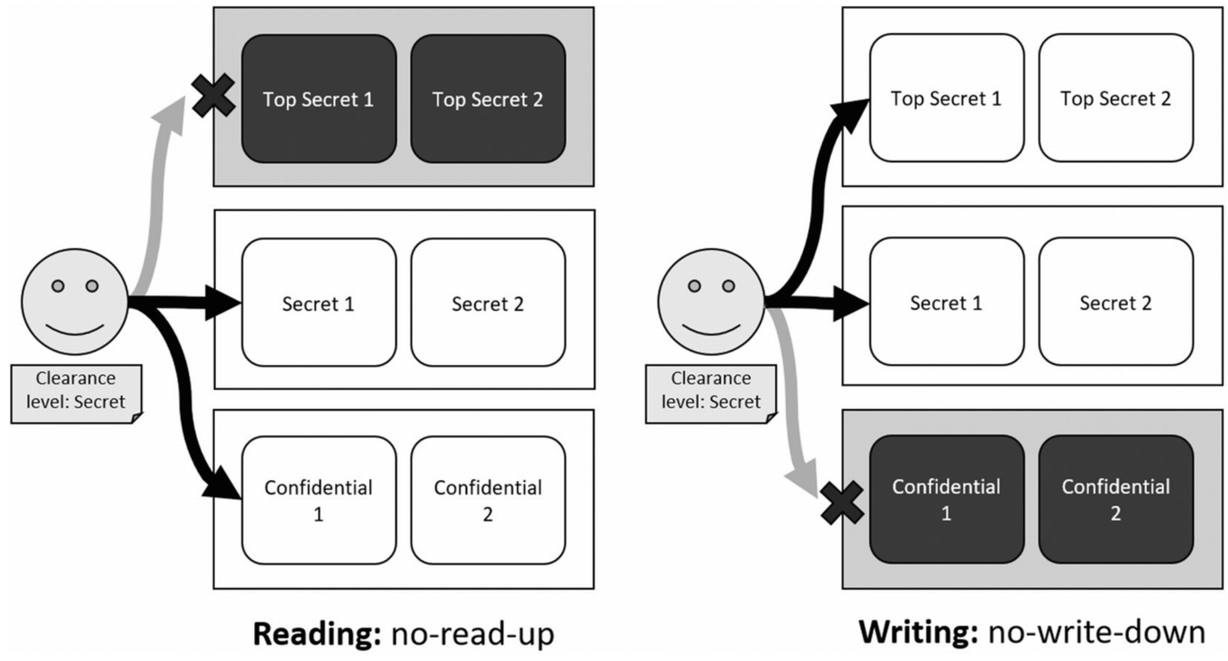


Figure 3-1 Bell-LaPadula model for information confidentiality. A principal can read objects at or below its clearance level, but it can only write objects at or above its clearance level

Story Time: Unintentional Longevity

The *-property, which is pronounced the "star property," got its name by accident. According to Bell:

> A condition to prevent deleterious flows was easily formulated, but a descriptive name was elusive. When I first raised the idea, I scribbled the heading "*-property" on the blackboard over a figure much like figure 2. After a burst of energetic discussion, I pointed out that if we didn't change the name right then, we'd be stuck with it forever. Nothing came to us and we continued our discussion. "*-property" it remained. [51]

If you are not working for the government or a government contractor, you might be tempted to ignore BLP. After all, what good does BLP do in contexts without top secret data? You have to remember that BLP was created in the 1970s when government and military computer systems were largely the only space in which security was really being taken seriously. This was going to be the birthplace of authorization policies.

But it *was* the birthplace. As I continue your tour of policies throughout the rest of the chapter, including policies found in commercial systems, you will see the evolution and development of authorization ideas that all started with BLP and the lessons learned from it. Understanding BLP, the problems it was trying to solve, and its strengths and weaknesses will better enable you to understand authorization policies in place today.

One lesson to take away from BLP is the concept of *mandatory access controls* (MAC). A mandatory access control is any access control that is defined and enforced based on a security policy. MAC stands in contrast to *discretionary access controls* (DAC) wherein a user can decide at their *discretion* what the access controls on an object should be. In many commercial systems, a user can take a file they have created or have access to and give access to other users within the system. Cloud systems allow the owners of a file to give access to other users in order to share data. You are most likely using DAC on your personal computer. All of these are examples of discretionary access controls. The user determines who does and does not have access.

MAC systems enforce a security policy on access controls regardless of the user’s wishes. The no-read-up property and the no-write-down property are considered mandatory controls. The system will enforce these rules at all times.

BLP does include a discretionary security property as well. This permits a subject with appropriate permissions (and of sufficient security label) to restrict access to the data even to other subjects with what would be permitted access under NRU or NWD policies. Often called “need-to-know” access, the idea is that even if you *could* access the data from a clearance perspective, maybe you *should not*. Nevertheless, these controls are discretionary. What subjects can and cannot see is based on this property and is determined at the discretion of the users of the system.

From Anderson’s perspective, the BLP policies are neither mandatory nor discretionary. They are policies, the goals of the system. The enforcement is mechanism. Anderson describes the systems that implement BLP policies as mandatory.[4] Other sources describe BLP as having two mandatory policies and one discretionary policy. I prefer Anderson’s formulation. Conceptually, there are just the policies (protection goals). The entire system should follow the policy; how it is enforced is up to the implementation. I will follow this approach for the remainder of the book.

Most of the other authorization policies I will discuss in this chapter require mandatory access controls to effectively implement the policy. Understanding BLP is a good way to get started in understanding what MAC is and why it is necessary.

BLP as a policy has some powerful characteristics. Most importantly, perhaps, is that BLP’s model is relatively simple and easy to understand and yet has significant security strength. At the policy level, there is much that can be reasoned about in terms of the protections provided. Another useful characteristic is that the properties are amenable to implementation, and it is relatively easy to figure out if the implementations are correct.

On the other hand, the BLP policy has some interesting limitations. The policy does *not* specify how to deal with a number of crucial issues. It does not, for example, deal with how data *should* be allowed to move from a higher classification to a lower one. In other words, the BLP policy does not explain how to declassify information or how to produce a redacted version.

To be clear, Bell and LaPadula did describe a system that dealt with these issues. But they did so by largely pushing the problems to the implementation. For example, in the BLP formulation, they describe “trusted subjects” that are not subject to the NWD property. In fact, in their formal description, the NWD property was stated to only apply to untrusted subjects. What makes a subject trusted? There were no formal requirements. They are simply described as not violating security policies.

Another way of thinking about “trusted” is that it refers to any part of the system that will result in a security failure if it is broken. These are the parts of the system that enforce security everywhere else or are otherwise required for the security to work. They are called “trusted” not because they *can* be trusted but because they *must* be trusted.

Because every system must have some trusted components, the term *trusted computing base* (or TCB) refers to the collection of all of such components, including, potentially, people such as trusted subjects.[5] In a security system, an analysis of the TCB is always necessary because, again, if it breaks, so does the system’s security. One desirable property of a TCB is that it should be *as small as possible*. The smaller the TCB, the better. Every piece of technology above a certain complexity has unknown bugs (computer errors), which means that there is always a hidden way that a system’s security can fail in almost every system currently running on the planet. A good portion of the battle between security experts and attackers is to see which side finds the problem first. In any event, the smaller the TCB, the fewer the bugs.[6]

Categorizing a subject as “trusted” in order for BLP to function adds risk to the system. In the first place, all of the reasoning about the security properties of BLP are thrown out the window for these subjects. Second, it increases the size of the TCB. Third, and finally, because there are no hard requirements for the trusted subjects, there is a danger for system designers to take anything that is hard or difficult about BLP and simply have a trusted subject perform the necessary operations outside the policy.

BLP also does not say how, for example, subjects or objects are created or how they are assigned their initial labels, nor does it discuss how these labels might be changed. Again, the practical BLP system that Bell and LaPadula introduced contains this functionality. But the concepts are not in the policy and cannot be evaluated from that level.

This was partially addressed by adding the *tranquility* property to BLP. *Strong tranquility* requires that labels on either subjects or objects do not change during normal system operation. Labels may only be set or changed outside of normal operations, such as having the system shut down and only accessible by a security officer or team. Although changing the labels is still outside the model, risk is reduced by only allowing such changes in a special, low-risk mode.

Alternatively, *weak tranquility* requires that labels are not changed in a way that violates some specified security policy. For example, a common policy for controlling weak tranquility is to permit a subject to start out at the lowest security level even if they have higher clearance. Remember that in BLP, the subject is the *running program*, not the actual user themselves. So the program starts out at the lowest level; if it needs to read higher data and the subject's ultimate clearance permits it, the program's label is adjusted accordingly. Notice that at any given point in time, a subject cannot write down higher data than it can read. BLP is not violated and the weak tranquility property is enforced according to this policy.

BLP has another problem: *when it works too well*! I say this with only a little sarcasm. In a BLP system, data is meant to move one way from low to high. Subjects that write high cannot even see the data they created and lose any control over it. So, when BLP is “working,” it can result in a system where data is compartmentalized and difficult to access or use. This reinforces the need for controlled reclassification and declassification that, to repeat, is not part of the BLP model.

Despite all of these difficulties, BLP was still a very good policy for authorization and continues to be used today in government circles. It was also analyzed, criticized, and debated ever since its formulation, and this, in turn, led to the creation of other models.

Biba

One such model, the *Biba* model, named after its creator Ken Biba, is our next sample policy [59]. This policy *inverts* the properties of BLP and requires both no-write-up and no-read-down! The two key properties of Biba[7] are written as

1. **The Simple Integrity Property**: A subject can only read data from objects with the same security level or higher.
2. **The * Integrity Property**: A subject can only write data to objects with the same security level or lower.

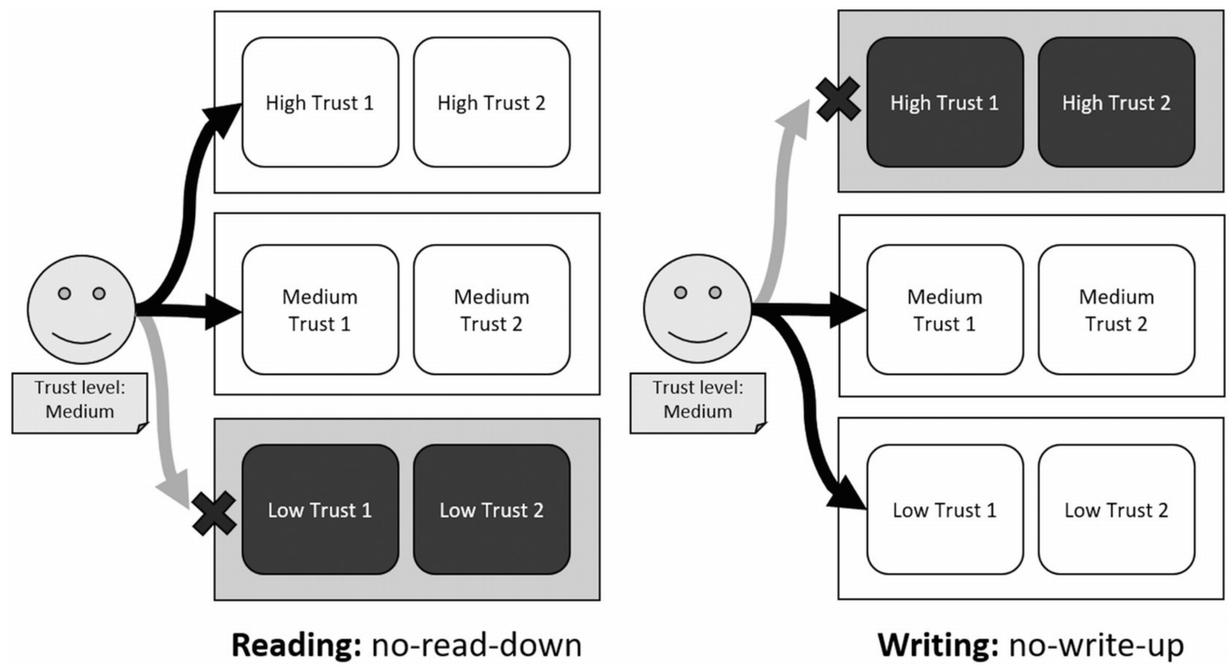


Figure 3-2 Biba model for information integrity. A principal can write objects at or below its trust level, but it can only read objects at or above its trust level

The limitations of these two properties, visualized in Figure 3-2, may seem strange. After all, why would you *allow* someone to read at a higher classification than their clearance?

The reason is that Biba is concerned with *integrity* and not *confidentiality*. For any given system, it is crucial to understand what needs to be protected and how. Ross Anderson explains that, "many systems fail because their designers *protect the wrong things*, or protect the right things *but in the wrong way*" [40, Chapter 1], emphasis added. Even though BLP has some excellent properties, they would be the *wrong* properties if those properties do not match the requirements of given system.

While there are many security properties that a system might want, three very common ones are *confidentiality, integrity, availability*, also known as the CIA triad. Confidentiality generally refers to keeping secrets secret. More accurately, confidentiality means that data can only be *read* or accessed by authorized parties. Integrity, on the other hand, refers to keeping data protected from malicious changes. It means that data can only be *written*, or changed, or deleted, by authorized parties. Finally, availability means that authorized parties cannot be kept from accessing or modifying data for which they are authorized to do so [124].

BLP is concerned with confidentiality—keeping data secret. It is concerned about what can be read. Even the no-write-down policy is not about writing per se, it is about making sure data from a higher level will not be read at a lower level.

Biba is not concerned with keeping data secret. The model is focused on ensuring that data cannot be modified by unauthorized parties, that is, integrity. The higher the classification of the data, the more protected it needs to be from edits.

The Biba policy, first proposed in 1975, found its way decades later into the Windows 7 operating system. Starting with this version of Windows, files are marked with a label from the set: Low, Medium, or High (also called System). Any files fundamental to the operations of the operating system are marked High. All other objects are marked Medium. The Internet browser is marked with a Low label. Anything downloaded by the browser is also marked Low. Windows 7 only implemented the no-write-up policy of Biba, but, in theory, this should prevent a downloaded virus or trojan from modifying any important files on the computer [40, Chapter 9].

The problem with Biba in Windows, however, is that there are just too many exceptions. Users download programs that need to operate like “normal” programs (and have at least the Medium label). Windows enabled this by permitting users to grant exceptions, but this just trains users to “click through.” Even if they did try to read and understand the warnings, this simply adds the burden of understanding which downloaded programs are and are not threats. It also eliminates the mandatory access controls and replaces them with discretionary.

You may have noticed that this is not unlike the problems in BLP. That system has to deal with exceptions too and does so with a mix of techniques that are outside the model, such as trusted subjects. This is a common problem for authorization policy. There almost always needs to be an exception to the policy. But exceptions, however necessary, are holes in the policy that become the common targets for the attackers. Many policies live or die not by their *normal* use but by how well they handle and protect *exceptional or boundary* conditions.

Domain and Type Enforcement

In the third stop on our tour, we get into more modern policies. Although, as you will see, this section is actually about a *category* of policies, rather than any single policy itself. This class of policy is called *Domain and Type Enforcement*, or DTE.[8] This model framework generalized classifications into a more powerful system that could support more advanced security operations [47]. Instead of relying on a linear up-down classification scheme, objects are assigned a *type*. Similarly, subjects (which, as with BLP, are running programs) are assigned a *domain*. An internal table of domain-type and domain-domain mappings defines what permissions are permitted to each domain for each other domain or type.

For example, in terms of domain-to-domain permissions, DTE policies can express that a program running in one domain is permitted to start, stop, or communicate with another program, either in the same domain or in another. On the domain-to-type side, permissions include read, write, and execute permissions.[9] Domain-to-domain privileges are specified within the Domain Interaction Table (DIT), and domain-to-type privileges are specified within the Domain Definition Table (DDT).

You should note that although each subject has a domain and each object has a type, there need not be very many different domains or types within the system. Each domain or type represents a class, not an individual. This overwhelmingly simplifies how many access controls must be specified. On a system with just 20 runnable programs and 100 files, there could be up to 2000 ($20 * 100$) permissions to specify, as each runnable program would need permissions specified for each individual file. But if in the same scenario there were only three domains and ten types, there would be at most only 30 ($3 * 10$) permissions to specify.

Moreover, DTE simplified things further by permitting *implicit typing*. DTE took advantage of file hierarchies to enable this. You may already be aware that you can organize your files on your PC or smartphone into folders and subfolders. But many of today’s nontechnical professionals are not aware that *all* of the files on their devices are organized into a hierarchy of folders (another word for folder is *directories*). Every file has a *path* from some root folder down to whatever subfolder immediately contains the file. DTE used this organization and assigned type permissions to directories (folders). Permissions assigned to a directory were propagated down to all subdirectories and files within it. However, the permissions could be overridden within subdirectories.

Using this kind of typing, DTE did not require specifying types for every single file. It only required specifying types on folders and letting the files within adopt that type.

In terms of the security capabilities provided, DTE can create policies that are more powerful than BLP. In fact, BLP could be expressed using DTE with proper configuration of domains and types. DTE can express confidentiality policies and integrity policies and also enable far more complicated concepts such as *assured pipelines*. This refers to when multiple programs interact with each other (e.g., when you download a file on your browser and then it launches another program to view it). DTE enabled limiting running programs within a pipeline to only communicate with the previous program in the pipeline and the next. This could inhibit a rogue program from trying to send data to an unauthorized party during the transformation of data.

DTE as a framework can be implemented in an operating system as a mandatory access control component. Any DTE policy can then be implemented on the system through configuration of the domains, types, and associated permissions. This means that a system using DTE for enforcement can switch policies to anything DTE-compatible without having to modify the operating system or programs running on the system.

You might be surprised to find out that you might be using DTE personally. The first prototypes to implement DTE were modified UNIX operating systems. Subsequently, a variant of the Linux operating system called SELinux (Security-Enhanced Linux) was designed and built to support (a minor variation of) the DTE policy. When Android was created for smartphones, it was based off of (and still uses a core of) SELinux. So if you have an Android phone, you are using a system with a DTE (or DTE-like) policy.

A Sample DTE Policy

Consider the following example DTE policy taken from a computer security textbook [60, Chapter 4]. Please note that I have rewritten the policy in a more narrative form so as to not require understanding of operating systems, programming, and other technical details.

This example policy restricts any user of the system from modifying the core programs of the operating system unless they have administrative access. This can be expressed in DTE with four domains (which categorize subjects):

- **User Domain**: For ordinary users
- **Admin Domain**: For administrative users
- **Login Domain**: For the login program (this program requires special controls)
- **System Domain**: For programs and services running in the background by the operating system

The policy also requires five object types (for categorizing files on the system):

- **Executable Files**: For program files that can be run
- **Readable Files**: For readable files
- **Writable Files**: For writable files
- **DTE Files**: For files used by the DTE enforcement mechanism
- **Generic Files**: For files created by user processes (any permissions)

For this simplified example, files are in exactly one of these types. A file with the type Writable Files might still be read from. The type is just a label.

The policy now requires rules. The System Domain has the following rules. Note the reference to an *init program*. Unix and Linux use a program called *init* to initialize the system and get it running.

1. The init program starts in the System Domain.
2. Subjects in the System Domain can create, read, write, or search any object with the Writable Files type.

3. Subjects in the System Domain can read, search, and execute any object with the Executable Files type.
4. Subjects in the System Domain can read and search any object with the Generic Files, Readable Files, or DTE Files type.
5. When the login program is launched (i.e., from the init program), it will transition to the Login Domain.

One of the key ideas here is the third rule that permits programs running in the System Domain to read, search, and execute files with the Executable Files type. Do you see what permission is *not* granted? *Create or write*! Remember that attackers sometimes break into computers by exploiting bugs or errors in computer programs. Sometimes, they can fully take over the program and make it do whatever they want. But notice that even if this happens to a program running in the System Domain, the attacker cannot modify files that are programs and can be run. Attackers would not be able to install (create) a Trojan horse file at the system level, nor would they be able to modify system programs (such as the login or init program).

The Login Domain also has some interesting rules. This domain is used exclusively by the login program. The reason for creating a special domain is because the login program serves as a kind of switching station. When a user logs in to the system, the computer starts some kind of initial program on their behalf. In Windows and MacOS, this is typically the user's desktop. If you log in as a different user, you have a different desktop. This is a running program, and it runs with the user's permissions and identity. In Unix and Linux (applicable for this example), the user usually starts out in a shell program.

Any programs launched from the Desktop program or shell program will also run with the user’s identity and permissions.

The login program is configured to start the appropriate program for a user once they successfully log in. It also launches the user’s initial program with the user’s identity and permissions. In the DTE context, however, this also means *switching domains*. If the user is an administrator, this needs to be the Administrative Domain; otherwise, it needs to be the User Domain. So, for this policy, the Login Domain has the following rules:

1. Only the login program runs in the Login Domain.
2. Subjects in the Login Domain can create, read, write, or search objects of the Writable Files type.
3. Subjects in the Login Domain can read and search objects of the Readable Files, Generic Files, or DTE Files type.
4. Subjects in the Login Domain can change the User ID (so the user’s initial program starts under their identity instead of the login program’s identity).
5. Subjects in the Login Domain can execute programs in the Administrative Domain or User Domain.

An important part of these rules for the Login Domain is that the Login Domain cannot execute any programs within its own domain! Its entire purpose is to switch to another domain, and the rules of the domain enforce this limitation.

In the interest of simplicity, I will not walk through all of the rules for the other domains in detail. Subjects in the Administrative Domain *do* have permission to modify the system’s executable files. An administrative user, therefore, can install new programs, upgrade the system, and apply patches. Subjects in the User Domain can execute system programs but cannot write to them. They are also limited to creating objects of the Writable Files type or the Generic Files type (they should not be able to create system programs).

This example policy illustrates the power and granularity of DTE. The trade-off is that the power and expressiveness comes at the cost of complexity. The beauty of BLP was the simplicity. It is easy to reason about and simple to understand. On the other hand, DTE is so expressive that it can be difficult to develop and manage DTE policies. DTE does provide a policy specification language called *domain-type enforcement language* or DTEL. Policy authors can use DTEL to implement policies such as the example policy I laid out in this section.

Unfortunately, DTEL only makes it easier to write the policy rules. It does nothing to reduce the actual complexity. SELinux's DTE-like default policy specification is thousands of lines of text. The low-level nature of DTE rules and the sheer size of a policy specification make it very difficult to intuitively understand whether or not the DTE specification matches the conceptual policy. Returning to the earlier DTE example, the Anderson-style conceptual policy is simply "system programs may only be modified by an administrative user." To put that one-line statement into a DTE specification required dozens of low-level rules. To try and solve this problem, researchers have worked on tools that try to analyze the DTE expressions [232]. Nevertheless, this is a significant challenge.

One final note. This type of policy does clearly deal with nonhuman authorization. In fact, the `system` domain is meant for programs that are running with *system* privileges rather than the authorization of a human user. It includes a program called `init` that is meant as a system startup operation that would make little sense to carry permissions associated with a specific user. With that said, because human users are so ubiquitous in authorization, these kinds of domains are often associated with a pseudo user (e.g., the “system” user). In fact, it is generally possible to log in to these accounts to perform system maintenance. Nevertheless, the permissions are more associated with the system itself than a specific user.

RBAC and ABAC

The last stop on our authorization model tour covers two other model frameworks called *role-based access control* or RBAC and *attribute-based access control* or ABAC. Like DTE, these are not any specific policy but, rather, create a framework for creating policies of a certain type. RBAC and ABAC differ from DTE and the other policies on our tour in that they are meant to be updated regularly. Although they have much in common, I will discuss each one individually.

RBAC RBAC was first proposed in 1992 [105]. RBAC defines rules based on roles rather than user identity. A user may have many roles but will only access resources under one role at a time. For example, an employee might have a functional role for their normal job, a role as a trainee (for ongoing professional training requirements), a personal HR role for seeing paystubs and managing elections, and perhaps even a social role for company-sponsored activities and online events. In an RBAC policy, each role has permissions assigned to it, and the user signs in with a specific role to access resources. Imagine if the user is performing their usual job function and is signed in with that role. It is payday and the user wishes to view their paystub. The user would log out under their functional role and sign in using the personal HR role. RBAC can also be configured with a multirole hierarchy wherein some permissions are based on the relationships between different roles as depicted in Figure 3-3.

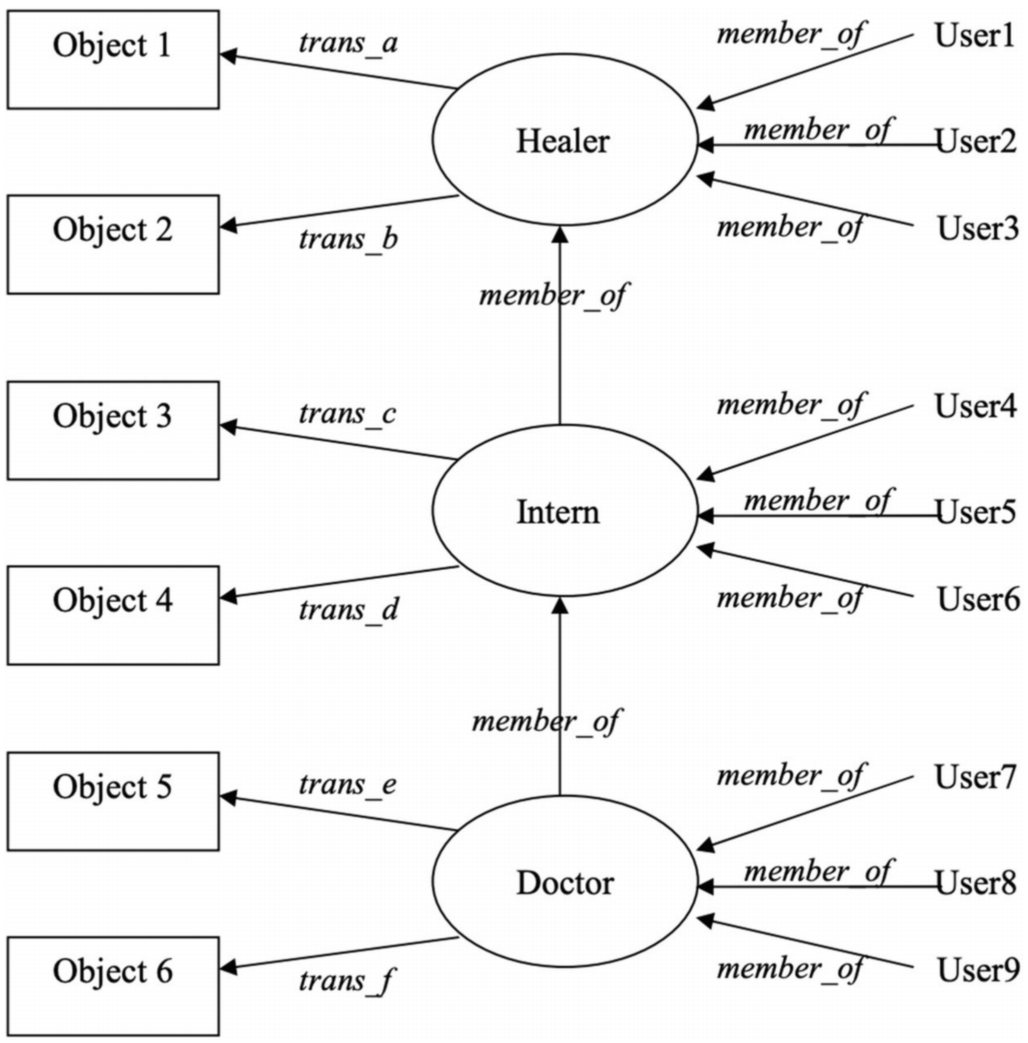


Figure 3-3 An RBAC multirole hierarchy wherein, for example, each member has access to all the objects at their own role and below them in the hierarchy. For example, in this case, the Doctor role has access to the transactions defined by the Intern and Healer roles. Included figure from "Role-Based Access Controls" [105]

Notably, there is a variant of RBAC known as administrative role-based access control or ARBAC [228]. ARBAC is the meta-level controls for RBAC in that it specifies how an administrator may change RBAC controls. As I said, RBAC is meant to change regularly. A person's role may change, or the need of a role to access a resource may change. ARBAC defines the authorization of these changes.[10]

There are a number of advantages to the RBAC approach. First of all, it helps create protective silos around the different ways someone uses cyberspace. If a user’s access for a given role is isolated from other functions, a security breach in one silo does not necessarily compromise the other. This also helps to separate risk. There might be a higher risk of compromise in a scenario such as a company activity (such as a networked gaming activity). Less secure computers will be in use, and less attention will be paid to security issues. RBAC helps to prevent higher risk activities associated with a role from increasing the risk of the other silos as well. Again, the principle of least privilege discussed near the beginning of the chapter is crucial.

Story Time: Super-Admin Surveillance

In March of 2021, attackers managed to infiltrate Verkada, a company that provides on-site security cameras. Even though the cameras are on-site, the video feeds are uploaded to Verkada. The expectation was that under most circumstances, only the customer should be able to view their own video feeds through cloud access. The attackers, however, compromised what are known as *super-admin* accounts. These accounts had complete access to all videos.

This is already bad. Verkada has stated that these accounts enable their technicians to assist with support requests from customers. However, generic super-admin accounts that can view all video, rather than a per-customer admin account, or some other kind of controlled access, are an unneeded vulnerability and risk.

But it gets worse. Not only did such accounts exist, but they were reportedly widely available within the company. According to the reports, at least 100 employees, including interns, had access to these super-admin accounts. Apparently, Verkada employees themselves were raising internal concerns about the sloppy access controls. In a previous scandal, Verkada male employees used Verkada's own internal cameras to take unauthorized pictures of some of their female coworkers [244].

The obvious moral to this story is that RBAC must actually be enforced. But a slightly more subtle point is that *internal* vulnerabilities often become *external* vulnerabilities. Attackers are effective at finding the weak points that organizations create for themselves.

Another advantage of RBAC is the ability to manage permissions as a user changes roles within an organization. If RBAC is used correctly, a user's role will be changed as part of the change of assignment or position. This means that previous authorizations that should no longer be enabled will be automatically turned off, and new permissions automatically provisioned.

It also makes creating an RBAC policy more manageable because the policy can be expressed in terms of organizational roles instead of specific users, technical classes (such as computer administrator), or other groupings. Thus, all permissions associated with a role can be modeled and understood independently of which individual people hold those roles. It also permits modeling of the *interaction* of roles within the organization. For example, the concept of *separating duties and concerns*, discussed earlier in Chapter 1, can be implemented and analyzed in an RBAC model. It is easy to verify that permissions that need to be partitioned are split across two roles.

An example [266] of a simple RBAC policy is

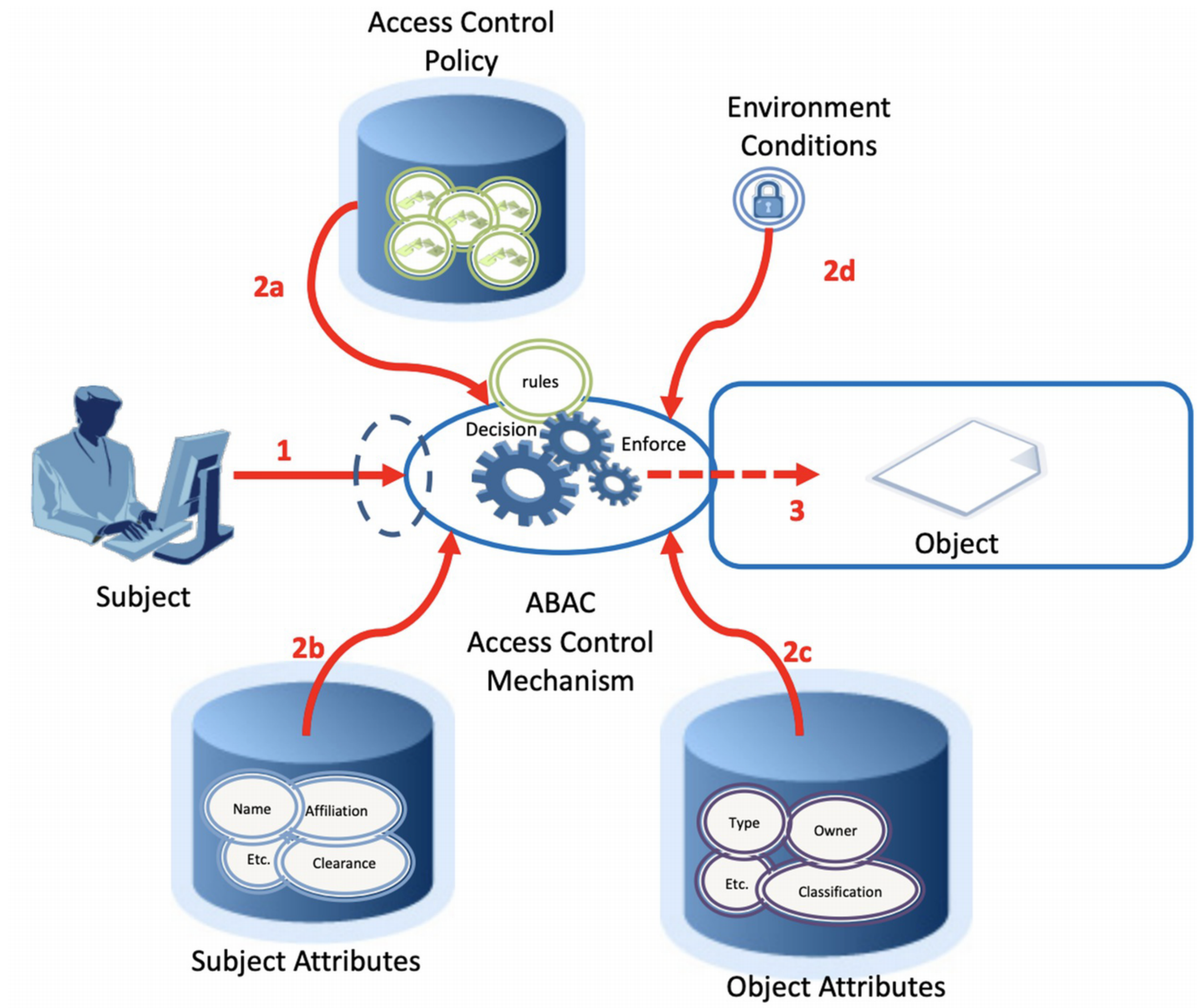
- **Software Engineering Role**: Has access to GCP, AWS, and GitHub
- **Marketing Role**: Has access to HubSpot, Google Analytics, Facebook Ads, and Google Ads
- **Finance Role**: Has access to Xero and ADP

- **Human Resources Role**: Has access to Lever and BambooHR

ABAC RBAC is still extremely common and widely used. However, a more general form of the concept, called *attribute-based access controls* (ABAC), is generally seen as more effective. ABAC permits access controls to be conditioned on any combination of attributes including role, location, project, and so forth as depicted in Figure 3-4. In December 2011, ABAC was recommended as the preferred access control mechanism for a security road map developed by a US government advisor group [74].

NIST defines ABAC[11] as follows:

> An access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions



1. Subject requests access to object
2. Access Control Mechanism evaluates a) Rules, b) Subject Attributes, c) Object Attributes, and d) Environment Conditions to compute a decision
3. Subject is given access to object if authorized

Figure 3-4 The different factors that are considered in the ABAC security configuration when a user requests access to a resource. Included figure from "Guide to Attribute Based Access Control (ABAC) Definition and Considerations" [74]

It is worth noting how much ABAC sounds like DTE. Subjects have attributes and objects have attributes. A unique set of subject attributes is more or less a domain, and a unique set of object attributes is more or less a type. ABAC is valued for its expressiveness. Here is an example of an ABAC policy for a financial environment [65]:

- A manager can view a transaction in their branch.
- No one can approve a transaction above their approval limit.

Here is another example that is described as an authorization example [65]:

- A manager can view any record.
- An employee can view a record in their own department.
- An employee can edit a record they own, if it is in draft mode.
- A manager can publish a record if the record is in final mode and it belongs to an employee below that manager.

Despite the expressiveness, or perhaps because of it, ABAC, like DTE, can be difficult to reason about. For example, it is difficult to evaluate how well a given ABAC policy is enforced by a particular mechanism because of the inherent complexity of the policy.

Common RBAC and ABAC Problems

RBAC and ABAC policies are very reasonable in an ideal world, but they sometimes break down in the real world. For example, an employee at a company will often take on a new role without completely being severed from the old one. That is, after an employee takes on a new assignment, it is overwhelmingly common that someone within the company will continue to ask them for help in their old role for some time afterward. Because of this, the employee, the people that ask for the employee's help, and IT are all hesitant to disable the old role. Instead of replacing the employee's old role, a new one is simply added. This is called *role creep*, and it can undo most of the advantages of RBAC (silos, modeling, etc.) while adding overhead and complexity that is hard to untangle.

The more general problem for RBAC and ABAC is the disconnect between the model and reality. The term *data* *rot* is used to describe data that is or becomes obsolete, out of sync, incorrect, or otherwise problematic. Rot is used to describe this problem because the incorrectness of data increases over time. Data stored in a system is static, but the real-life elements associated with the data change regularly. The disconnect between data and reality can be described as a *data quality* problem and can have significant impacts on the effectiveness of an organization in achieving its objectives [176].

Data quality is also critical for computer security in general and authorization in particular. The quality of an organization's data with respect to the people, systems, data, roles, and permissions in it determines to what extent the modeling relied upon in establishing system security is relevant or applicable.

In concluding this survey of authentication policies, let us note how important it is to understand and rely on models that have already been developed. In some cases of completely new technology, a new security policy must be created from scratch. However, most of the time, existing security policies can be used, adapted, or combined. In looking over these policies, you should be able to recognize all of the experimentation and research that has gone into them. If you create a new policy, it will take some time before you will have the feedback, evaluation, and *wisdom* to really understand the effectiveness of the policy conceptually and when implemented. Using policies that have already been beaten up and, even if bloodied, have stood the test of time is a better starting point.

Access Control Technologies

Once a policy or model is chosen, it must be enforced by a mechanism. Even DTE, which is often described as an enforcement mechanism, is still relatively high level.

Lower-level technologies enforce the actual access to a file or other resources. Access controls can be implemented at many levels, including the operating system, programs, or ancillary programs such as databases. These ancillary programs are sometimes called *middleware*, as they do not get used directly but provide support to other programs. Access controls are also increasingly network based for dealing with cloud storage and processing. The following concepts may appear in any of these levels, but for simplicity I will describe them from an operating system perspective.

Most computer systems can group concrete resource permissions into one of four categories: create, read, update, and delete. This common set of permissions is often referred to by the mnemonic *CRUD*. Some technology systems explicitly use CRUD to describe the permissions that can be assigned. Most of these are data storage systems such as databases.

Although other systems use different permissions, most of them can be categorized as a create, read, update, or delete permission, or a combination of two or more of these. For example, most operating systems have permissions that include read, write, and an explicit permission for *execute* that permits a user to run a program. Conceptually, however, program execution is a combination of read permissions on the file and update permissions on the processor. Running a program means copying (i.e., reading) a file out of memory and having the processor execute each instruction. It may be helpful to refer to Appendix B for more details.

In short, CRUD, while an explicit set of permissions for some data systems, is also a good *conceptual* model for most if not all other permissions that can be assigned.

For example, consider a *cyber-physical* system. A cyber-physical system includes physically manipulable components and includes examples such as robotic assembly line arms controlled by computer software. Imagine such a system in which only authorized technicians are allowed to manipulate the robotic arms manually. What category of permission does “move robot arm” fall into? Most likely, this is an example of an update permission. The movement of the robotic arm is just a *side effect* of changes to the *state* of the controlling program.

The term “state” in this context refers to an idealized, conceptual description of the internals of the program at a given point in time during its operation. The description is exhaustive. To reiterate, this is *during* operation. The state of the program can and does change constantly. Inputs to the program change internal data and, consequently, move the program into a new state. Inputs can be from other computers, physical sensors, or even humans. For example, many programs change state over time based on inputs based on a computer’s clock.

Cyber-physical systems, like any other program, have states. The only difference is that for these systems some changes in state trigger a physical effect. However, from a permission perspective, the operator of the system was granted permission to change the state of the controlling program or its data, and the physical operation was the consequence of that *update* of data.

From this discussion, you may have also noticed that you can have CRUD permissions for many operations within the same system. A system can have CRUD permissions on user accounts, files, device access, and so forth. Administrative users have permissions to create, read, update, and delete user accounts. And a new user created by the administrative user will have create, read, update, and delete permissions on some subset of files within the system. A user can also grant permissions to various apps on a phone to access the device’s camera.

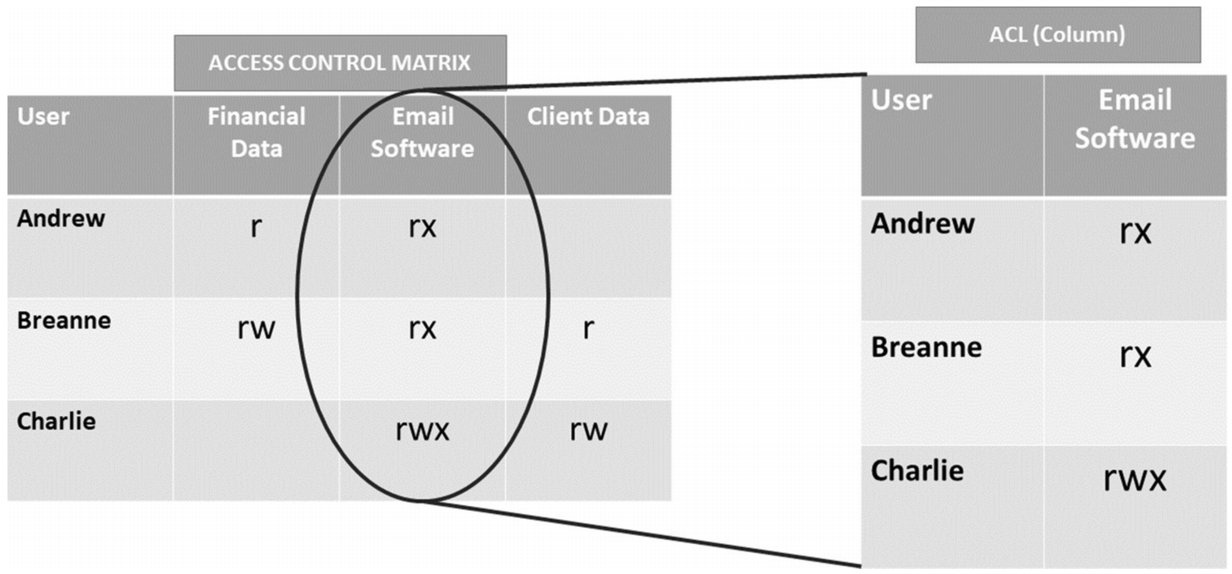
Access Control Lists

In all of these systems, there must be some way of assigning, storing, and changing the permissions granted to an authenticated user. The most common method is an *access control list* or ACL (pronounced “ackle”). An ACL is permission data for any specific resource, whether that is a file, a program, a system, or so forth. The ACL lists all of the users that have access to the resource and which permissions are granted.

A real-world analogy might be a bouncer at a club with a list of permitted patrons. In this example, the club is the resource, the list is the ACL, and the potential patrons are the users. The bouncer’s list can include various access levels. Some patrons might be a VIP and have access to areas off-limits to the standard patron. Other people might be performers coming to play live music and have special access to use the club’s sound system. These different levels of access are analogous to the permissions for each of the users in the ACL.

Computer system ACLs are often stored with the resource itself. Permissions could be kept in a big table, like a spreadsheet, where each row is a username and each column is a file. The permissions would be stored in the cell found at the intersection of a username row and file column, as shown in Figure 3-5. The problem, however, is that the table could get enormous. If there are N users and M files, there would have to be NxM entries in the table. This kind of format is called an access control matrix (ACM). On personal computers, that might be manageable, but it is too large for systems with multiple users. An ACL reduces the complexity by basically splitting out a column with each user’s permissions for just that resource and storing it with the resource itself.

The nature of ACL permissions depends on the model or model family they are designed to support. The ACLs of many personal computer systems are discretionary and not mandatory. In these systems, the creator of a resource is usually the initial *owner*, but ownership can be changed or assigned to other principals. The owner has all possible permissions but can also permit other users to have permissions to the resource. It is also common that an administrative user is not bound by the ACLs. Not only are they not restricted by them but they can also change them regardless of the owner’s preferences.



User	Financial Data	Email Software	Client Data
Andrew	r	rx	
Breanne	rw	rx	r
Charlie		rwx	rw

User	Email Software
Andrew	rx
Breanne	rx
Charlie	rwx

Figure 3-5 An access control matrix has all permissions for every user and for every resource. This matrix shows a Unix/Linux-style *rwx* permissions (read, write, execute). The ACL is a column of the matrix and has all of the users authorized for a given resource

On the other hand, in systems with mandatory access control support, permissions cannot be changed by the administrator at will. Instead, a policy installed into the system determines who can change what. As I described earlier in this chapter, DTE expresses its policy enforcement rules with what basically is another access control list for the domains. Internally, the operating system uses the DTE access controls to manage all of the runtime access controls. The difference between the two sets of controls is that the DTE controls are defined and installed before the system starts. Once the system is running, all other access controls and changes to access controls are governed by the DTE policy.

ACLs can also support groups. Discretionary systems, such as Unix and Linux, do not assign permissions per user (beyond the owner). Instead, permissions are divided into permissions for the owner, permissions for a defined group, and permissions for anyone on the system (sometimes called “world” permissions). The user can create groups with arbitrary membership, and this allows the user to assign (the same) permissions to any set of other users.

Linux will often display permissions like this:

```
rwxrwxrwx
```

Each letter refers to a permission. The “r” refers to read, “w” to write, and “x” to execute. The first three letters are the permissions granted to the owner, the second three are the permissions granted to the group, and the last three are the permissions granted to the world. Thus, the preceding example set of permissions grants all three permissions to all three sets of users. The following set of permissions, however, grants read and write to the owner, read to the group, and no permissions to the world:

```
rw-r-----
```

Most often, these permissions are associated with files and directories, which are Linux’s equivalent to folders. However, Linux keeps a special set of files that represent hardware resources, such as a camera. Permissions set on these special files can be used to restrict access to the hardware.

Standard Linux does not have support for the grouping in RBAC, and especially in ABAC, as this requires more complicated machinery to be built into the ACL. For ABAC, the ACL would also require a policy evaluation at runtime.

ACLs are very intuitive and easy to implement. This is part of the reason they are so widely used. On the other hand, they have some limitations. One important use that ACLs do not handle is *delegation*. Delegation is when an original user with a permission wishes to permit a proxy user to use that permission on the original user’s behalf. Delegation is used, for example, when someone has an assistant. The original user may want the assistant to help manage their calendar. But with an ACL, the only solution is to add the assistant to the list as well. This can be a problem because in a centrally managed system (such as a corporate system), the original user may not have permission to add the assistant to the ACL. Additionally, many traditional ACL systems do not track if one user’s permissions are really on behalf of another.

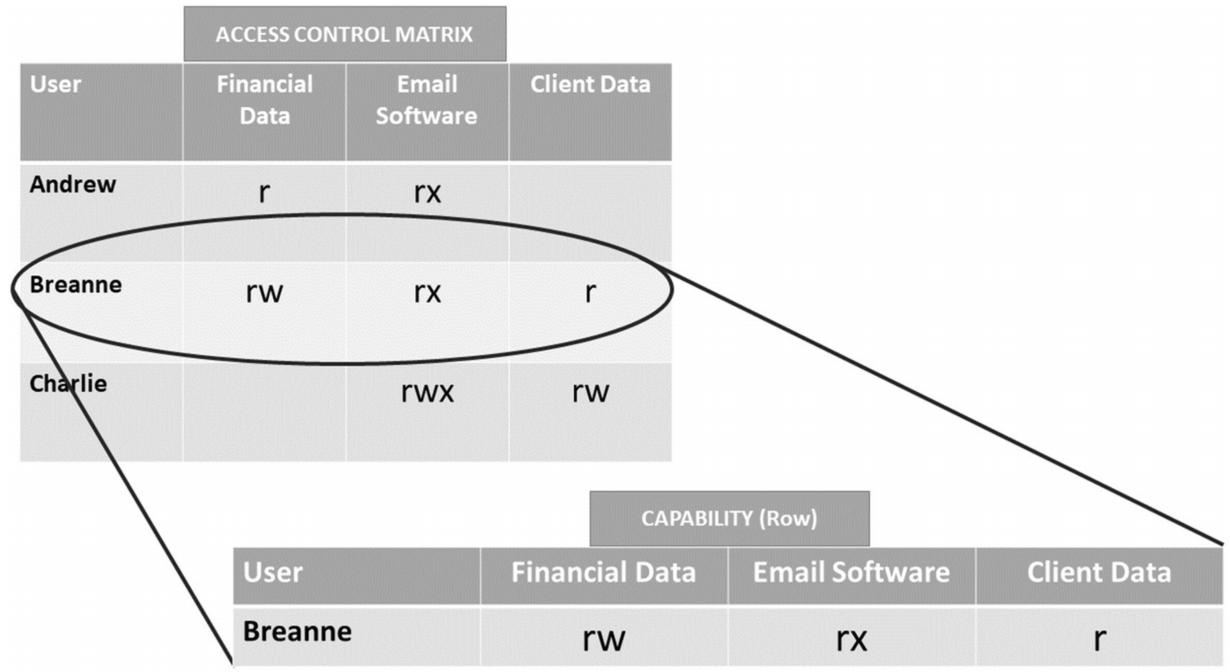
Some cloud-based email and calendar systems, such as Google’s, do permit delegating access using ACLs. But the delegation is limited by the nature of ACLs.

Capabilities

An alternative to ACLs is *capabilities*. A capability is some kind of data that permits the holder of the data to access an associated, protected resource. A capability is sometimes modeled as a row in the theoretical access control matrix of a system, whereas the ACL is like a column holding all of the users that are permitted to access a given resource. The column-like nature was previously illustrated in Figure 3-5. Figure 3-6 shows the alternative row-based concept of capabilities.

However it is modeled, it is conceptually permissions that the user gets to hold onto themselves. There are some potential dangers to a capability that must be managed. First, capabilities should not be forgeable. I will discuss cryptography more in Chapters 5 and 6, but cryptography (mathematical codes) can be used to create data that cannot be altered (undetectably) and for whom the author is provable. Using this kind of cryptography, the operating system can create a capability when the user logs in. The capability is protected so that it cannot be altered and provably came from the operating system access controls. An attacker cannot create such a capability themselves.

Another problem is that of copying. A user should not, without permission, be able to copy their legitimate capability to someone else for use. This is also solvable by having the correct user of the capability named in the capability itself. If an attacker steals the capability or tricks the user into releasing it, it will do them no good (unless they can also steal the user's identity). If the attacker tries to use the capability, the system will recognize it was not issued to them. The attacker cannot change the username identified in the capability because of the cryptography protections mentioned.



ACCESS CONTROL MATRIX

User	Financial Data	Email Software	Client Data
Andrew	r	rx	
Breanne	rw	rx	r
Charlie		rwx	rw

CAPABILITY (Row)

User	Financial Data	Email Software	Client Data
Breanne	rw	rx	r

Figure 3-6 This figure shows the same matrix as Figure 3-5. However, instead of an ACL (column), a row is broken out. This represents a *capability*

that can be issued to the user and indicates all resources to which they have access

One area where capabilities are sometimes seen as weaker than access controls is *revocation* of permissions. If a user should be stripped of their access to a resource (perhaps because they changed jobs or left the company), it is easy to change the data in an ACL. But if that user was issued a capability, how does one force them to voluntarily relinquish it? This problem can be solved with a *resource proxy*. It is not necessary to give a user direct access to the resource. Instead, a proxy resource is created and capabilities are issued for the proxy. When a user presents the capability, they access the proxy that subsequently forwards the requests and responses to the real resource. When access needs to be revoked, the proxy is deleted or disabled, a new proxy is created, and new capabilities are issued to the still-authorized users.

Another alternative is for capabilities to be temporary. A user can be given a capability that is only valid for a period of time and must be renewed thereafter. The time information can be written into the capability itself and must be valid to access the resource.

As explained earlier, *uncontrolled* copying is not allowed. However, capabilities can be easily delegated if delegation is allowed for the capability. A user can create a specialized copy of the capability that adds information about whom the capability is being delegated to and for how long, as shown in Figure 3-7. Although the cryptography prevents the original from being modified, the user can create their own additional data that can be attached to the original. When the delegated user arrives with the delegated capability, the resource can identify whether or not the capability can be delegated and whether it was delegated from an authorized party to the delegated user.

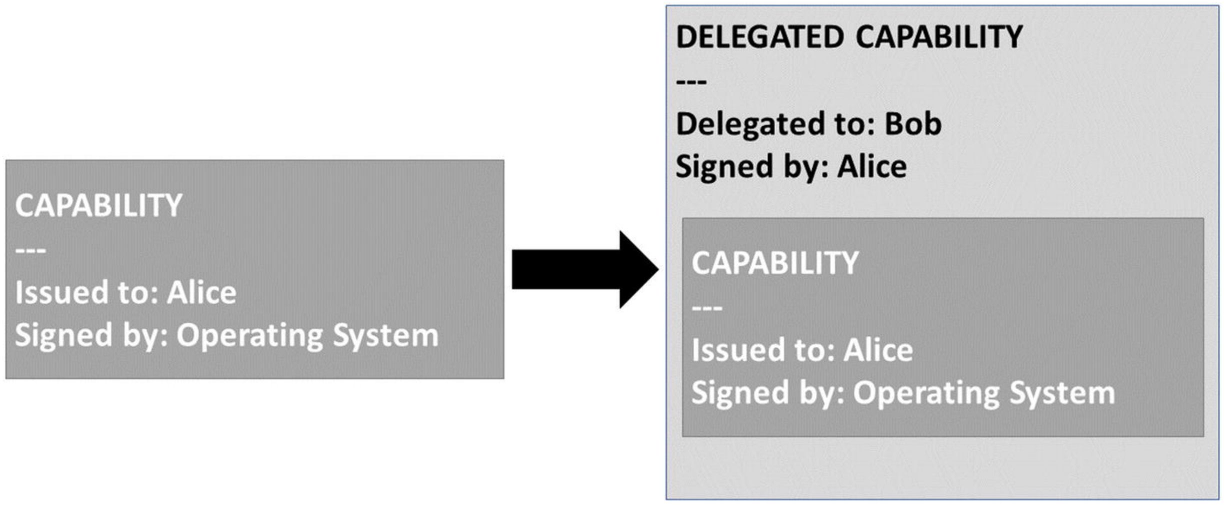


Figure 3-7 A capability can be stamped with the name of the authorized recipient. In this illustration, the original capability is issued to Alice, and only Alice may legitimately use it. Alice's capability is signed by the Operating System proving that the Operating System gave out this credential. Alice can delegate her capability by wrapping it in a new credential that she generates. She cannot change any of the original or it would break the signature. But she can add an outer layer that indicates she is delegating the original and have it signed in her name. When Bob presents the delegated capability, the resource will verify that Bob is presenting the capability, then verify that Bob was delegated the capability by Alice, and that Alice was issued the capability by the Operating System

As with ACLs, capabilities can be used to support DAC or MAC systems.

Access Control Implementation Issues

Not all access control technologies are focused on permissions assigned to users and the programs that run in their names. Especially in mandatory access control systems, it is sometimes important to protect applications from each other independent of the user that controls them.

The Android operating system, using the SELinux base, is capable of isolating the applications (apps) that run on the device. Apps, for the most part, cannot see the files of any other apps. This prevents an app, especially a malicious app, from interfering with, stealing data from, or corrupting any other app. The only mechanism for interaction is through a few trusted system services and mediated requests. This kind of isolation is even stronger than using access control lists or capabilities; it simply disallows everything except carefully monitored request handling.

Modern Windows operating systems also provide a complex set of access controls that include elements of ACLs and capabilities, as well as discretionary and mandatory access controls. A good overview of these features can be found in Anderson's *Security Engineering* book [40, Chapter 9]. But the key point is that modern systems combine many of the technologies discussed in this chapter into their operations.

Complete Mediation and Reference Monitors

Regardless of the conceptual approach (e.g., ACL vs. capabilities) or the implementation, access controls generally need to follow the principle of *complete mediation* and use an effective *reference monitor*.

The principle of complete mediation states that "all accesses to objects be checked to ensure they are allowed" [60, Chapter 14]. The reason the word mediation is used is because there is typically some kind of manager component that is responsible for enforcing the access controls. This manager could be, for example, the operating system on a host. But the point is that *all* access should have to go through the manager and that *every* access is checked.

Whatever form the manager takes, from the security perspective, it is a reference monitor. Or, stated affirmatively, the reference monitor "is an access control concept... that mediates all accesses to objects by subjects" [60, Chapter 20]. In other words, the reference monitor provides the complete mediation. Obviously, much of the security of the system depends on the effectiveness of the reference monitor.

Access Control and Psychology

Speaking of Android, smartphones are a wonderful example of how access controls can be great in theory but completely fail when the psychology is poorly aligned. In addition to protecting apps from each other, Android, like most smartphones, also attempts to protect you and your phone from your apps. For privacy reasons, apps do not have permissions to phone resources such as the camera, microphone, contact list, and calling functions without explicit permissions from the user.

The problem with these kinds of access controls is that they stand between the users and a *reward*. Imagine a user that has a smartphone. The user hears about a new app (perhaps a popular game) or realizes they need some kind of utility. Either way, they head to the app store and search for the app. Once they find it, perhaps they even read some reviews. The user decides they want the app. They click the download button. Success! The app is on their phone! They try to open the app.

Everything stops. The app will not open. Apparently, the app will not run unless the user gives the app permissions to use something on the phone. Android asks the user if they should allow it.

What do you think the user will do? Will the user carefully consider the risks of giving the app access? Will they rationally evaluate whether or not an app of this type needs those permissions? Or do you think the user will install the app without thinking much about it?

If you answered the latter, you are probably correct for the majority of users. There are many reasons for this. The first and most important is that the user was looking forward to their reward: using the app. They had thought about it, sought for it, and obtained it. Then, at the last minute, when they were about to enjoy the fruits of their (admittedly minimal) efforts, they found themselves blocked by a choice. The urge for the reward is a significant factor in the user's behavior.

The other issue is a lack of understanding of the security risks. So what if the app can access the user's contact list? What is the danger really? The warning messages do not instruct the user in the risks and assume that the user already understands them. It assumes rationality when there is no evidence of rationality or even the information necessary for rationality (rational decisions require sufficient information). Users are also somewhat conditioned to believe that app stores largely hand out secure apps. The companies behind the app stores are always touting how secure they are as a company. The users get a sense of security when they get something from the official store, and that also overrides their thoughts of risk management.

Asking users to make these kinds of decisions is always fraught with peril. In many ways, it punts on the very difficult problem of deciding what is and is not secure to the least qualified. It absolves the owners of the app store of responsibility. The incentives at play are terrible from a security perspective. The companies hosting the app stores take a cut of every app sold (or a percentage of the advertising from free apps). Yet, they bear little to no responsibility or liability for the bad things the apps do.

Side Channels

Before wrapping up this chapter, I will discuss a little bit about *authorization leaks*. An authorization leak occurs any time correctly enforcing a model's properties still results in an operation that though technically legal is semantically equivalent to an unauthorized access. The most common authorization leak is an *information leak* wherein an attacker is able to obtain unauthorized information through legitimate channels.

One method for an information leak is to use a *side channel*. A side channel is any system output that is either unintentional (it was not intended to be an output) or is an intentional output but unintentionally includes extra information. This has been demonstrated in BLP. For example, a user is supposed to be able to write, but not read, to a folder of a higher classification than their clearance. Not reading includes not knowing the names of files within the folder. But if the user tries to insert or create a file of a given name into the folder and a file of that name already exists in the folder, the system may give them an error. This leaks to the user that a file of a given name already exists. The side channel in this example is the error reporting of the folder. All of the BLP requirements were followed but information leaked from high to low.

Side channels are notoriously difficult to identify. Side channels have included, for example, the *heat* emanating from a computer, the blinking lights of a network router, and even the time it takes to compute information. *Passive side channels* are side channels that only require observation. *Active side channels* require attacker input, usually in the form of a disruption that causes the unintentional release of information.

Summary

Security engineering depends on the use of good policies that guide and control the deployment of mechanisms to enforce them. Nevertheless, as illustrated in this chapter, models often have rough edges and use cases for which they have weaknesses. They can also have deployment challenges, especially for exception handling.

This is one reason why developing a new policy from scratch should be avoided when possible. The policies in this section, from BLP to ABAC, have been around for years. They have been implemented, tested, evaluated, refined, and revised. Their implementations have been analyzed and experience gained from real deployments. When evaluating a vendor’s technology, they can be good starting points for understanding what problems it will and will not solve. Even if the product implements an entirely new model, it can be beneficial to compare it to the others.

This chapter also discussed some of the basics of low-level access controls. Specifically, ACLs and capabilities are two alternatives for granting a principal or subject access to a resource. Other mechanisms like isolation are helpful.

Authorization continues to be a difficult problem. Developing or choosing the correct model is hard, implementing (and verifying the implementation of) a model is hard, and keeping the authorization data in sync with reality is hard. Getting the psychology and incentives of an authorization system is also difficult as illustrated by the deceptively easy and commonly used approach of letting users choose to grant permissions to apps. This method pushes the most difficult decisions to the least prepared.

Even if somehow we got all of these hard things correct, attackers might still be able to extract useful information from a previously unknown side channel.

It is important to keep all of these limitations in mind when evaluating an authorization technology from a vendor. Vendor marketing will always focus, at best, on the product’s strongest features. More often, it will make irrelevant or unhelpful statements or promises. Knowing the real problems inherent to all authorization technology will help you ask the right questions and make better decisions about how well the technology will help your organization.

Further Reading

If you feel like reading a long mathematical proof, you can always read the original Bell-LaPadula paper cited in this chapter. The authors do not write their model in much of a narrative format. Instead, they attempt to establish a proof about what a secure system is and that the BLP system is so. Bell wrote a paper in 2005 looking back over the development of BLP that describes a bit more about the process and why they did what they did [51].

As always, Ross Anderson's *Security Engineering* is a great read. Chapter 6 in the third edition is all about access controls and goes into the mechanisms used by operating systems, software, databases, and other components. He also covers some technical attacks, such as buffer overflow attacks, that I will touch on in Chapter 7. But his treatment in the context of access control is good for understanding how the attackers get around these defenses [40, Chapter 6]. Matt Bishop's book *Computer Security: Art and Science* (second edition) is excellent, but I only recommend it for readers with a technical background. If you do dive in, Chapter 2 covers an overview of the conceptual access control matrix, and Chapter 16 dives into ACLs, capabilities, and a few concepts not covered here [60].

Most readers will find topics around RBAC and ABAC to be the most immediately practical. Every semester, I have my students do a Google search for "problems with RBAC" as a way of seeing a range of responses from the field. This is an informal method, of course, but it is very real. For more details on RBAC, I recommend the work of Ravi Sandhu. In particular, [230, Section 1] gives an approachable overview of the history of RBAC and some of its limitations. Various RBAC implementations have been proposed over the years, and various attempts have been made at standardizing them [104, 229]. For greater technical detail, especially around design and implementation details, you can refer to [49] and [134]. For another intense (and more recent) study, NIST researchers published a book in 2017 entitled *Attribute Based Access Control* [138].

I especially recommend reading Danette McGilvray's book about improving data quality in an organization. I will be citing from this book in other chapters, but authorization is a good place to emphasize it. I cannot stress this enough: you cannot secure your organization if your information quality is poor. If you do not know accurate and timely information, you cannot correctly handle roles, permissions, or any other authorization task correctly. Nor should you assume that if you do not know it the attackers will not as well. You have to understand that you and the attacker have inverse relationships in terms of normal vs. abnormal conditions. You will almost always know more than the attacker about your systems' *normal* operations. The attacker will almost always know more than you about your systems' *abnormal* operations. Increasing your data quality reduces the abnormal operations in your organization, giving the attacker fewer places to work and giving you more places where you are better informed. Read the book and make your security team read the book [176].

References

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

47. Badger, L., D. Sterne, D. Sherman, K. Walker, and S. Haghighat. 1995.

Practical domain and type enforcement for unix. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, 66–77.

49. Barkley, J., K. Beznosov, and J. Uppal. 1999. Supporting relationships in access control using role based access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, New York, 55–65. Association for Computing Machinery.

51. Bell, D. 2005. Looking back at the Bell-La Padula model. In *21st Annual Computer Security Applications Conference (ACSAC'05)*

52. Bell, D.E., and L.J. LaPadula. 1973. Secure computer systems: Mathematical foundations. *Draft MTR, The MITRE Corporation*, 2.

54. Belokosztolszki, A. 2004. Role-based access control policy administration. Technical Report 586, University of Cambridge. UCAM-CL-TR-586.

59. Biba, K.J. 1977. Integrity considerations for secure computer systems. Technical report, MITRE Corporation.

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

65. Brossard, D., G. Gebel, and M. Berg. 2017. A systematic approach to implementing ABAC. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control (ABAC'17)*, New York, 53–59. Association for Computing Machinery.

74. Chung, D. Ferraiolo, D. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. 2019. Guide to attribute based access control (ABAC) definition and considerations. Special Publication (NIST SP) 800-162, National Institute of Standards and Technology, Gaithersburg.

78. Columbus, L. 2019. 74% of data breaches start with privileged credential abuse.

104. Ferraiolo, D.F., J.F. Barkley, and D.R. Kuhn. 1999. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)* 2(1): 34–64.
[Crossref]

105. Ferraiolo, D.F., and D.R. Kuhn. 1992. Role-based access controls. In *15th National Computer Security Conference*, 554–563. National Institute of Standards and Technology.

121. Group, J.T.F.T.I.I.W. 2020. Security and privacy controls for federal information systems and organizations. Special Publication (NIST SP) 800-53r5, National Institute of Standards and Technology, Gaithersburg.

124. Guttman, B., and E.A. Roback. 2017. An introduction to computer security: The NIST handbook. Special Publication (NIST SP) 800-12r1, National Institute of Standards and Technology, Gaithersburg.

134. Hitchens, M., and V. Varadharajan. 2000. Design and specification of role based access control policies. *IEE Proceedings-Software* 147(4): 117–129. [Crossref]

138. Hu, C.T., D. Ferraiolo, R. Chandramouli, and D. Kuhn. 2017. *Attribute Based Access Control*. Norwood: Artech House.

159. Landwehr, C.E. 1981. Formal models for computer security. *ACM Computing Surveys* 13(3): 247–278. [Crossref]

161. Leveson, N.G., and C.S. Turner. 1993. An investigation of the therac-25 accidents. *Computer* 26(7): 18–41. [Crossref]

176. McGilvray, D. 2021. *Executing Data Quality Projects: Ten Steps to Quality Data and Trusted InformationTM* . Elsevier Inc.

228. Sandhu, R. 1996. Rationale for the rbac96 family of access control models. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, New York, 9–17. Association for Computing Machinery.

229. Sandhu, R., D. Ferraiolo, R. Kuhn, et al. 2000. The NIST model for role-based access control: Towards a unified standard. In *ACM Workshop on Role-Based Access Control*, vol. 10.

230. Sandhu, R.S. 1998. Role-based access control. In *Advances in Computers*, vol. 46, 237–286. Elsevier.

232. Sarna-Starosta, B., and S.D. Stoller. 2004. Policy analysis for security-enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, 1–12. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

244. Sonnemaker, T. 2021. Verkada allowed at least 100 employees, including interns and sales staff, to access customers' camera feeds. *Business Insider*.

266. Tunggal, A.T. 2022. What is role-based access control (RBAC)? Examples, benefits, and more.

280. Whitney, L. 2021. Microsoft power apps misconfiguration exposes data from 38 million records.

Footnotes

1 Note: This acronym will mean different things in different chapters, so pay close attention to context.

2 I am adapting this statement from the Therac-25 incident, a case of software *accidental* failure. The Therac-25 was a cancer-treating, radiation therapy machine that, because of software bugs, inadvertently killed a number of people and injured others. As reports of accidents came into the company, they would make a fix to something they thought was the problem and in at least one case then claim it “produced a five order of magnitude increase in safety” [161]. I have heard similar expressions from vendors selling or senior management buying various security products. In almost every case, statements such as these are meaningless marketing at best and extreme incompetence at worst.

3 Unfortunately, these terms are sometimes used a little differently across the security world. Some sources refer to *users* only as the people themselves, *principals* only as the account of the user, and *subjects* only as the programs launched by users. The original BLP formulation explicitly stated that subjects are processes (running programs) but also said that they were “surrogates” for users [52]. NIST uses subject to refer generally to all three, and I will do the same for convenience [121].

4 Anderson only discusses the NRU and NWD properties. But he describes the systems that enforced these two properties as being mandatory access controls.

5 Sometimes, TCB is used to refer specifically to just a given set of hardware or software that provide certain trusted operations within a computer system. For example, TCB systems are used to enforce phones booting from only software authorized by the manufacturer. When talking about a conceptual policy, as I am in this chapter, TCB needs to include all the components that must be trusted for the policy, including people.

6 One exception is technology mathematically proven to be correct. Some hardware and some software can be proven to have no bugs of certain types. There are many limitations, however, including that proofs become impossible

beyond certain sizes of systems and corresponding complexity. There is interest, despite these limitations, in using formally proven systems precisely for TCB components. Because proofs generally require relatively small systems, this is yet another reason for having small TCBs.

7 In Biba’s original formulation, Biba identified additional requirements not considered here. However, these two properties are the key ideas commonly identified for Biba’s model. Also, the names “Simple Integrity Property” and “* Integrity Property” were not used in Biba’s original paper, and I do not know who introduced them or when. But they were in use as early as 1981, just a few years after Biba’s initial publication [159].

8 Anderson describes “type enforcement,” which preceded DTE, as a model (singular) and implies that DTE is also a model [40, Chapter 9]. However, both the original paper and other sources refer to DTE policies (plural). Given that DTE really does not express any security properties by itself, I have chosen to describe it as a model framework. The original paper describes it as a mechanism, but the authors were also presenting an actual implementation of the conceptual system. The choice of the term *model framework* is my own.

9 There are two permissions related to running a program. One is whether or not one domain is allowed to start a process in some other domain (including its own domain), and the second is whether or not it is permitted to start programs of a particular type.

10 There are a wide range of definitions for RBAC. There are a wide range of research papers that discuss variations of RBAC creating what is called the “RBAC Family” [228]. RBAC has been described as an alternative to both MAC and DAC [49] or even as “solving the described problems” of MAC [54]. Other authors, however, state that RBAC is a form of MAC [48,132]. Notably, the same author, John Barkley, is the author of one citation that says it is an alternative and one citation that says it is a variant. Another author, not willing to call RBAC a MAC, described it as “nondiscretionary” [104]. Of course, within the broader IT world, it is often used without a formal definition at all and simply refers to any policy that associated roles with permissions. For the purposes of my discussion in this section, I am using this less formal definition while noting that the pros and cons I discuss are more or less applicable to the more formal models.

11 There are many variants and definitions of ABAC as well. However, the NIST definition is sufficient for the purposes of this chapter.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_4

4. Cryptography Foundations

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide

This chapter introduces the concept of *cryptography*, or mathematical codes used to protect data. It can be a tough concept. To help make this concept more comprehensible, this chapter covers some of the goals and requirements for cryptography. It then uses some historical examples to illustrate a subset of these principles. People have been using secret codes since before computers. These examples can be easier to understand but can also effectively introduce some concepts like *key size*, *block size*, *brute force*, *block ciphers*, *stream ciphers*, and *cryptanalysis*.

Key Concepts

1. Cryptography is the mathematics of data protection.
2. A *key* is data that enables a cryptographic operation (e.g., encryption); without the key, the operation cannot be performed.
3. Cryptographic systems are designed to be secure so long as secret keys are kept secret; the attacker may know anything else about the system.

4. *Symmetric* cryptography is so called because inverse operations use the same key, for example, encrypting and decrypting with the same key.
5. *Block ciphers* are symmetric cryptographic algorithms that work on chunks of data called "blocks."
6. *Stream ciphers* are symmetric cryptographic algorithms that work on data one bit at a time.
7. *Asymmetric* cryptography uses *key pairs* where a *private* key performs one operation (e.g., encrypting) and a *public* key performs the inverse (e.g., decrypting).
8. *Encryption* can be used to provide data confidentiality by making the data unreadable (without the appropriate key to decrypt).

Common Pitfalls and Misunderstandings

1. The security of a cryptographic system should only require that the key be secret.

Useful Vocabulary

- **Confidentiality**: Data can only be read by authorized parties.
- **Data Integrity**: Data can only be modified by authorized parties.
- **Entity Authentication**: The publisher (e.g., transmitter or creator) of the data can be confirmed.
- **Hashing**: A technique that takes any data (e.g., a document) as an input and spits out a small piece of data that serves as a fingerprint of the original.

The previous two chapters on *authentication* and *authorization* are foundational to computer security technologies. Generally speaking, security requires identifying principals and determining what they are allowed to do. But these controls would be worthless without mechanisms for enforcement. If someone is not allowed to read a file, what prevents them from being able to do so? This chapter and the next collectively introduce *cryptography*, a technology that enables certain kinds of protections for data, even if the data falls into the wrong hands.

Cryptography. You might have heard of this. You might not. But cryptography is, in my opinion, one of the most amazing developments of humankind *ever*. Such assessments are certainly objective; but for me, what we are about to learn in this chapter surpasses the marvel of world wonders such as the Pyramids, Stonehenge, or the Hanging Gardens. Some of the things cryptography can do literally seem like magic.

It is not magic, of course. It is *mathematics*. Given that this book is meant for an audience that is not necessarily trained with a technical background, you might be one of the many people that dislikes mathematics. You may have had an intense, negative emotional reaction when you saw the word. If you are one of these people, please take a big deep breath. Everything will be fine.

So what is it? At a basic level, cryptography is mathematics used to protect information such as computer data. You have already learned a little about authentication and authorization. Cryptography is one of the means by which information can be protected so that only authenticated and authorized parties can access it. It can also be used to help in the authentication and authorization processes themselves. Cryptography is not the only way of protecting information, but it is one of the most widely used. Much of the modern Internet infrastructure depends upon cryptography to function securely.

But the math behind cryptography can be counterintuitive and very opaque. Fortunately, there are historical examples that predate computers that are a little easier to study and analyze. Despite their differences from modern cryptography, they can do a good job of teaching some core concepts.

In this chapter, I will start with some historical examples that introduce the intuition behind some aspects of cryptography. After that, I will provide some background on what modern cryptography aims to achieve and some prerequisite components. This will prepare you for the next chapter that gets into modern cryptography in more detail.

Introducing Cryptography Through Historical Examples

In this section, I will be using historical examples to teach you about *encryption*. As I emphasize later, there is more to cryptography than just encryption. However, if you can understand historical encryption, it will help orient you for everything else.

Encryption is the process of encoding a message such that the message is (or is supposed to be) unreadable. Decryption is the process of decoding the message into a readable or understandable form. Some piece of data, known as the *key*, is used to encrypt. Without an appropriate key, it is (or is supposed to be) impossible to decrypt the encrypted message.

In modern cryptography, *symmetric cryptography* uses the same key to encrypt and decrypt, while *asymmetric cryptography* uses different keys to perform each function.

All of the historical examples that follow are symmetric. Because the same key is used for encryption and decryption, these historical examples require that both the sender and receiver of a message must share the same key. The security objective is that, assuming only authorized parties have the key, unauthorized parties will not be able to decode the message and read/understand it.

The process by which data is encoded or decoded is called an *algorithm*. A computer algorithm is a specific series of steps that, when followed, produce an expected output. So, an encryption algorithm is the series of steps that transform the original message into an encrypted message. The decryption algorithm is the series of steps that transform the encrypted message back into the original message. Encryption algorithms generally require the original message and the symmetric key as inputs, and decryption algorithms generally require the encrypted message and the symmetric key as inputs.

In cryptographic terminology, the original message is often called the *plaintext*, and the encrypted data is called the *ciphertext*. The decryption function should transform the ciphertext back to plaintext, but sometimes, for clarity, it is called the *recovered plaintext*. The encryption and corresponding decryption algorithm are, together, a *cipher*.

The Caesar Cipher

The use of ciphers to protect information from being read (confidentiality) goes back a very long way in history. Julius Caesar used a very simple cipher that has become known as the Caesar Cipher. In the Caesar Cipher, each letter of the alphabet is mapped to another letter of the alphabet using a simple *shift* mapping as shown in the following:

A	B	C	D	E	F	G	H	I	J	K	L	M
D	E	F	G	H	I	J	K	L	M	N	O	P
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Q	R	S	T	U	V	W	X	Y	Z	A	B	C

In this cipher, A maps to D, B maps to E, and so forth.[1] To encipher or encrypt a message, each letter of the plaintext was replaced with the corresponding letter in the table:

```
HELLO WORLD
KHOOR ZRUOG
```

In this listing, “HELLO WORLD” is the plaintext and “KHOOR ZRUOG” is the ciphertext. To decrypt, the process is reversed. Each letter of the ciphertext is found in the bottom row and substituted with the corresponding letter in the upper row. So far, so good.

But the eagle-eyed reader might already be wondering, “where is the *key* in all of this?” After all, I told you that encryption and decryption algorithms required a key. How come we did not use a key here?

We did use a key, I just did not call it that. To illustrate, consider that Augustus Caesar changed his letter substitution mapping as follows:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
C	D	E	F	G	H	I	J	K	L	M	N	O
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
P	Q	R	S	T	U	V	W	X	Y	Z	A	B

What, exactly, did he do? Did the *algorithm* change? No! The algorithm is the same. What changed is the mapping of letters such that A maps to C instead of to D. All other changes are driven by this change. We could say that Augustus changed the key of this algorithm from “D” to “C” [40, Chapter 5].

This very simple cipher introduces us to a very important characteristic of a given cipher: *the key space*. The key space refers to the number of keys that can be used in the given algorithm. In the Caesar Cipher, using the modern English alphabet, there are either 25 or 26 keys depending on whether or not you want to consider A mapped to A to be a legitimate key.

Having only 25 usable keys is a very big problem for a cipher. How long would it take an adversary to try all possible keys? Even without a computer, it would not take an adversary very long at all to try all possible 25 keys. The adversary could even precompute the tables for all 25 keys and try all possible combinations in just a few minutes.

This leads to the first requirement for an effective cipher. The key space must be sufficiently large to make it impractical to try all possible combinations of keys. In Chapter 2, you learned about *brute force* for breaking password hashes. In that case, an adversary can try all possible passwords to try and find a matching hash. The same approach can be used by attackers to break a cipher. They can use brute force to try all possible keys to see if it decrypts the information into the plaintext. Without a sufficiently large key space, brute force will destroy the confidentiality of the data.

Security experts are often asked if it is a viable strategy to try and keep the *algorithm secret*. After all, if the adversary does not know the algorithm, how can they even try a key in the first place? From experience, it is well known that algorithms can be reverse engineered and far more easily than intuition might lead you to believe. This observation led to what is now known as Kerckhoffs’s principle:

> The system must not require secrecy and can be stolen by the enemy without causing trouble. [36]

The alternative, the idea that the system can be secure through secrecy, is often called *security through obscurity*

[36] or *security by obscurity* [39]. Security through obscurity is generally seen as harmful in the security community, and especially in cryptographic contexts [66, Chapter 1].[2] Kerckhoff's principle is specifically about cryptography systems and keys, but the concept is so broadly applicable in security that it has its own name: the principle of open design. This principle states that the security of a mechanism should not depend on the secrecy of its design or implementation [60].

Story Time: Navajo Code Talkers—Security by Obscurity?

When teaching my classes, I am occasionally asked by a student whether or not Navajo Code Talkers are a good example of security by obscurity being successful. If you are unfamiliar with the Navajo Code Talkers, they were a group of Native Americans of the Navajo tribe that served as special signalmen in the US Marine Corps in World War II. Because Navajo was an oral language (there was no written version at the time), a proposal was made for using native Navajo speakers as code operators for radio transmissions.

Notably, the Code Talkers did not just speak normally in their native language. They created a code where different terms were assigned to certain objects. For example, dive bombers were identified as "ginitsoh" meaning "sparrow hawk" because "the sparrow hawk is like [the dive bomber]—it charges downward at a very fast pace." They also created words to represent each letter of the alphabet in case there was a need to spell things out [30].

In terms of the strength of this code, there are a few things to keep in mind. First, this was an oral code only. It could not be used for anything written. The Enigma machine, on the other hand, could be used for teletext transmissions. But this also meant that the Allies could collect a larger number of samples with few transcription errors for cryptanalysis.

Second, we have no detailed information about how the Japanese approached cracking the Navajo military code [30]. Without knowing what progress they did and did not make, and whether or not it was a priority, it is difficult to know how resilient the code was. For example, they were cracking *other* US codes as well as the codes of the other major nations. How many resources they were allocating to cracking this radio code is unclear.

Third, there was a code underneath the language. When a message was sent in Navajo, the receiver, although a native speaker of Navajo, did not always know what the message meant until the English equivalent of the words were looked up in a code book. How much security was there in the Navajo language and how much security was there in the code book?

In terms of what we know about the Navajo code, it is certainly breakable. The fact that *it was never broken* does not mean *unbreakable*. For example, the fact that each letter of the English language was assigned one or more words means that it is relatively trivial to crack the code if the analyst has a single sample of the plaintext and ciphertext regardless of whether or not the analyst speaks Navajo.

At least one linguistic scholar has stated that the security of the code talking can be traced to difficulty that the Japanese radio operators had in producing consistent transcriptions of what came over the radio [139]. A system that depends on the enemy *hearing* phonetic sequences poorly will not survive a skilled and motivated cryptanalysis team.

In summary, while Navajo Code Talkers were undoubtedly valuable to the US military action in World War II, the approach of an unknown language is not a model for modern cryptographic systems.

So, we must *assume* that the attacker knows that a message was enciphered using the Caesar Cipher. If the attacker knows that, how difficult will it be for them to break the message? Given that the Caesar Cipher has only 25 keys, we have to throw it out. It is too easy to break with brute force on the small key space.

Generalized Monoalphabetic Substitution

The Caesar Cipher is known as a *monoalphabetic substitution cipher*. That means that each letter of plaintext is substituted for exactly one letter of ciphertext. But because of the way the shift cipher is configured, there are very few keys and, consequently, very few configurations of character substitutions. But does that have to be the case? Could we have a monoalphabetic substitution cipher that has a sufficiently large key space?

What about a completely random mapping of letters instead of doing Caesar's shift? For example, consider the following:

A	B	C	D	E	F	G	H	I	J	K	L	M
X	N	B	Y	A	M	L	S	V	P	R	K	W
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	C	G	I	U	D	T	F	O	H	J	Q	E

This mapping creates what is sometimes called a *permutation* cipher. This is because the letters are *permuted* or reordered. Each permutation is a key in this cipher.[3] How many possible keys are there?

To figure this out, suppose we started with the letter A on the top row. How many possible mappings are there to other letters? Assuming we will not permit A to map to A, there are 25 mappings for the letter A. But what about for the next letter, B? How many possible choices are there for this substitution? The answer is *24* because we already used one for A, and it cannot be used for B as well. Moving down the line, C will have 23 possible substitutions, D will have 22, and so on. This means that there will be $25 \times 24 \times \ldots \times 2 \times 1$ possible permutations. If you remember this kind of thing from your elementary or junior high school days, this is called *factorial*. How big is 25 factorial?

15,511,210,043,330,985,984,000,000

That is a pretty big key space. It is not as big as the key spaces used today, but it is bigger than key spaces used up until about the year 2000. So this must be a much stronger cipher than the Caesar Cipher?

In actuality, no. It is harder to break using brute force on the key space, but it is trivial to break using *cryptanalysis*. Cryptanalysis is the set of techniques used to break codes without keys, usually by looking for patterns or other artifacts that reveal information within the encoded message.

Breaking a monoalphabetic substitution cipher is easy because so much of the original message patterns are visible in the ciphertext.

Look at the following enciphered message:

```
SAKKC HCUKY
```

This message is encoded using the same permutation listed previously, so you can easily decode it. But even if you did not know the permutation (i.e., the key), you could probably start to make some very good guesses.

First of all, there is a “KK” in the ciphertext. Because this is a monoalphabetic substitution cipher, you know that the K is mapping to the same letter both times. That means that the plaintext word had two of the same letter. If you know that the original message is in English, you can quickly search a dictionary for words that have two letters in the middle. You can also do tricks like substituting in common letters. E, for example, is the most common letter in the English language. Other features and characteristics of the language bleed through from the plaintext to the ciphertext. These kinds of artifacts make breaking such messages pretty easy to do.

Increasing the Block Size: The Playfair Cipher

The problem of the monoalphabetic cipher is that its *block size* is too small. By block size, I mean the number of symbols that are encrypted at a time. The block size of a monoalphabetic cipher is one character. And as already discussed, this is far too easy to cryptanalyze.

For this reason, the makers of secret codes advanced to using *digraphs* or encryption using *two* letters at a time. In other words, the block size was increased from one to two. A good example of this is the *Playfair* cipher.

To set up a Playfair cipher, you need a word-based key. For our example, we will use SECURITY. Next, create a five-by-five grid; put the letters of the key into the grid from left to right and top to bottom, as shown in the following:

S	E	C	U	R
I	T	Y		

Note that no letters in this grid can be repeated. SECURITY only uses each letter once. If a code word is chosen with duplicate letters, the duplicate letters are dropped. For example, the code word HELLO would only put in the letters HELO.

Once the code word is in place, add the other letters of the alphabet in order. Skip any letters already in the table from the code word. Also, because there are 26 letters but only 25 squares, one letter of the alphabet has to be dropped. By convention, the J letter is dropped. Our example grid looks like this:

S	E	C	U	R
I	T	Y	A	B
D	F	G	H	K
L	M	N	O	P
Q	V	W	X	Z

Now that our table is set up, we can begin to encipher messages. The algorithm is as follows:

1. If the length of the message to be enciphered is not divisible by two, add an X to the end.
2. Break the message up into pairs of letters.
3. If a pair of letters is in the same row of the table, replace with the next two letters to the right (wrap if necessary).
4. If a pair of letters is in the same column of the table, replace with the next two letters down (wrap if necessary).
5. If a pair of letters is not in the same row or column, the pair forms a box; replace the letters with the two letters of the opposite corners of the box, with each letter replaced with the corner in the same row.

Following this algorithm, and using our table, encrypt the message “ATTACK AT TEN O’CLOCK.” To prevent an enemy from easily guessing whole words, first remove the spaces and punctuation. Our message to encrypt is “ATTACKATTENOCLOCK”.

Following the algorithm, the first step is to see if we need an extra letter. Our message is 17 long, which is not divisible by two. So we add a Z to the end.

Next, the message is broken up into letter pairs:

```
ATTACKATTENOCLOCKX
AT TA CK AT TE NO CL OC KX
```

Following our rules for enciphering each pair of letters:

1. AT - BY (same row)
2. TA - YB (same row)
3. CK - RG (box corners)
4. AT - BY (same row)
5. TE - FT (same column)
6. NO - OP (same row)
7. CL - SN (box corners)
8. OC - NU (box corners)
9. KX - HZ (box corners)

Thus, the ciphertext of our message reads:

```
BYYBRGBYFTOPSNNUHZ
```

To decrypt this message, the process is followed in reverse.

There are several key observations from this cipher. First, what is the key space? The answer to this question is different in theory than in practice. In theory, the key space is very large. It is still 25 factorial! Although passwords are used for convenience, what is really being done here is a *permutation* of 25 letters of the alphabet! *Any* permutation could be used so long as there was a relatively easy way of transmitting the permutation to recipients. No code word is actually needed.

In practice, however, code words are used, and this reduces the key space to probably around 2000 words or so. In any event, it would be trivial for a modern computer to try several thousand code words very quickly. But the use of a code word to fill in a complete permutation is very similar to a concept used today called *key expansion*. Sometimes, a key needs to be a particular length. There are various ways of taking a key and expanding it to an almost arbitrary size. In fact, one of the most common encryption algorithms used today, AES (Advanced Encryption Standard), uses key expansion in its internal algorithms.

Another interesting feature of Playfair is the use of the letter Z if the original message is not a multiple of two in length. In modern terminology, we would describe this extra letter as *padding*. Modern cryptography often has to work on chunks of data of a given size. If the data cannot be divided up into chunks of exactly that size, it often has to be padded.

Story Time: Insulted by Padding
During October 1944, the largest naval battle of World War II was fought in the Leyte Gulf of the Philippines. The battle was actually spread out over four separate engagements of naval forces. One of these engagements, the Battle of Samar, involved a weak and unprepared American naval force under attack from a superior Japanese force. Task Force 34, the larger and more

powerful American force, had been successfully drawn away by a decoy fleet that used carriers (high value targets) as bait. Task Force 34 was under the command of the extremely aggressive Admiral William “Bull” Halsey Jr.

When the weaker fleet came under attack at Samar, coded messages were transmitted by radio asking for the position of Halsey’s ships. The message sent out by Halsey’s senior officer, Admiral Nimitz, was, “Where Is Task Force 34?” The message was encrypted using the standard codes of the time. However, US operators were concerned about the beginning and ending of messages because they often used the same words (such as “Dear” for a beginning word or “Yours” near the end). To keep Japanese code breakers from deciphering the messages, padding of nonsense words were to be used at the beginning and the ending before encipherment. Obviously, the padding was also to be removed after decryption.

However, when Admiral Nimitz sent his message, the radio operator made some errors. First, he thought there was an extra emphasis on “Where” and added in a “Repeat” into the message, making it sound more emphatic. Second, and worse, instead of using nonsense words for the padding, as was the protocol, he used the words, “The World Wonders.” When the message was received on Halsey’s ship, the operator did not recognize these words as padding and left them in.

Thus, when Halsey got Nimitz’s message, it read, “Where Is, Repeat, Where Is Task Force 34. The World Wonders.” Halsey assumed this was a sarcastic rebuke from Nimitz and was so angry he literally threw a tantrum of such proportions an aide had to grab him and tell him to pull himself together [150], [267, Chapter 28].

Both the monoalphabetic ciphers discussed in this chapter and Playfair are block ciphers. Each one takes a chunk of plaintext of a specific size (one and two characters, respectively) and applies the algorithm to the chunk. The same key is used on each transformation.

The size of the block is very important. Playfair is much harder to crack than any monoalphabetic cipher because there are so many more variations in the ciphertext that help to obscure features of the plaintext. In the previous example text, “ATTACK AT TEN OCLOCK,” the letter C shows up twice in the plaintext. But because the letters are enciphered in pairs, the letter C is substituted with S in the CL pair and with U in the OC pair. This means one cannot simply assume the most common letter in the ciphertext is a common letter in the plaintext because there is not a one-to-one mapping for each letter.

At the same time, the Playfair block size of two letters is not large enough by modern standards, and it is still subject to cryptanalysis. For example, in the plaintext “ATTACK AT TEN OCLOCK,” you can see that the first four letters of the plaintext are ATTA. Because this is the same letter pair twice (but reversed), they map to the same letters in the row. Thus, the first four letters of the ciphertext are BYYB. An analyst trying to decode the message would know that the plaintext message has some form of palindrome at the beginning, and this would narrow the candidates for the plaintext considerably.

Still, Playfair leads us to our second requirement for a good block cipher: the block size must be sufficiently large.

Introducing Stream Ciphers: The Vigenere Cipher

Not all ciphers are block ciphers, however. Another form of symmetric cryptography is a *stream cipher*. In this section, I will introduce a stream cipher developed in the 1500s that was not generally breakable until 1863! This cipher is the Vigenere cipher.

As I explained, a stream cipher does not work in blocks. It works one symbol at a time but not as a block. This should not be confused with the Caesar Cipher or any other monoalphabetic cipher. A monoalphabetic cipher is a block cipher because it encodes one block at a time. Each block (of one character) is processed the same way and with the same key. In the Caesar Cipher, for example, once the key is set, each block (of one character) always encodes *to the same output*. That is, for a given 1 of the 25 possible shifts, the letter “A” will always be enciphered to the same output and so will every other letter in the alphabet. You can tell if something is a block cipher if, for a given key, the same plaintext block always corresponds to the same ciphertext block.

The Vigenere stream cipher, being a stream cipher, also deals with one letter at a time but wherein the encipherment of any given letter depends on *where that letter is encountered in the stream*.

To illustrate, I will walk through an example. The Vigenere cipher, like the Playfair cipher, requires a code word. For a given plaintext, the code word is repeated as many times as is necessary to make the sequence as long as the plaintext. If we use our Playfair cipher example of ATTACKATTENOCLOCK and the password SECURITY, it would look like this:

```
ATTACKATTENOCLOCK
SECURITYSECURITYS
```

The last copy of SECURITY is cut off once the necessary length is reached.

The encoding of each letter is performed by what is more or less a Caesar Cipher. However, the key for each Caesar Cipher shift is taken from the code word at that point in the stream. In the preceding example, each letter in the plain text is enciphered using the letter of the code word underneath it as the key. So the first character, “A,” would be enciphered using the Caesar Cipher with the key or shift of “S.” Of course, in the Caesar Cipher, “A” is the first letter, so with a shift of “S,” “A” maps to “S.”

The second letter of the plaintext is “T,” but “T” is substituted using a Caesar Cipher with a shift of “E” (instead of a shift of “S”). We can write this out as we did before:

A	B	C	D	E	F	G	H	I	J	K	L	M
E	F	G	H	I	J	K	L	M	N	O	P	Q
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
R	S	T	U	V	W	X	Y	Z	A	B	C	D

Following this table, you can see that the “T” maps to an “X.” So our first two ciphertext letters are “SX.”

Writing out the Caesar Cipher for each letter becomes cumbersome. Instead, it is easier to just treat the key letter as a numerical shift. For example, “S” is the 19th letter of the alphabet, so we shift “A” 18 times. “A” is the first letter of the alphabet, so it is in position 1. Adding 18 to 1 gives us 19. So “A” maps to the 19th letter of the alphabet, or “S.”

On the other hand, “E” is the 5th letter of the alphabet, which is a four-position shift from “A.” Shifting “T” four positions is “X.” For any letters that go past Z, the shift merely wraps around to the beginning.

The full encoding of the plaintext with the password is

```
SXVUTSTRLIPITTHAC
```

The Vigenere cipher was so good that it gained a reputation for being unbreakable. One reason the cipher was strong was because letter frequency analysis is almost impossible. But there are other ways of breaking the cipher.

The biggest weakness of the cipher is the repeating nature of the code word. All that is necessary to break the code is to *know how long the key is*. With no other information, it is still trivial to break because it reduces the problem to breaking the easy Caesar Cipher. The code word SECURITY, for example, is eight letters long. That means that every 8th letter (1st, 9th, 17th, etc.) is all the same Caesar Cipher key. For short messages like ATTACK AT TEN OCLOCK, that probably is not very helpful. But for a much longer message, the cryptanalysis is relatively simple.

Even if the length of the code word is not known, there are not many possible options. An analyst could try lengths between three and ten without too much trouble.

How Strong Is an Encryption Algorithm

One important lesson from the previous historical examples is that the strength of a cryptographic system is often measured by how much ciphertext you have to have before patterns start to emerge. Vigenere is certainly by cryptanalysis, but to do so requires significantly more ciphertext than for the Playfair cipher. And the Playfair cipher requires more ciphertext to be broken than the Caesar cipher.

Story Time: Playfair at the Movies

The Playfair cipher made an appearance in the movie *National Treasure 2*. In the opening scene, set in 1865, a man in a Washington D.C. tavern is approached by some shady gentlemen and asked to decode a message. The man informs them that it is encoded with a Playfair cipher that is uncrackable without the key. He is shown a phrase, from which he deduces the key and begins to decode it. Just as he is finishing, he gets word that President Lincoln has been shot and realizes that the people asking for the decoding are connected to it. As he burns the message, he is shot and soon dies.

Later, in modern times, a descendant must crack the code again for various plot reasons. He and his friends, including a computer whiz, are seen manually putting in random code words to try and decode the message. After some time struggling, they find out the phrase from which the keyword is drawn and decrypt the message.

The film is interesting for accurately showing how the Playfair 5x5 decoding box is set up and how a message is encrypted and decrypted. So far, so good.

But almost everything else is as inaccurate as one might expect from a movie that depicts the US Declaration of Independence having a treasure map on the back. Right at the beginning, the protagonist’s ancestor had the code word almost immediately. He should have been able to decode the message in a few minutes. But instead, the movie depicts it happening during the time it takes an accomplice to leave the tavern, get to the stage, assassinate Lincoln, and for word to have spread through the city. That is at least 30 minutes. The movie even implies that there is some code “breaking” going on instead of just straightforward decryption.

The other funny inaccuracy is the idea of manually putting in random words as the key to see if it is the right one. A real computer whiz would have simply downloaded a dictionary file and had the decryption program automatically try for one right after another. The keyword would have been found pretty quickly.

Although our modern cryptographic systems are so advanced beyond these classical systems that any comparison is almost useless, some of these algorithms have recommendations on a maximum amount of data that should be encrypted with a key before the key should be discarded and a new one used.

Foundations

Now that you have seen a bit about secret codes and encryption in history, it is time to lay the foundation for modern cryptography.

According to the Handbook of Applied Cryptography:

> *Cryptography* is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. [179]

There is a lot to unpack from this definition. First, what is information?

Information—Binary Data

By way of explanation, "information" is axiomatic. That is, the word is not more formally defined. Information is presumed to be any kind of information but, for practical purposes, can be expressed *as a number*. In the world of computers, all data is numbers. A digital video? Numbers. A video game? Numbers. A spreadsheet? Numbers. Just as importantly, any data, no matter how big, can be expressed as a *single number*, but the bigger the data, the bigger the number.

A more thorough explanation of how computers use numbers is discussed in Appendix B. Briefly, however, the key thing to understand is that all data stored, transmitted, or operated on in a computer is represented as numbers (or a number). And, within the computer, the numbers are represented as *binary* numbers, meaning they use only ones and zeros to represent the numbers.

Here are the numbers zero through nine written in binary form:

```
0     0
```

```
1      1
2     10
3     11
4    100
5    101
6    110
7    111
8   1000
9   1001
```

Appendix A explains how binary numbers work in more detail. But as long as you are willing to believe that every number can be represented with ones and zeros, you know enough to move forward.

Because all data in a computer is just binary data (i.e., a binary number), the cryptographic math works the same on video data as it does on documents, and the same on PDFs as it does on email messages. Basically, cryptography is just mathematical operations on numbers, and every piece of data is a number. Some of the examples in this chapter will visualize actual binary data to illustrate how the cryptography works.

As explained in the appendix, because binary is so long and cumbersome it is often written in hexadecimal as a kind of shorthand. So, unless it is absolutely necessary otherwise, all binary numbers will be written in this hexadecimal short form. So, for example, here is a sequence of binary numbers and the corresponding hexadecimal:

```
101 1100 1010 0011 0111
5   c    a    3    7
```

Because “10” can be “10” in binary, “10” in hexadecimal, or “10” in decimal (our normal numbering system), I will either tell you explicitly what type of numbers I am showing you or I will use prefixes. The prefix for binary is “0b” (e.g., `0b1001`), and the prefix for hexadecimal is “0x” (e.g., `0x5ca37`).

It is not particularly important that you can convert back and forth between binary and hexadecimal or that you understand why computers use binary in the first place. All that you need to know is

1. All data (video, audio, documents, email, web pages, etc.) is just numbers.
2. Computers store and process all numbers as binary numbers (ones and zeros).
3. Cryptography math works on numbers.
4. Hexadecimal numbers are a shorthand way of writing binary numbers.

Because data in computers is stored and transmitted as ones and zeros (binary numbers), sizes are expressed in terms of ones and zeros. A single one or zero is called a *bit*. Or, in other words, a single bit of storage in a computer can hold either a one or a zero.

For various historical reasons, bits are grouped by 8s. Eight bits is called a *byte*. Kilobytes hold approximately 1000 bytes (exactly 1024 bytes), megabytes hold approximately 1,000,000 bytes (exactly 1024×1024 bytes), and gigabytes hold approximately 1,000,000,000 bytes (exactly $1024 \times 1024 \times 1024$ bytes).

One more metric may help think about these sizes. For most western alphabets, a single character (meaning, a single letter, number, punctuation, or other symbols) can be represented by a single byte. So, the word “hello” could be stored in five bytes of data. The sentence “This is a test!” could be stored in 15 bytes (don’t forget bytes for the spaces and the exclamation point!). So, a kilobyte could store about 1000 characters, and a megabyte could store about 1,000,0000. If you look at the size of a document that can be formatted, like a Microsoft Word document, you will notice that the size of the file is quite a bit larger than the number of letters and symbols in the document. That is because the file also has to store significant data about fonts, styles, formatting, and so forth. Still, this hopefully helps you think about the approximate size of something when working with bytes.

Information Security Goals

Returning to the definition of cryptography, the Handbook of Applied Cryptography states that the common goals of information security are confidentiality, data integrity, entity authentication, and data origin authentication. Conveniently, these four goals are not only very common but very instructive. Discussing these goals will help you understand what we need cryptography for.

The first goal is *confidentiality*. Confidentiality is about making sure information is only readable by authorized parties. Sometimes, this is also called *secrecy*. This term can be a little confusing because some people use “secret” to mean something not talked about or something that is unknown. In information security, the data is still transmitted, and its existence is not necessarily hidden. But confidentiality means that the unauthorized parties cannot understand the data. Usually, this means that data is *transformed* such that the data is rendered *meaningless* to anyone without authorization.

The second goal is *data integrity*. In information security, data integrity means that data cannot be *undetectably* altered. In other words, there is some authorized form of the data. If the data has integrity, parties will be able to determine if the data has been changed in an unauthorized way, such as forgery.

The third and fourth goals are presented together. *Entity authentication* involves knowing an authorized identity of a party, as discussed in Chapter 2. In that chapter, we talked about a number of methods for identification, but primarily *human* authentication. In this chapter, you will see that mathematical codes can be used to identify *nonhuman* parties to each other. *Data origin authentication*, on the other hand, is about proving the source or publisher of data. That is, identifying the party that either created or transmitted the data. Data origin authentication includes data integrity because if the data is modified, so is the party that created it.

To help illustrate these concepts, imagine a prisoner communicating with an accomplice outside of the prison.[4] These two would like to plan to break the prisoner out of prison. But the prisoner and the accomplice know that the prison staff will be opening and reading all letters sent between them. How can they have information security on their communications?

First, the prisoner and the accomplice would like the prison staff to not know about their breakout plans. They need a way to send letters that, even if opened and read, will reveal nothing (correct) about their plans. In other words, the prisoner and the accomplice want *confidentiality* on their communications even though they know they will be intercepted.

But the prisoner and the accomplice realize they need something else. Maybe the prison guards already have suspicions about a jailbreak. Maybe the prisoner is being carefully watched. Maybe the prison would like to identify an accomplice to arrest and imprison them too. What if the prison sent fake letters to induce them to reveal themselves? What if the prison sent a letter telling the accomplice to show up on a certain date and a certain time in order to arrest them?

The prisoner and the accomplice need to be able to identify themselves to each other. They need to be able to prove that they really are the prisoner or the accomplice, respectively. This is *entity authentication*.

Even with entity authentication, the prison could still *alter* letters. For their information security purposes, the prisoner and the accomplice must also have a way to tell that all the data they receive comes from the other. Any alterations, including additions and deletions, must be detectable. If these two can come up with such a solution, it is an example of *data origin authentication* and inherently includes *data integrity*.

In cyberspace, we have some of the same problems as the prisoner and the accomplice. Our data communications, by the very nature of how computer systems work, *are being handled by parties we cannot trust*. Data between your computer's browser and your bank, for example, are being handled by a number of intermediate systems that are used in transmitting the data back and forth between the two. But these intermediate parties should not be trusted with your bank data. Unprotected data intercepted by these intermediates can be read and, in many cases, altered or forged. Fortunately, cryptography can be used to provide the necessary information security properties to secure these communications so that even when an intermediate is handling the data, they cannot abuse it.

Cryptography is largely based on operations that involve some kind of *key*. A key is data that controls a cryptographic operation in some way or another. Generally speaking, without the correct key, an operation cannot be performed correctly. Keys are typically security-critical data.

As briefly mentioned earlier in the chapter, there are two major forms of cryptography in use today. One form is called *symmetric key* cryptography (or just *symmetric cryptography*), and the other is called *asymmetric key* cryptography (or just *asymmetric cryptography*). Symmetric cryptography is built around operations that use a single key. Asymmetric cryptography is built around operations that use two keys: a public key and a private key. Common algorithms and uses will be discussed in the next chapter.

However, there are two core concepts that need to be understood first: *XOR* and *hashing*.

XOR

Working with the ones and zeros of binary data opens up an interesting form of mathematics called *Boolean Algebra*. A *boolean* is a true or false value, and the comparison to a one or a zero should be intuitive. Boolean Algebra is the mathematics around true and false.

For example, Boolean Algebra has AND, OR, and NOT operations. AND and OR operations have two boolean inputs each and one output. In the case of AND, both inputs must be true in order to get a true output (true AND true is true). In the case of OR, either input must be true in order to get a true output (true OR false is true). NOT takes a single input and inverts it (NOT false is true). This is easily applied to binary numbers. The following table is called a *truth table,* and it depicts the inputs and outputs of the AND operation:

Input 1	**Input 2**	**Output**
0	0	0
0	1	0
1	0	0
1	1	1

On the other hand, the truth table for OR looks like this:

Input 1	**Input 2**	**Output**
0	0	0
0	1	1
1	0	1

Input 1	Input 2	Output
1	1	1

Hopefully, the concepts of AND and OR are clear. In the AND operation, both input 1 AND input 2 must be 1 in order for the output to be 1. In the OR operation, either input 1 OR input 2 must be 1 in order for the output to be 1.

But beyond AND, OR, and NOT is another operation called Exclusive Or or XOR. Exclusive Or only produces true values when *either but not both* inputs are true. The truth table for XOR is as follows:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Again, notice that if either input is true, the output is true. But if *both* inputs are true, the output is false.

XOR is one of the most magical mathematical operations ever devised. It may not look like much when you are working with one bit at a time, but the true power is revealed when working with sequences of bits. When using XOR on two sequences of bits, each pair of bits is combined separately as shown in the following:

	11011011
⊕	10110001
	01101010

Starting with the rightmost bits, we have a 1 and 1, so the output bit is 0. Moving one bit to the left, we have 1 and 0, so the output bit is 1. The next bits are 0 and 0, so an output bit of 0. And so on.

Here is where the magic starts to come in. XOR *is its own inverse*. Watch what happens when we take the output from the previous problem and XOR it with the first input:

	11011011
$\oplus$	01101010
	10110001

The output is 10110001! This was the second input in the first problem! In short, if you start with some binary number *A* and XOR it with *B*, you can recover *A* by XORing the output with *B* again. Mathematically, it looks like this: $(A \oplus B) \oplus B = A$.

More intuitively, it is literally like a magic show. XOR made a number disappear and then reappear! You can take any number and "hide" it by XORing it with another number. Then you can make the number come back by XORing it with the same number again!

XOR will be used in a number of very powerful cryptography operations in later sections. This crucial ability to completely change data and then change it back to its original form is a critical part of many algorithms.

Hashing

One of the most crucial concepts for understanding many other cryptography operations is *hashing*. Hash functions are designed to produce *fingerprints* on computer data.

You had a very brief introduction to hashing in Chapter 2. All I explained in that part of the book is that hash functions are "one-way" functions. They take input data, like passwords, and spit out what looks like random data. If you recall, for security reasons websites typically store the hash of a password instead of the password itself. Now you will dig a little deeper into what hashing really is and what it means.

First, when we say “function,” what do we mean? In this context, a function is just like the ones you learned about in your junior high and high school algebra classes. A function takes an input and produces an output.

```
f(x) = y
```

If, again, you never liked math, do not let the terminology confuse or frustrate you. A simpler explanation is that a function is a *transformation*. These transformations are given names. Names can be specific, but letters like *f* and *g* are often used as generic names for these transformations. Inputs are expressed in between the parentheses that follow the function name:

```
function_name(input) = output
```

Functions transform input data into output data according to some kind of mapping including mathematical formulas. For example, imagine a function that adds one to any input. This function transforms data in a very simple way:

```
f(x) = x + 1
f(0) = 1
f(10) = 11
f(25) = 26
f(-13) = -12
```

The function *f* in the preceding example transforms 0 into 1, 10 into 11, 25 into 26, and –13 into –12.

One requirement for a function is that for any given input there is *exactly* one output. In other words, any given function must transform 0 into the same output every time. However, it is perfectly fine for a function to transform two inputs to the same output. Imagine if our function *f* output 1 if the number was even (divisible by 2) and 0 if the number was odd:

```
f(1) = 0 (odd)
f(2) = 1 (even)
f(3) = 0 (odd)
f(4) = 1 (even)
```

In this example, the function only ever has two outputs. An infinite number of inputs will transform to 1, and an infinite number of inputs will transform to 0. But any given input will only ever transform to 1 or 0, but not both.

Turning back to a hash function, a hash function, like any other function, transforms input data to output data. As discussed at the beginning of the chapter, *all* computer data is a number. Even a huge video file is just a really, really, really big number. Hash functions take an input of any size and transform it into an output of a relatively small size. This is sometimes called a *digest* function.

To better explain this, we will use computer sizes to talk about how big or small the numbers are. Remember, all computer data is a number, and all numbers are stored in a computer as binary data measured in bits or bytes. So, when I say that a hash function takes an input of any size and transforms it into a relatively small output, what kind of sizes am I talking about?

Any size means any size. The input to a hash function can be as large as the computer can hold. Video files can be hundreds, or even thousands, of gigabytes. All of that can be the input to a hash function. To be clear, however, the input to a hash function does not have to be large. What is important is that there are no limits (beyond the computer's capacity to store the data).

The output size of a hash function is fixed. That means for a specific hash function, the output is always the same number of bits. On the low end, some hash functions produce outputs that are 128 bits (16 bytes). On the larger end, hash functions spit out hashes that are 512 bits (64 bytes). In this case, these are very small sizes as far as computer data goes. As explained earlier in the chapter, 64 bytes can hold at most 64 characters (letters, numbers, and punctuation).

Why would anyone want to transform data, potentially very large data like video files, into just 16 to 64 bytes of output? What good can possibly come from that?

As stated at the beginning of the section, hashing is designed to produce *fingerprints* for data. Just like physical fingerprints, a fingerprint of data does not have to be large so long as it is sufficiently effective at identifying the data. As long as the 16 to 64 bytes of hash data can identify gigabytes of video data, it serves its purpose.

The actual formulas behind hashing functions are complicated, and I will not attempt to describe them. Instead, it is more important that you understand what they do at a qualitative level. One way of thinking conceptually about hash functions is to imagine that they assign a *random* number of the appropriate size to every possible input. Once assigned, the same input generates the same random number every single time.

Imagine a magically perfect hashing function we will call *h*. This magical function will do exactly what I just described. The first time it hashes an input, it will magically assign a random input of exactly 64 bytes (512 bits). If it ever hashes the same input again, it will magically produce the same output.

Continuing this example, suppose we have three inputs. The first is the data of an office spreadsheet, the second is a music file, and the third is an entire folder of documents related to the design of a top secret design of a new technology (compressed into a single archive file). If we hash each one, we will get three outputs, each one of exactly 64 bytes. Written in our pseudomath notation:

```
h(spreadsheet) = hash1
h(music_file) = hash2
```

```
h(design_docs) = hash3
```

Because our function is magical, it picked perfectly random numbers of 64 bytes for each one.

How would each hash work like a fingerprint? If each number is picked at random, the odds that the magic function picked the same hash number for each two inputs are very small. Without explaining why, the odds are one in 2^{256} . That's two raised to the 256 power, or multiplying two by itself 256 times. The actual number is $115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936$! That number is 78 digits in length! In other words, the odds of two inputs producing the same output, while technically possible, are impossible in practice. Thus, each hash becomes a kind of fingerprint for the input data because it is impossible in practice that any other data will have the same hash output.

Also, recall that our magical function will output the same hash if we do use the same input in the future. Pretend that somehow each of these pieces of data was altered. Somebody updated the spreadsheet, the audio file was corrupted by a virus, and an industrial saboteur changed the design of the new technology to make it not work.

```
h(updated_spreadsheet) = hash4
h(corrupted_music_file) = hash5
h(altered_design_docs) = hash6
```

If we have access to the original hash outputs (e.g., hash1, hash2, hash3), then we can quickly determine that the data has been changed, because when we recompute the hash, the output will be different.

If a hacker wanted to alter the data undetectably, they would need to be able to create altered data that produced the same hash output. As I explained, this is *theoretically* possible because there are an infinite number of inputs and only as many outputs as fit in 64 bytes. Could an attacker figure out some way to come up with changes to the data that produce the same hash?

The hacker could try to take a hash value and work backward. Imagine that the hacker wants to change the spreadsheet. The hash for this data is hash1. The hacker could try to find another spreadsheet that produced hash1 as well. How hard would that be?

Really, really hard. In hashing terminology, the hash is called an *image*, and any input that hashes to a specific hash is called a *preimage*. For our magical hashing function, the image is assigned randomly. So the hacker cannot predict in any way what preimages produce the hash1 image. The only option is to hash random inputs (preimages) and *hope* that they happen to hash to the right hash output (image). For our hash function with output of 32 bytes (or 256 bits), the hacker would have to try approximately 2^{512} different inputs before finding a matching hash output. I will not even bother writing out this huge number: it is 155 digits long. That is how many spreadsheets the attacker would have to try before they are likely to find one that produces the same hash output. Even with fast computers, that is considered very unlikely.

Of course, real hash functions are not magic. The outputs of these functions are not perfectly random but are based on complicated formulas. But the formulas are *designed* to have the same or very similar properties as the pretend and magical function we were imagining. For one, the output should be *unpredictable*. The true hash algorithm should produce an output that is more or less indistinguishable from the magical random assignment of our imagined function. Even the slightest change in the input should completely change the output such that there is no correlation between the input and the output hash.

Here are some of the other features the hash function needs to have:

- **Compression**: No matter the input size, the output size (in bits) should be fixed.
- **Preimage Resistance**: Given a hash output (image), it is impractical to find a matching preimage (e.g., because it would require trying 2^{512} different preimages to find a matching 64-byte hash).
- **Collision Resistance**: It is impractical to find any two inputs that hash to the same output (e.g., because it would require trying 2^{256} different input pairs to find two that produce the same 64-byte hash).

Because real hash functions are based on formulas, and not magic, it sometimes turns out that very smart (and/or very determined) people discover that there are flaws in the formulas. These flaws weaken these properties and make the hash function more predictable than it is supposed to be. That is, a “good” hash function should be more or less indistinguishable from the magic hash function described earlier. If someone figures out how to more easily find collisions and/or preimages than completely random guessing, the hash function is considered to be *broken*. Typically, a broken hash function is obsoleted and replaced with a newer, (hopefully) more secure alternative.

In the world of hashing, some of the common algorithms that are used or have been used are

- **MD5**: 16-byte (128-bit) output. Broken and obsolete.
- **SHA1**: 20-byte (160-bit) output. Broken and obsolete.
- **SHA-256**: 32-byte (256-bit) output. Still used.
- **SHA-512**: 64-byte (512-bit) output. Still used.
- **RIPEMD**: 16-byte (128-bit) output. Broken and obsolete.
- **RIPEMD-160**: 20-byte (160-bit) output. Still used.

Just because I have listed an algorithm as broken and obsolete does not mean it is never used. Sometimes, the use of a hash is not security critical, and an obsolete hash does the job just fine. Or perhaps the system is, itself, out of date, and it is using older technology. Or, it may be that it is something that needs to be replaced, but it will take time before the switchover can be made for any number of logistical reasons.

But the most important thing to understand is that hashing algorithms are crucial and ubiquitous components. They are used by themselves for many purposes. But as you will see in the subsequent sections, hashes are also used for various symmetric key and asymmetric key operations.

Summary

In this chapter, you were introduced to historical approaches to encryption as well as some of the goals and prerequisites for modern cryptography.

Historical encryption can be very helpful for getting an intuition around concepts used in symmetric cryptography. These include

1. **Key Space**: This is how many possible keys there are for an algorithm. If there are too few (like in the Caesar Cipher with just 25 keys), an attacker can easily break a message using *brute force*.
2. **Block Size**: For block ciphers, the block size must be large enough. The Playfair cipher is harder to cryptanalyze than the Caesar Cipher because it works on blocks of two characters instead of one.
3. **Stream Ciphers**: Do not encrypt a block at a time. Instead, they process each symbol in the stream separately, using its place in the stream as part of the encipherment process.
4. **Key Expansion**: A key can be expanded to fit certain size requirements. In Playfair, there needs to be 25

size requirements. In Playfair, there needs to be 25 letters of key material. But Playfair's algorithm includes a mechanism to expand a code word of any size to the required 25 letters.

5. **Padding**: Sometimes, encryption algorithms require the plaintext to be of a certain size. Playfair, for example, requires an even number of characters. When the plaintext is not long enough, padding characters can be added to make up the difference. These must be removable after the decryption process.
6. **Cipher Strength**: One measurement for the strength of a cipher is how much ciphertext must be generated before patterns or other weaknesses emerge.

Although modern cryptography is almost unrecognizably different from these historical examples, understanding these examples will make these concepts easier in the next chapter.

Modern cryptography is interested in more than just *confidentiality*, one of the common goals of encryption. Some of the other relevant goals are *data integrity*, *entity authentication*, and *data origin authentication*.

Finally, to be prepared for the next chapter, you learned about XOR and some of its "magical" properties. Specifically, XOR is able to "hide" information in random or pseudorandom data. Moreover, because XOR is its own inverse, information hidden inside random or pseudorandom data can be easily reextracted.

You also learned about hashing algorithms, which produce short codes calculated on arbitrary data. These short codes act like fingerprints for the input data.

Further Reading

For a much deeper dive into the technical aspects of cryptography, including the security goals, theoretical background, and extensive mathematical analysis, I recommend the Handbook of Applied Cryptography, which is freely available online [179].

If you are interested in the history of cryptography, Kahn’s book *The Codebreakers* [147] always comes highly recommended. You can also find books about intelligence and code breaking that focus on specific time periods in history. For example, the book *Most Secret and Confidential: Intelligence in the Age of Nelson* [168] looks at all the different means by which Great Britain, and especially its Navy, acquired intelligence during the Napoleonic years. The operational issues, such as the use of fast frigates to spy on a port, the interception of diplomatic mail (which did involve some code breaking), and even semaphore reading, provide insight into much information there is to gather, the various means of obtaining it, and the difficulty in putting it altogether.

World War II is a really amazing time period for cryptography and cryptanalysis. Fortunately for the Allies, there was effective code breaking of the Axis codes in both the Pacific and European theaters. Many of the books about the Battle of Midway discuss to some degree or another the US naval code breakers that deciphered the Japanese codes that enabled them to predict the time, location, and configuration of the Japanese flee at Midway [204, 258]. On the European side, one of the most important stories is that of Alan Turing, who was instrumental in breaking the German enigma machines [242, 281].

References

30. Adkins, A. 1997. Secret war: The navajo code talkers in world war II. *New Mexico Historical Review* 72(4): 10.

36. Almeshekah, M.H., E.H. Spafford, and M.J. Atallah. 2013. Improving security using deception. Technical Report 13, Purdue University, 11. CERIAS Tech Report 2013-13.

39. Anderson, R.J. 1993. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS'93)*, New York, 215–227. Association for Computing Machinery.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing.
[Crossref]

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

66. Bruce, S. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.-2nd*. Wiley.
[zbMATH]

139. Huffman, S. 2000. The navajo code talkers: A cryptologic and linguistic perspective. *Cryptologia* 24(4): 289–320.
[Crossref]

143. Johansson, J.M., and R. Grimes. 2008. The great debate: Security by obscurity. *TechNet Magazine*.

147. Kahn, D. 1996. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner.

150. Kennedy, D.M. 1999. Victory at sea. *The Atlantic Monthly* 51–76. www.theatlantic.com/magazine/archive/1999/03/victory-at-sea/306272/.

168. Maffeo, S.E. 2000. *Most Secret and Confidential, Intelligence in the Age of Nelson*. Naval Institute Press.

179. Menezes, A.J., S.A. Vanstone, and P.C.V. Oorschot. 1996. *Handbook of Applied Cryptography*, 1st ed. Boca Raton: CRC Press, Inc.
[zbMATH]

204. Parshall, J., A. Tully, and J.B. Lundstrom. 2005. *Shattered sword: The untold story of the Battle of Midway*. Washington, DC: Potomac Books.

242. Siddiqui, R. 2013. Alan turing: A note on his role as world war II cryptanalyst. *International Journal of Applied Engineering and Technology ISSN: 2277-212X (Online)* 3: 21–26.

258.

Symonds, C. 2011. *The Battle of Midway*, Pivotal Moments in American History. Oxford University Press.

267. Tuohy, W. 2007. *America’s Fighting Admirals*. Zenith Press.

281. Wilcox, J. 2015. *Solving the ENIGMA: History of the Cryptanalytic Bombe*. National Security Agency Center for Cryptologic History.

Footnotes

1 Obviously, the Roman alphabet was not exactly the same, but hopefully the idea is clear.

2 While security through obscurity is largely viewed as harmful, there is still a debate about it [143].

3 Although not shown here, there is a way of converting each permutation into a number so that it is easy to identify the key beyond just writing out the entire permutation.

4 I am not endorsing clandestine communications between violent criminals. If this example causes you any moral consternation, you may consider the prisoner to be a political prisoner that protested against an ideology of your choice.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_5

5. Core Cryptography Technology

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide

This chapter provides an overview of modern *symmetric encryption* and *asymmetric encryption*.

Key Concepts

1. *Symmetric* cryptography uses the same key for the inverse of an operation.
2. *Asymmetric* cryptography uses a public and private key pair.
3. *Encryption* is often used to provide data confidentiality.
4. *Message Authentication Codes* (MACs) are usually created by symmetric algorithms for data integrity.
5. *Signatures* are usually created by asymmetric algorithms for data integrity and entity authentication.
6. “Proving” the identity of a party (i.e., for entity authentication) typically requires having some kind

of axiomatic starting point.

Common Pitfalls and Misunderstandings

1. You Are Not A Cryptographer (YANAC); never try to create your own cryptographic algorithms or systems.
2. Confidentiality does *not* imply data integrity; unreadable data *can* be meaningfully altered!
3. Authenticated Encryption with Additional Data, or AEAD, combines confidentiality and integrity.
4. Private keys should not be used after disclosure.

Useful Vocabulary

- **MAC**: Message Authentication Code[1]
- **OTP**: One-time pad
- **AES**: Advanced Encryption Standard
- **IV**: Initialization vector
- **nonce**: Number used once
- **ECB**: Electronic Code Book
- **CBC**: Cipher Block Chaining
- **CTR**: Counter mode
- **AEAD**: Authenticated Encryption with Additional Data
- **GCM**: Galois Counter Mode
- **ChaCha20**: A newer symmetric stream cipher that is an alternative to AES-CTR
- **ChaCha20-Poly1305**: An AEAD version of ChaCha20

In the previous chapter, I introduced a few basic concepts about cryptography. This was meant to help prepare you for a discussion of modern cryptographic technologies. In this chapter, I will walk you through some of the core cryptographic techniques used to protect information throughout cyberspace. Although I will dig into the mathematics *a very little bit*, I will primarily describe what it *does* rather than how it *works*. And I will also use some overly simplistic examples and analogies to help explain terms and concepts. By the end of the chapter, you will be able to understand what cryptography is capable of doing when done right, some of the configuration necessary to do it right, and why it is difficult to do it right.

As a side note, most of these techniques do not work, or do not work well, in isolation. In the next chapter, I will discuss operational cryptographic technologies that combine multiple cryptographic techniques, computer networking operations, and other components into a cohesive security system.

Symmetric Cryptography

As explained in the previous chapter, symmetric cryptography is so named because a single key is used symmetrically. Most cryptographic operations come in pairs. The two sets of operations discussed in this section are encryption/decryption and Message Authentication Code (MAC) generation/verification. All four of these operations require the use of a key. In symmetric cryptography, *the same key is used for both operations in a pair*.

In this section, I will walk through how these algorithms work and what is necessary for them to be secure. After all, just because we have protected information does not mean that an adversary will not try to *unprotect* the information. A lot of our discussion will be focused around how the cryptography can be broken if not done correctly.

Modern Block Ciphers

Modern block ciphers work differently than the classical ciphers just discussed because instead of working on letter symbols, they operate on bits (zeros and ones). There is no significant theoretical difference. If you were paying close attention, you might have picked up on the fact that the letters in the examples were being used like a number. The Caesar Cipher, and especially the Vigenere Cipher, *added* letters together. However, bits allow us to more universally represent all possible data. Using bits, we are not bound to a single alphabet and can encrypt a video as easily as an email message.

A "good" cipher should have at least the following qualities:

1. Large block size
2. Large key space
3. "Avalanche" property

Large block size and key space should be somewhat familiar given our study of the historical examples. In modern block ciphers, it is expected that block size will be at least 128 bits. Key sizes are generally no smaller than 128 bits as well, but for various reasons, 256 bits will become the minimum size in the not too distant future.

The *Avalanche* property does not really show up in our historical examples because our block size of one or two letters is generally too small to see it. But the basic concept is that any change, no matter how small, should completely change the output. By "completely," I mean that approximately 50% of the output bits should flip if the input changes, *even if the change in the input is only one bit*.

This property is important because if less than 50% of the bits are changing for certain inputs, this may allow cryptanalysis to make predictions about inputs and outputs. This would, of course, weaken or break the code. There are other cryptanalysis approaches that “good” ciphers should be resistant to, but those are outside the scope of this book. Suffice it to say that a lot of work goes into making these block ciphers resistant to any kind of analysis.

Like our Playfair and Caesar Cipher examples from the last chapter, modern block ciphers have the same size inputs as outputs. That is, the block size specifies the number of bits that are used as inputs and the number of bits that are output from the function. Because the input and output are the same size, the block cipher can be described as a *permutation* function. I used the word “permutation” before to describe reordering the alphabet. Here, it refers to reordering bits.

Suppose our block size is 128 bits. Any input will be one of the possible combinations of 128 ones and/or zeros. There are only so many possible combinations (2^{128} to be precise). The output will also be one of the possible combinations of 128 ones and/or zeros. Our block cipher is said to provide a permutation by taking one of the possible 2^{128} sequences as an input and spitting out another one of the possible 2^{128} sequences as an output.

This permutation is *keyed*. That is, the block cipher takes some kind of key as an input, and the key determines the permutation. If keys are also 128 bits, then our key space is 2^{128} possible keys.

In the section on hashing, I described a magical hash function *h* that worked “perfectly.” That is, it created a perfect hash function with perfectly random outputs. Here, I will describe a pretend, magical block cipher function *b* that works similarly. As with hashing, there is no perfect function in reality, and there is no magic either. But this pretend magic block cipher function will help to define what imperfect, real-world functions *try* to approximate.

For our pretend, magical function $b\_encrypt$, the function will take a key and a plaintext as inputs and spit out a permutation as the ciphertext. For this example, the key, plaintext, and ciphertext are all 128 bits:

```
b_encrypt(key, plaintext) = ciphertext
```

This magical function $b\_encrypt$ can tell if it has ever seen the key and plaintext pair before. If not, it (magically) assigns a perfectly random permutation (a completely random selection of 128 bits) as the output. If it has seen it before, it just outputs the same value that it did previously.

The $b\_encrypt$ function also has an inverse function called $b\_decrypt$. This function, also using magic, takes a key and ciphertext as input and provides the correct plaintext:

```
b_decrypt(key, ciphertext) = plaintext
```

In other words, this function can (using magic) find the key and plaintext pair given to the $b_encrypt$ function that produced the ciphertext. Using this magical knowledge, it can produce the plaintext as an output.

The magic of the random assignment is important. If the assignment of the ciphertext permutation to the plaintext was truly random, there is no cryptanalysis that can reveal any information about the permutation. A completely and perfectly random assignment of a plaintext block to a ciphertext block cannot be scrutinized for patterns or clues because random choices have no patterns or clues.

Unfortunately, we do not have any magic that can do this kind of perfectly random assignment. However, we have mathematical functions that can work so well that it is difficult to tell the difference between them and the magical random functions just described. Cryptographers and mathematicians call these functions *pseudorandom* functions. But do not let the *pseudo* part of the name fool you. That is not meant to be an insult or some kind of negative modifier. The functions are not truly random, so they must be, by definition, *pseudo*random. But there are pseudorandom functions that are so good that when used properly it is impossible to tell if the output is from a pseudorandom function or a truly random function. Modern block ciphers are expected to have this kind of output (specifically, block ciphers are a subset of pseudorandom functions called pseudorandom permutations because their output is always a permutation of the input, as discussed earlier).

That is not to say that weaknesses are never found. Sometimes, after a lot of study and analysis, a block cipher is shown to be “broken,” meaning there are ways of finding patterns or artifacts. Or, in other words, the block cipher is demonstrated to not operate like the magic random function. If the failure is significant, then the function is obsoleted and retired.

One example block cipher is the Data Encryption Standard (DES) that was introduced in 1975. This cipher is obsoleted for many reasons. Its key space is too small (56 bits), and its block size is also too small (64 bits). But it turns out that the function itself is flawed. Between 1993 and 2001, researchers demonstrate that with enough known pairs of plaintext and ciphertext, the key could be discovered with high probability. By a pair, I mean the plaintext and its associated ciphertext for a given DES key. With a large number of these pairs, between 2^{39} and 2^{43} pairs, the key can be recovered with about an 80–85% success rate [145, 152, 175].

To be clear, this attack, although demonstrable, is unlikely to occur in actual practice. It is difficult to imagine an instance where an attacker has between $2^{39} = 549{,}755{,}813{,}888$ and $2^{43} = 8{,}796{,}093{,}022{,}208$ plaintext-ciphertext pairs. Nevertheless, DES is considered extremely unsafe. Even if the current attacks may not be the most practical in person, their existence is seen as foreshadowing practical attacks in the future.

This is also a good time to talk about the different ways of "attacking" a cipher. In the preceding example, the attacker has to have a sufficient number of plaintext-ciphertext pairs. This is called a *chosen plaintext* attack. That is, the attacker chooses a plaintext and gets to see the output (for a given key). Why would an attacker ever get to see this? The whole point is to keep the attacker from seeing the plaintext, right?

It all depends on your *threat model*. An account holder communicating with their bank could know the data being sent to their bank and then tap their own line to see the encrypted data. Or, an attacker may have a way of tricking someone into encrypting a message on their behalf and observing the output.

The flip side of a *chosen plaintext* attack is the *chosen ciphertext* attack. As you might already be guessing, this means that the attacker can submit ciphertext samples to the system and observe the decrypted data. *Chosen plaintext/ciphertext* permits submitting either plaintext or ciphertext to the system and observe the output.

A different style of attack is a *related key attack*. In this model, an attacker varies the key slightly each time and monitors the output for patterns [66, Chapter 12].

Someone that wants to attack or break a cipher uses attacks such as these to see if they can get the real-world cipher to reveal in what ways it might not live up to the theoretical ideal set by the pretend, but magical, block cipher discussed earlier. For example, the preceding DES attack, even if relatively impractical, shows that DES is weaker than the magic cipher described by $b\_encrypt$ and $b\_decrypt$. Remember, $b\_encrypt$ uses magic to assign everything perfectly randomly. For the magic cipher, it does not matter if an attacker has 2^{39} or 2^{43} plaintext-ciphertext pairs. *Everything* is assigned randomly so no amount of data observed reveals *anything* about any other permutations. The "attack" on DES, from a certain point of view, quantifies how not perfect (or not magical, if you prefer) the *pseudorandom* function is.

Even if you have not followed all of the math in this chapter, you should try to understand the concepts of how attackers work. Real attackers have been able to get specific plaintexts encrypted for their analysis (chosen plaintext), and real attackers have been able to get specific ciphertexts decrypted for their analysis (chosen ciphertext). When you are thinking about threats to a system that relies on cryptography, you should generally assume the attackers have these kinds of capabilities.

Advanced Encryption Standard

The most commonly used block cipher today is called AES or Advanced Encryption Standard. That is not the original name. There was a competition to replace DES with a newer algorithm, and many different block ciphers were submitted. The algorithm created by Joan Daemen and Vincent Rijmen, known as *Rijndael* (pronounced "rain-dahl," this word is a combination of the last names of the authors), was the winner of the competition and is now known as AES.[2]

AES has a block size of 128 but can have variable key sizes of 128, 192, and 256. AES has been heavily scrutinized over the last 20 years, and a couple of minor vulnerabilities have been found. For example, the very best key recovery attacks reduce the brute-force attack from trying all possible 2^{128} combinations for a 128-bit key to 2^{126}

. Not only would that still take a *billion years* on any hardware we expect to have in the foreseeable future, the technique also required storing approximately 2^{88} bits of data, or more than 38 trillion terabytes of data. That is more storage than the entire earth is using right now [278]. In other words, this attack is not even remotely practical.

Modes of Operation

In summary, AES is a strong cipher. So all we need to do to secure data is encrypt with AES, right?

In Figure 5-1 is a simple image I created. In Figure 5-2 is the same image encrypted using AES. Does that look like very good encryption to you?

What went wrong? Why did AES do such a bad job of encrypting the data?

The answer goes back to block size and properties like Avalanche. Remember that the AES block cipher is just 128 bits. The image is many times bigger than that. To encrypt it, the image is broken up into 128-bit chunks just like messages for Playfair were broken into two-character chunks. AES encrypts each 128-bit chunk one at a time.

But what happens if the same chunk is encrypted twice? AES is a function. The same plaintext encrypted by AES with the same key always produces the same output. If there are *patterns between the 128-bit chunks, those patterns will show up in the output*. In other words, this is an image, not text, which means that the letters are not being encrypted, only the pixels that our minds interpret as letters. Each letter is made up of a number of 128-bit chunks. The patterns between the 128-bit chunks cannot be concealed by AES because AES only encrypts 128 bits at a time. If the encryption takes a chunk of all black or all white pixels in an input block, then the output block is always going to be the same, which means that large white or black regions may repeat patterns. The Avalanche property only applies within the 128-bit input. But a change in one block does not propagate to any subsequent block.

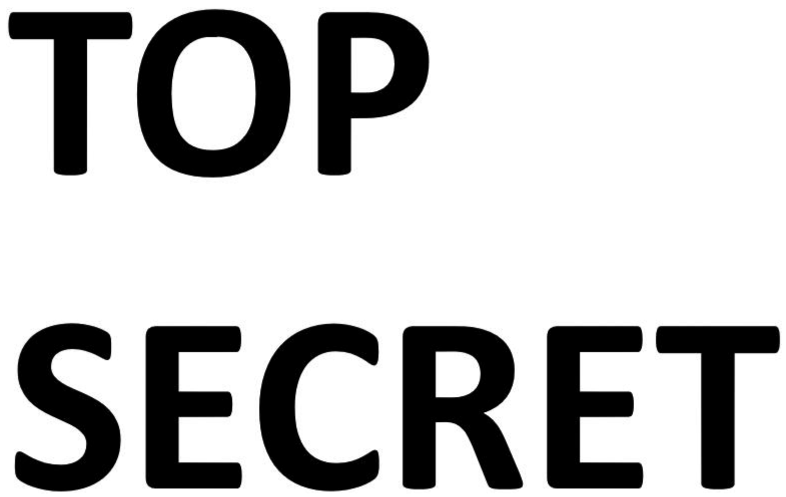


Figure 5-1 An image with the text “Top Secret.” Encrypting it should make it unreadable, right?

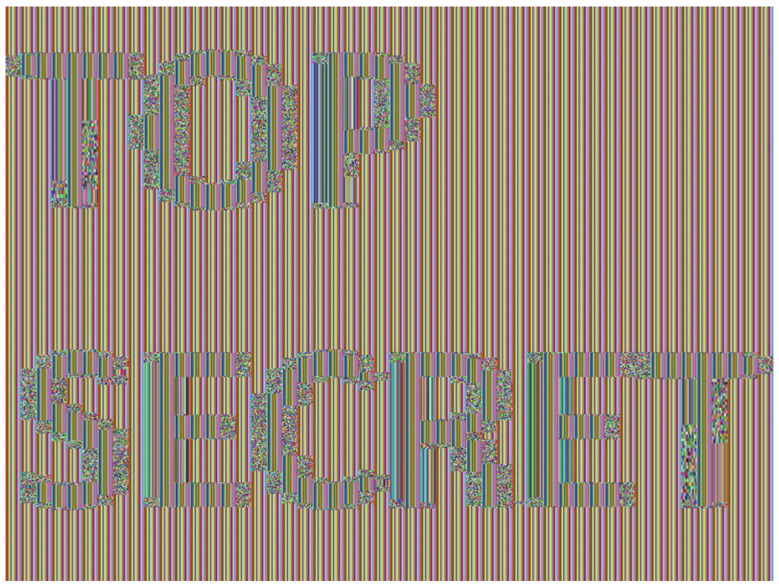


Figure 5-2 This image was encrypted using ECB mode. This message is not very confidential

AES, if we used it this way in practice, would be pretty much worthless. In practice, however, we use “modes of operations” that solve these problems. The naive approach used in this example, that of encrypting each chunk independently, is called Electronic Code Book mode, or

ECB mode. ECB should *never* be used except for testing purposes.

Story Time: Zoom Loves ECB

No matter how basic a cryptographic error is, it seems like you can always find somebody that did it. ECB is well known for being unsuitable for anything except testing. This is basic cryptography. Just about every student in a cryptography class will have seen an image like Figure 5-2. Every cryptographer knows ECB leaves artifacts in the ciphertext that expose details of the plaintext.

And yet, Zoom was using ECB for “protecting” their video call communications in 2020. Admittedly, streaming video is both processor and bandwidth intensive. Still, there are existing techniques that are meant to be used with streaming data. Why Zoom chose to use ECB instead of the better approaches is a mystery. When confronted by security researchers from Citizen Lab, a research group within the University of Toronto, the CEO of Zoom admitted to the problems and promised to resolve them quickly [97].

An interesting question that will never be answered (without a whistle blower, a forensic investigation, or legal discovery) is *why* did Zoom choose to use something that is well known to be a bad idea? One possibility is that they did not have any cryptographic expertise in their engineering organization. That seems unlikely to me, but maybe they deprioritized security to such an extent they created an engineering organization that did not know better. Another possibility is that they had someone with the necessary skill, who warned them this would happen, and they did it anyway. The year was 2020, the Covid quarantine was taking off, and Zoom stocks were on the rise. Maybe they just were rushing.

Whatever the reason, Zoom also broke one of the other cardinal rules of cryptography: don't roll your own [172]. This is an expression about not creating your own cryptographic algorithms or systems. Instead, use well-tested, thoroughly evaluated systems for which there is high assurance. Just as YANAC, even professionals should not develop a system with sufficient time to have the system reviewed by other experts, evaluated, explored, and researched. The goal is always to know about weaknesses *first*, before attackers find out about it.

We need a better mode of operation than ECB to eliminate the structures between blocks. We need something that ties all the blocks together so that changes in one block impact the encryption of subsequent blocks. The Avalanche property should apply to the whole of the data being encrypted, not just 128 bits at a time. In fact, the goal is to encrypt the entire data *as a whole*, not break it into a bunch of separate, individual encryptions of 128-bit chunks.

One way of doing this is known as Cipher Block Chaining mode or CBC mode. Please note, the rest of this section on CBC is going to get a little technical. The goal is to learn a little bit about how cryptography depends on some very picky, very subtle elements to be "correct." The main lessons to take out of the next paragraphs are

- While ECB encrypts in independent separate blocks (which is not secure as seen in Figure 5-2), CBC encrypts all of the data like a single input producing a completely pattern-free output as seen in Figure 5-4.
- CBC requires an *initialization vector*, or IV, as an input.
- The IV can be public; unlike the key, it does not have to be kept secret.
- The IV, although public, does have to be unpredictable to the attacker.

- A different IV needs to be used with each encryption, especially if the same key is used.
- Using different IVs enables output ciphertext of AES-CBC mode to be unique, even if the same inputs are used with the same keys.

If the following explanations about how CBC works are overwhelming, these points are what is most important.

CBC mode is designed to enable the block cipher (e.g., AES) to operate on large data as if it were all being encrypted at once. Recall that AES is a block cipher that is designed to encrypt a 128-bit chunk at a time. CBC extends it so that it is effectively encrypting any size of data together. CBC works by chaining together the various blocks of data. The chaining process works by XORing the ciphertext output of one block into the plaintext input of the next block. Figure 5-3 depicts how this encryption mode works.

Ignore the part of the figure that says IV for now. I will explain it in a moment. Instead, notice that each block is still being encrypted one at a time. However, the second block has the ciphertext of the first block XORed with it *before* being encrypted. Recall from earlier in the chapter that XOR can enable a piece of data to "disappear" or be hidden within another number. Also, remember that the output of AES is unpredictable without knowing the key and the plaintext. So the first output block C1 cannot be predicted by an attacker. That means the attacker also cannot predict what number gets created when the second plaintext block, P2, is XORed with C1. This unpredictable block is then fed into AES again producing C2, which is also unpredictable.

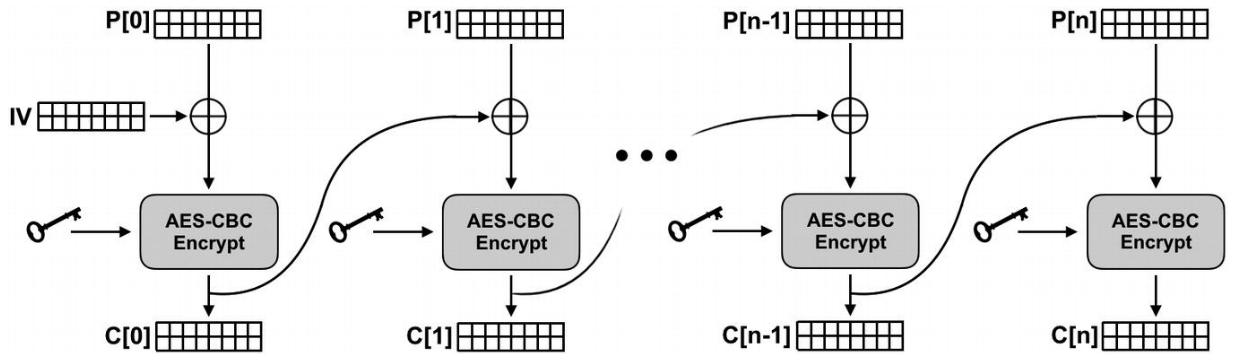


Figure 5-3 Visual depictions of CBC encryption

Figure 5-4 This image was encrypted using CBC mode. Much better!

More importantly, if the first block changes in any way, even a single bit, the ciphertext output for that block (i.e., C1) will also change. Because that output is XORed into the input of the next block, if C1 changes, the input to the second block changes too! That means that the change to the first block, even if the change is a single bit, will also completely change the output of the second block! And the output of that is fed into the input of the next block, and so on and so forth. This means that a change to any input block will change the output of that block and any subsequent blocks![3]

Cipher Block Chaining mode effectively eliminates all structure within data. Encrypting the image with this approach produces what appears to be static, as shown in Figure 5-4.

Now that the chaining part is explained, it is time to discuss the IV. The IV shown in Figure 5-3 is short for *initialization vector*. The IV is just random data that is mixed in with the very first block. Unlike the key, it does not have to be secret. The IV is used to make sure that the same message does not encrypt to the same output twice. Specifically, a different IV must be used for every encrypted message under the same key. By mixing in this random data, even if it is publicly known, it ensures that the data encrypted by the first block is not predictable. And this means that the ciphertext of the first AES block is also unpredictable, even if the message is a repeat from before.

Think what would happen if AES-CBC mode were used to encrypt two messages with the same key and IV. The outputs, even using the chaining of CBC, would be the same. Remember, AES with the same key always encrypts the same input to the same output. That does not change for CBC mode. Using a different IV, however, a bit of randomness is mixed in with each encryption operation, making the input, and therefore the output, unique.

For these reasons, the same key and IV *should never be reused* on two different plaintexts. I usually recommend that the same key never be reused just to be safe. But if the same key and IV pair is reused, duplicate plaintext would be recognizable as duplicates. Worse, however, reusing the same key and IV pair actually leaks information, and the information leaks can enable attackers to completely compromise the security. In fact, even though the IV does not need to be a secret, it *must* be unpredictable. If an attacker can predict the IV, then CBC-mode encryption can be compromised.

Just to be absolutely clear, when cryptographers talk about “two different plaintexts,” it has nothing to do with the contents of the message. It has to do with whether or not the algorithm has to “start over.” In the case of CBC, for example, any data being encrypted in the same “chain” is all part of one message no matter how the sender and receiver choose to split it up.

So, for example, someone sending secret messages might consider “attack at dawn!” and “cancel the attack” to be two different plaintexts. However, if they were sent as part of the same cipher block chain, then, from the perspective of CBC, they are just one plaintext. However, if a video file is split into two pieces and each one is encrypted separately, starting over each time from the first block of the CBC chain, then it is two plaintexts from the perspective of CBC even though the data is all part of one file.

Does this sound like a lot of very small, subtle rules upon which all of the security of the system depends? It does and that is why cryptography is very dangerous to play around with. You should be sensitive to the fact that cryptography can break down in many unexpected and (for a noncryptographer) unintuitive ways. I will return to this more at the end of the chapter in the discussion about You Are Not A Cryptographer (YANAC). For now, CBC has helped introduce just how cryptography *works* but also how it can *not work* if done incorrectly.

CBC is not the only mode of operation for block ciphers like AES. Although there are many others, I will not review them here. I will talk about a mode called *counter* mode in the next section on stream ciphers. And I will discuss some other modes of operation called *Authenticated Encryption with Additional Data* (AEAD) or alternatively *combined modes of operation*. AEAD algorithms perform both encryption and produce Message Authentication Code (MAC) at the same time. Because we have not talked about MACs yet, I will wait to discuss those until later.

It is worth noting, however, that these combined modes are almost universally considered better. For this, and other reasons, some modes like CBC are becoming outdated. They are still used, however, and they are very instructive about why these kinds of modes are needed and what types of problems they solve. So it is worthwhile to understand the principles behind CBC even as it becomes obsoleted.

Modern Stream Ciphers

As stated earlier in the chapter, symmetric encryption is typically broken down into block ciphers and stream ciphers. To repeat, block ciphers divide things up into chunks (i.e., “blocks”) of size *n* and then substitute the *n* bit plaintext for a specific *n* bit ciphertext (based on a chosen key). On the other hand, stream ciphers encrypt one symbol at a time where, for most modern ciphers, a symbol is a single bit (a one or a zero). Unlike a block cipher which always encrypts the same input to the same output (for a given key), a stream cipher almost always changes how it encrypts as it goes along. That is, how it encrypts a given symbol may depend on what it has encrypted before or how many symbols it has encrypted[4] [179]. You saw this in the Vigenere cipher where the encryption of a given symbol was determined by its index in the plaintext and the corresponding index in the repeating key.

As with most technologies in this book, it is impossible to cover stream ciphers comprehensively. However, the following tour will introduce you to both the core concepts as well as some commonly used stream ciphers.

One-Time Pad

The first stream cipher on our tour is called the *one-time pad* or OTP. OTP is an amazing way of encrypting things, and it is one of the ideas I had in mind when I said that cryptography was like magic. And unlike some of the long ins and outs I went through for block ciphers, it is (conceptually) simple to describe and understand.

How simple? Basically, OTP has two steps:

1. Create a *random* key that is the same size as the data to be encrypted (i.e., the plaintext).
2. XOR the random key and the data together.

That is it. That is the whole algorithm. No special modes of operation, no complicated block sizes. (It does use XOR, however; if you need to, go back to the earlier section in the chapter on XOR and review.)

But how good is it? Really good. It is one of the only algorithms known to be *information-theoretically secure*. This means that the encrypted message provides *no* information about the plaintext. An attacker could study the ciphertext forever, and there would be no possible way to crack it. In fact, OTP cannot even be attacked with brute force. The attacker can try to decrypt the ciphertext with every possible encryption key and *still* not know the correct plaintext [60, Appendix C], [66, Chapter 11], [40, Chapter 5].

Certain types of readers are probably waiting for the *but* that must surely be coming. After all, why waste half a chapter talking about AES and block ciphers if OTP is this good? What is the catch?

I already told you the catch. Take a moment and reread the two steps of the algorithm. As you look at those two steps, does anything strike you? What is the hard part of this?

The first challenge of this algorithm is in how the key is created. The word "random" is emphasized on purpose. For OTP to work, the key must be random *for each message*. In other words, for every single message meant to be encrypted by OTP, there must be a unique and randomly generated key for that message. In fact, if the same key is used more than once, the security completely breaks down.

Still, I already told you that CBC mode for block ciphers required a unique key and IV for each unique plaintext. So this is not that much different.

The real problem is that OTP requires the key *to be the same size as the plaintext*. Think about that for just a minute. Maybe this is no big deal if you are sending plaintext like "attack at dawn!" but what about a 25GB video file? To encrypt the video file with OTP, you would have to have a 25GB key! Even if it is possible to generate that much random data,[5] how will it be transmitted to the receiver? How will it be secured and protected?

Except for very high-security data (e.g., high-level diplomatic and security traffic), OTP is almost never used in practice [40, Chapter 5]. The key management problem is just not reasonable for most systems.

Nevertheless, OTP is a good starting point because in some ways it represents the ideal function, kind of like the pretend "magic" hash and encryption functions from the previous sections. OTP is ideal that the other stream ciphers try to emulate. If you can understand how OTP works and the basics behind *why* it works, the other (more practical) stream ciphers will be easier to grasp.

Fortunately, understanding the basics of OTP mostly requires understanding XOR. Recall that XOR has some really neat properties. Using XOR, data can be “hidden” inside other data and then easily recovered using the same operation.

In an OTP context, suppose we have our plaintext *P* and a key *K*, where both of them are the same size (i.e., have the same number of bits). We can produce a ciphertext *C* by XORing these two sequences together. And, we can recover the plaintext from the ciphertext by XORing the ciphertext with the same key. In other words

$$C = P \oplus K$$

$$P = C \oplus K$$

Again, do not let the math notation turn you off to this amazing concept. We can “hide” the plaintext *P* within *C* by mixing it with *K* using XOR. All of the data of *P* is within *C*, but it is hidden. But *P* can be recovered from *C* by XORing with *K* again. Thus, using this simple operation, data can be concealed and recovered quickly and easily.

If it helps, one way to think about XOR is that it is a bit flipping function. There are only two possibilities for each bit of plaintext input: zero or one. There are also only two possibilities for each bit of key: zero or one. Think of the bits of plaintext input as the starting information and of the bits of the key as *transformation* instructions that can transform the input into output. If the key bit is a zero, no transformation takes place. If you refer back to the truth table earlier in the chapter, you will see that any bit that is XORed with 0 remains the same ($0 \oplus 0 = 0$ and $1 \oplus 0 = 1$). On the other hand, if the key bit is one, the input bit is *flipped*. That is, any bit XORed with a 1 is inverted ($0 \oplus 1 = 1$ and $1 \oplus 1 = 0$).

Because the length of the key and the length of the plaintext are the same, the OTP key is a transformation instruction for each input bit. Thus, each input bit will either remain the same (i.e., if the corresponding key bit is

0) or it will be flipped (i.e., if the corresponding key bit is 1). The reason XORing with the same key restores the original output is because when the *ciphertext* is XORed with the same key, each bit will either still remain the same (i.e., it was not flipped when the plaintext was XORed, and it is not flipped when the ciphertext is XORed again) or it will flip back (i.e., it was flipped when the plaintext was XORed, and it is flipped back when the ciphertext is XORed again).

You might be wondering how this can possibly “hide” or encrypt the plaintext. After all, only *some* of the bits are flipping. Some of them are staying the same. In fact, if by some random chance the entire key was zeros, the plaintext would not change *at all*! Maybe that does not seem like very good encryption.

The reason OTP works is because one of the interesting properties of XOR is that you can get *any output you want* for an input. Because the key can be thought of as transformation instructions, there exists some set of instructions that can transform an input into a given output so long as the input, output, and key are all of the same length. If an original plaintext message is “attack at dawn” and this message has been encrypted with OTP and a key, there is *a different* key (of the same length) that would “decrypt” the message to “attack at noon” or “attack not now” or “go out to eat!” (note, all three messages are 14 characters; do not forget to include spaces and punctuation). So, if the key is truly chosen at random, it would not matter even if the key ended up being all zeros and the ciphertext was the same as the plaintext. The adversary would still not know if that was the right plaintext because *every plaintext of the same length is equally possible*.[6] This is also why brute force does not work against OTP. Not only is the key space far too large for messages of even small size, but even if the attacker somehow managed to find the right key, they would have no way of knowing.

Figure 5-5 An image with the text "Top Secret." This figure is identical to Figure 5-1

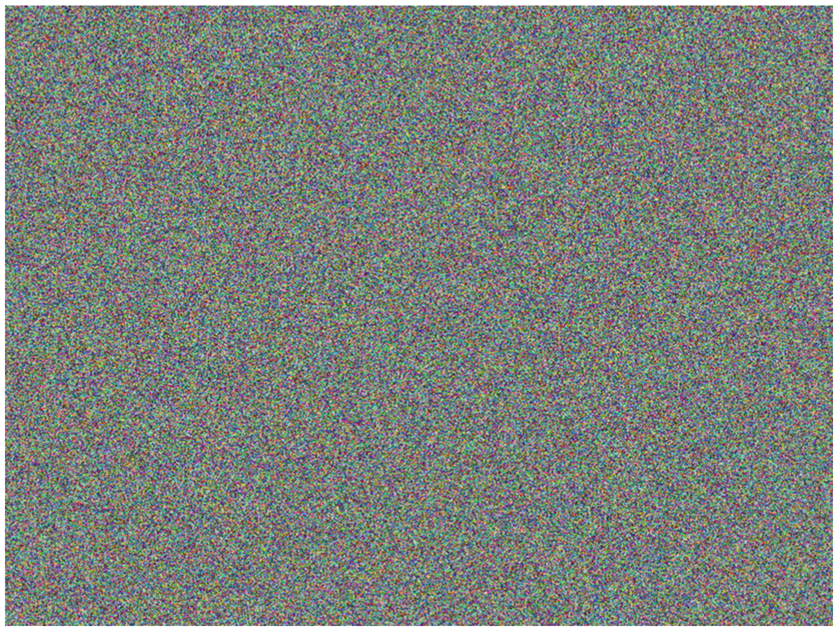


Figure 5-6 An image created with random data. Because there are no patterns or structure, it looks like static

To help illustrate this more visually, consider the image in Figure 5-5, which you have seen before. And now, look at the image in Figure 5-6.

This image is a visual representation of random data. Importantly, the size of the random data is *exactly* the size of the top secret image. What will happen if we XOR[7] these two images?

The output looks random too! This is the "magic trick" of XOR at work. The first image XORed with the random data completely "hides" the first image within the randomness as you can see in Figure 5-7. This is the basic concept of the one-time pad.

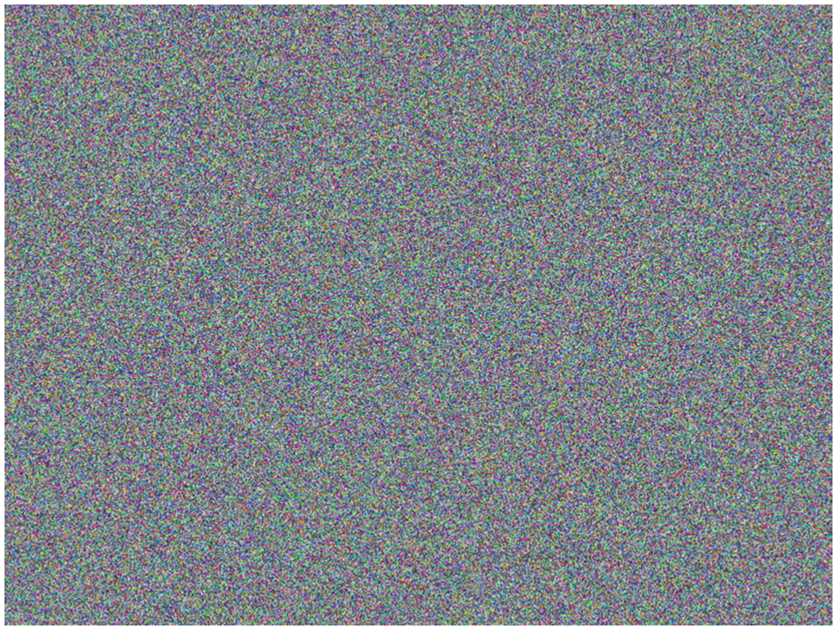


Figure 5-7 An image created from the XOR of Figures 5-5 and 5-6

But it is important to understand that the key *absolutely must be random*. If the key has any predictability, the attacker can use that to try (and will often succeed in) breaking the encryption.

There are a couple of other rules. As I stated before, the OTP key must *never* be reused. For one thing, unlike the block ciphers I talked about, if the attacker has the plaintext and the ciphertext of an OTP encryption, they immediately have the key. Just like you can combine the ciphertext and the key with XOR to recover the plaintext, you can combine the ciphertext *and the plaintext to recover the key*! So if a key were ever reused, an attacker in possession of a plaintext and ciphertext would recover the key and be able to use it to decrypt the second message as well.

Even if the attacker does not have a plaintext and ciphertext pair, if a key is reused there is a very weird thing that can happen *when you XOR the two ciphertexts together*. What follows is a bit complicated.

To see what happens, here is the math of XORing two OTP ciphertexts together that were encrypted with the same key:

$$\begin{aligned}
C1 &= P1 \oplus K \\
C2 &= P2 \oplus K \\
C1 \oplus C2 &= (P1 \oplus K) \oplus (P2 \oplus K) \\
&= (P1 \oplus P2) \oplus (K \oplus K) \\
&= P1 \oplus P2
\end{aligned}$$

Let's walk through what just happened. There is a first plaintext message called $P1$ and a second plaintext message called $P2$. Each one was encrypted with an OTP using the same key K. The ciphertexts ($C1$ and $C2$) are simply each plaintext message XORed with the key K. That part, at least, should make sense.

Next, the two ciphertexts are XORed together. Mathematically, this is the same as the first plaintext XORed with the key, XORed with the second plaintext XORed with the key. Hopefully, that substitution should also make sense.

To understand the next steps, you need to know that XOR as a mathematical operation can be moved around like addition (it has the commutative and associative properties). So the K values are regrouped together, and the two plaintext values are regrouped together.

The last thing that happens is the $K \oplus K$ is removed. Why? Because any number XORed with itself is all zeros. To see how this happens, remember that if a bit is XORed with 0, it remains the same, and if it is XORed with 1, it is inverted. So, if a number is XORed with itself, all of the 0 bits are XORed with 0 (thus staying 0), and all of the 1 bits are XORed with 1 (thus getting flipped to 0).

And because any number XORed with 0 is itself, the XOR of *K* with itself can be removed. All that is left is $P1 \oplus P2$.

You might be wondering what the big deal is with $P1 \oplus P2$. The attacker got neither plaintext. They only got the XOR of them. Does that not mean that the two plaintexts are mixed together? Are they not hidden within the XOR of the two?

The problem is that plaintexts are almost never random. Remember, if there is *predictability* in the key of an OTP operation, it can often be broken. By taking two ciphertexts (encrypted with the same OTP key) and XORing them together, you have replaced two plaintexts hidden in completely random data to two plaintexts “hidden” in predictable data. Attackers can use the predictability of plaintext data to find out information about one or both, sometimes completely deciphering the data altogether. This is not a theoretical exercise. Apparently during the Cold War, the Soviet Union reused key data for their OTP encryption, and American counterintelligence was able to decipher messages because of this mistake [40, Chapter 5].

There is one other very important “problem” with one-time pad encryption. OTP is great for *confidentiality*, but it provides *no data integrity*. What this means is that even if attackers cannot *read* the data, it does not mean they cannot *change* the data. I describe this as a problem in quotes because OTP does not try to provide data integrity. This is not a failing in any way of OTP, but it does mean that real messages protected with OTP would need additional protection mechanisms such as Message Authentication Codes, such as those described later in the chapter.

You might be surprised to learn that attackers can change messages they cannot read. To see how this works, remember that if an attacker knows both the ciphertext and the plaintext they can recover the OTP key. To repeat, the XOR of the ciphertext and the plaintext yields the key just like the XOR of the ciphertext and the key yields the plaintext. But if the attacker can *guess or predict* the plaintext, the result is the same. If an attacker can intercept an OTP-encrypted message and guess the plaintext, they can extract the key by XORing the ciphertext and plaintext together. They can then take the key and use it to OTP-encrypt a completely different message that they transmit in its place. The recipient will be none the wiser.

As an example, suppose that an attacker has somehow compromised the network between two banks and can intercept and alter any message that flows from one to the other. Assume that the two banks are using OTP encryption to protect their communications.

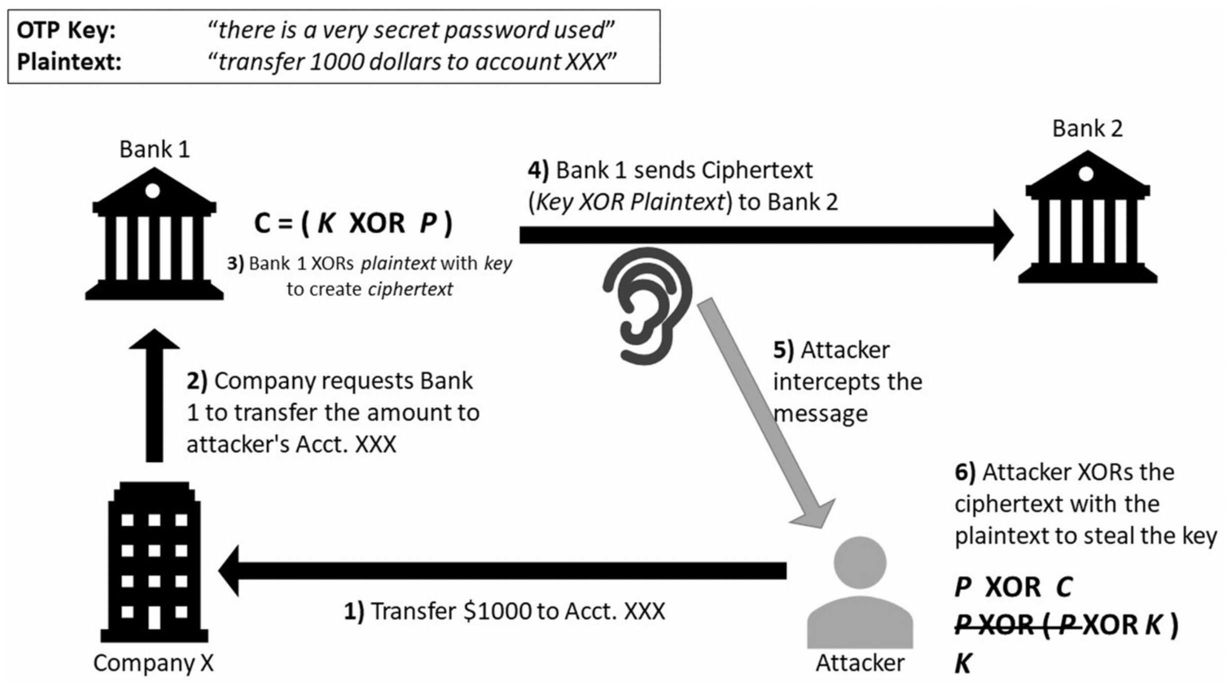

OTP Key: "there is a very secret password used"
Plaintext: "transfer 1000 dollars to account XXX"
Bank 1
Bank 2
C = (K XOR P)
3) Bank 1 XORs plaintext with key to create ciphertext
4) Bank 1 sends Ciphertext (Key XOR Plaintext) to Bank 2
2) Company requests Bank 1 to transfer the amount to attacker's Acct. XXX
5) Attacker intercepts the message
6) Attacker XORs the ciphertext with the plaintext to steal the key
P XOR C
~~P XOR (P~~ XOR K)
K
1) Transfer $1000 to Acct. XXX
Company X
Attacker

Figure 5-8 An MITM attacker is able to recover the OTP key by knowing the plaintext and ciphertext

The attacker finds some company that uses the first bank as its institution, and the attacker has an account at the second bank. Next, the attacker executes some kind of transaction with the victim company such that the company instructs the first bank to transmit 1000 dollars from the company's account at the first bank to the attacker's account at the second bank.

Assume that the attacker knows the format of the messages. For this example, pretend it is just a simple English phrase "transfer 1000 dollars to account XXX." When the victim company executes the transaction, the first bank encrypts this message and sends it over the network to the second bank.

But the attacker intercepts it. The attacker cannot read it, but they still know what the plaintext message is. By XORing the ciphertext and the plaintext, they are able to obtain the OTP key. Now the attacker creates a new message that reads "transfer 9999 dollars to account XXX" (notice that this message is the same length). They XOR the new message and the OTP key together and send it to the second bank. The new message decrypts just fine, and the second bank has no idea that it is forged. This hypothetical example is depicted in Figure 5-8.

The lesson to learn from this is that encryption is often used primarily, if not entirely, about confidentiality. In other words, encryption is primarily used to prevent an unauthorized party from *reading or understanding* the encrypted data. On the other hand, it generally does *not* provide *data integrity*, which prevents an unauthorized party from *altering* the data.

To deal with this problem, encryption must typically be combined with other techniques, such as Message Authentication Codes (MACs). Or, as I alluded to with block ciphers, this problem is also solved with AEAD algorithms that implement a “combined” mode of operation that performs both encryption and generates a MAC at the same time. Both of these topics are discussed later in the chapter.

AES Counter Mode

AES was covered in the section on block ciphers, and yet here it is in the section on stream ciphers too. Using a very neat trick (yes, this one feels like magic to me too), it is possible to convert a block cipher into a stream cipher. This trick is called *counter mode* (another mode of operation, just as CBC was a block cipher mode of operation).[8]

Before diving into AES counter mode (often abbreviated AES-CTR), I will point out that there are various types or categories of stream cipher. AES-CTR is an example of a *binary additive stream cipher*. This kind of cipher works in a way that is analogous to the one-time pad. There are two steps:

1. Generate a *key stream* that is the same size as the data

to be encrypted (i.e., the plaintext).

2. XOR the key stream and the data together.

Notice that the only difference between this type of cipher and the one-time pad is the use of a key *stream* instead of a key. In AES-CTR mode, and in many stream ciphers, the key stream is generated from an initial key and is a form of *key expansion*. Recall that in the Playfair cipher, the key material needed to be 25 characters to fill the five-by-five grid, but a code word could be *expanded* to fill it. Similarly, a relatively small key of 32 bytes can be expanded to an almost unlimited size.[9]

The main difference between any particular binary additive stream ciphers is how the key stream is generated. AES-CTR generates its key stream by, in fact, *counting*. A high-level description of the entire algorithm is

1. Choose a random key (128, 196, or 256 bits).
2. Choose a random starting number (128 bits) called the *nonce*.
3. Encrypt the nonce with AES using the key.
4. Add one to the nonce and encrypt that number with AES using the key.
5. Repeat this process until the desired amount of key stream is reached.

The word *nonce*, by the way, is a shortened version of “number used once.” The starting number of the counter is like the IV for CBC mode. It can be public, but it must be random and used only once.

Notice that AES is *not* encrypting *the plaintext*. This is a stream cipher where encrypting the plaintext is done by XORing the plaintext with the key stream. AES is being used here to only generate the key stream. And it does that by encrypting *n* (the nonce), $n+1$, $n+2$, and so forth until there is enough key stream to XOR with the plaintext. In fact, in AES-CTR mode, as in OTP, the AES decrypt operation is never used because the decryption of the plaintext is performed by XORing the ciphertext with the key stream.

Counter mode relies on the intersection of a couple of properties. First, because of how XOR works, when the attacker cannot split out the plaintext or key stream from the ciphertext so long as the attacker does not know (or cannot guess) either one of them. So, assuming the attacker does not know the plaintext already, they cannot extract the plaintext without knowing the key stream. The security of AES-CTR depends on the attacker not being able to know or guess the key stream.

The attacker, however, cannot know or guess the key stream so long as the key is random (not predictable) and so long as the key-nonce pair is not reused. The counter is not a secret. The fact that the attacker knows the counter numbers does not give them any clues about the key stream. Without the key, the attacker cannot know or predict what the ciphertext output of any encryption is. Therefore, the attacker cannot predict the key stream even if they know the counter numbers used to generate it.

By using a key stream instead of a key, AES-CTR mode is far more practical for real-world cryptography. A 32-byte key is far easier to transmit, store, and protect. The trade-off is that the key stream will not be truly random like the OTP key. Although it will be very cryptographically strong and resistant to analysis, it will have patterns that could be vulnerable. In practice, however, ciphers like AES-CTR are not compromised through this kind of analysis.

Another possible weakness is the brute-force analysis of the key. Unlike OTP, stream ciphers like AES-CTR can have their key uncovered by the attacker through brute force. But while this is theoretically possible, it is impossible in practice for a completely random key. It would require the same kind of billions of years described for breaking AES keys in the section on block ciphers.

So while AES-CTR mode and other ciphers like it are theoretically weaker than OTP, they are not weaker in practice. That could always change. For example, a researcher could uncover a vulnerability in AES tomorrow that would completely break it. Such a thing is not possible for OTP.

Where stream ciphers like AES-CTR break down is often in the very same ways that OTP breaks down, for example, reusing keys, not using really random keys, or using AES-CTR without some kind of data integrity protection.

ChaCha20

Another binary additive stream cipher is called *ChaCha20* by Daniel J. Bernstein. Unlike AES counter mode, ChaCha20 is not a block cipher adapted to be a stream cipher. It was designed to be a stream cipher from the start. I will not dive into the technical fundamentals of the algorithm here. Instead, I will discuss some of its features and properties that stand in contrast to AES and other algorithms. This will be a useful illustration of different motivations for choosing one cipher over another.

By way of background, Bernstein first created a very similar algorithm called Salsa20 in 2005 (first published in 2007). Conceptually, Bernstein created a *hash* function that hashes a key, a nonce, and a counter into an output block. To generate an additional block, the counter is incremented by one and a new hash generated. The key stream is generated by producing as many output blocks as necessary to have as many bits as the plaintext [57].

Story Time: Bernstein Fights to Decriminalize Cryptography

Depending on your age and your exposure to technology, you may be surprised to know that in the early 1990s, the United States considered encryption to be a *weapon*. The US government categorized encryption as a “munition,” which impacted the ability of individuals in the United States from exporting cryptographic ideas or source code above a certain strength. If you were a researcher and wanted to publish a paper about cryptography, under certain circumstances you would have to register as an *arms dealer*. This happened to a young mathematical genius named Daniel J. Bernstein, the same Daniel J. Bernstein who went on to create the ChaCha20 cipher.

Represented by the Electronic Frontier Foundation (EFF), Bernstein challenged the US government in court in 1995. The legal team successfully defended Bernstein and won concessions from the court that the cryptographic ideas he was formulating and proposing were forms of *speech*, protected by the Bill of Rights in the US Constitution.

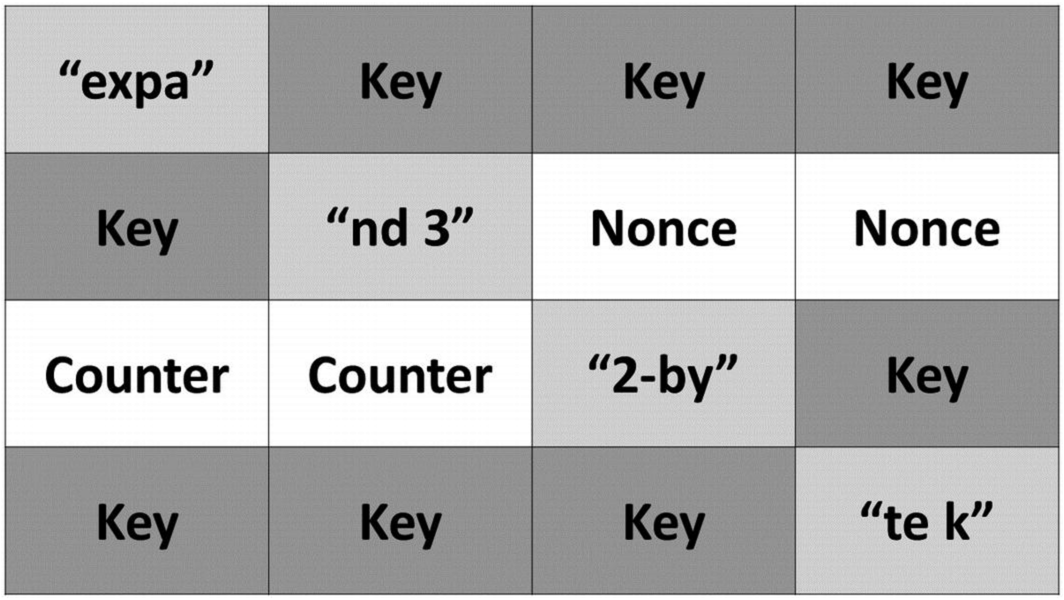
One of the judges in the case wrote:

> This court can find no meaningful difference between computer language, particularly high-level languages as defined above, and German or French....Like music and mathematical equations, computer language is just that, language, and it communicates information either to a computer or to those who can read it...

According to the EFF, this case helped to establish in law the principle that communicating to someone through writing a program is still communication. In other words, it is *speech* and entitled to all the legal protections afforded speech [89].

I emphasized that each block is generated by a hash function to draw attention to this fundamental difference between Salsa20 and AES-CTR. AES-CTR is based on the AES cipher that is *invertible*, while Salsa20 uses a one-way function that is not. Hopefully, it will be obvious by this point in the discussion that for counter mode, being able to invert is not necessary and, in fact, not used by AES-CTR. Recall that the AES-CTR mode does not encrypt and decrypt plaintext directly; rather, AES is used *in just one direction* to encrypt the counter in order to produce the key stream. In counter mode, AES's decryption operation is vestigial.

Because Salsa20 was designed to be a stream cipher from the beginning, there is no need for the block generation algorithm to be invertible.



"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Counter	Counter	"2-by"	Key
Key	Key	Key	"te k"

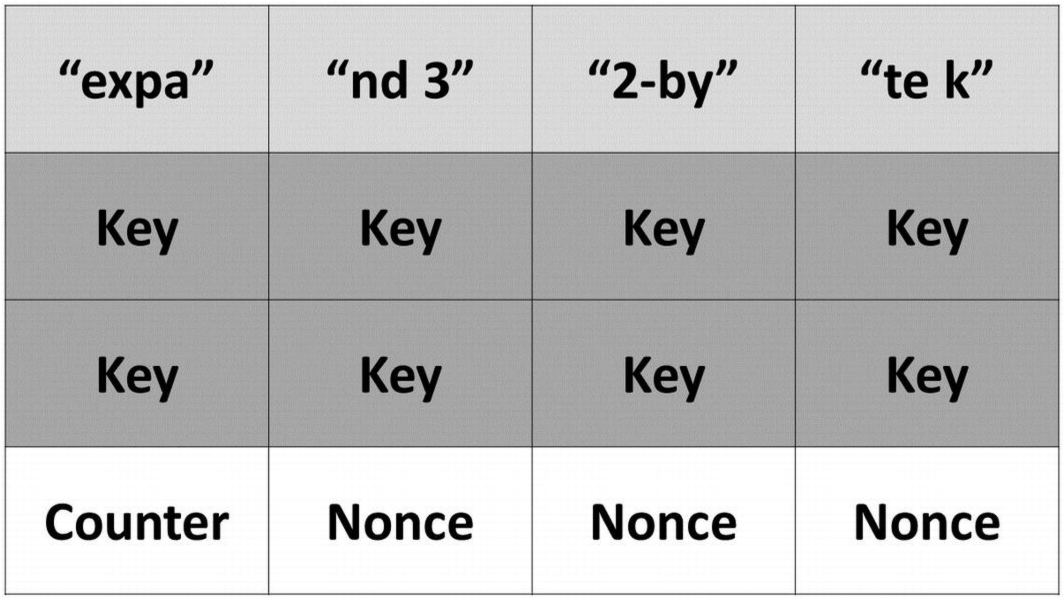
Figure 5-9 The initial configuration of the Salsa20 block

By default, the Salsa20 algorithm requires a 256-bit key and a 64-bit nonce. The counter is also 64 bits. An additional 128 bits of fixed data is also used to bring the total number of bits in the block to 512. In some ways, the construction of the block might seem reminiscent of creating the Playfair five-by-five block. As illustrated in Figure 5-9, the key, nonce, counter, and constant data are split up and spread throughout the 4-by-128-bit block. Once configured, the Salsa20 algorithm does a repeated series of mixing operations on the columns and rows. The mixing operations combine the various column and row values with each other in a way that causes input bits to influence a wide range of output bits. These mixing operations are organized into rounds, and the "20" in Salsa20 refers to having 20 rounds. By the end of the mixing, the output is unrecognizable from the input and appears to random data.

It should be noted that the 512-bit output is just the first in a potentially long sequence of key stream. The very next block will be generated with an input that is exactly the same except that one or two bits will have flipped as part of incrementing the counter value. The hashing component of Salsa20 has, however, the Avalanche property, and changing just one bit of input results in a completely different output.

You may have also noticed from Figure 5-9 that the constant values that are inserted into the diagonal of the input block is human-readable text. In fact, combined together, these values spell out "expand 32-byte k". This is an example of what some cryptographers call the "nothing up my sleeve" principle. The constant value fed into the input is not *supposed* to matter. It is just supposed to be some arbitrary bit of data to expand things out to the full 512 bits. But how do you know if you can trust the creator of an algorithm to not have inserted some weird vulnerability?

In security, a secret or hidden way of cracking open some protection system is often called a “back door.” Typically, they are inserted or created by an authority figure or the creator such that there is an alternative way into the system. While some governments have more or less demanded that all computer security systems have a back door, this has, so far, been resisted by the security community. Because of these kinds of intentional attempts to build weaknesses into systems, many cryptographers and security engineers are suspicious of hard-coded numbers or configuration values that have no explanation. Values like “expand 32-byte k” are very unlikely to be any kind of special mathematical value and prove that there is “nothing up my sleeve.”



“expa”	“nd 3”	“2-by”	“te k”
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

Figure 5-10 The initial configuration of the ChaCha20 block

A few years after creating Salsa20, Bernstein introduced some variations to create the ChaCha20 algorithm. This algorithm has a very similar setup and operation with a few changes. For example, the initial configuration of the 512-bit block is shown in Figure 5-10. As you can see, some elements are moved around, and a few others have been slightly altered.

In addition to these changes to the input, the mixing operations were also modified to improve certain cryptographic features beyond the scope of this book. One change you probably noticed that is *not* important is the change to the counter. In Figure 5-9, there were 64 bits of nonce and 64 bits of counter. In Figure 5-10, there are 96 bits of nonce and 32 bits of counter. In Bernstein's original paper, there was no difference. ChaCha20 also had 64 bits of nonce and 64 bits of counter. The version shown in this figure is how ChaCha20 is specified in the open standard known as RFC 7539. This standard acknowledges the minor change and did so to conform with other similar ciphers [194]. This change does reduce the maximum amount of key stream that can be produced. It is possible to use the 64-bit counter if a longer key stream is needed.

To summarize the differences with AES-CTR so far, ChaCha20 (and its predecessor, Salsa20) is used exclusively for stream ciphers. Whereas AES's block operation is invertible (encrypt and decrypt), ChaCha20's operation is more like a hash (one way only). Also, ChaCha20 was explicitly designed to be transparent with no secret or hidden values.

There are a few other important differences. ChaCha20 (and Salsa20) uses very basic and simple computer operations. This makes ChaCha20 very fast. While ChaCha20 cannot go faster than AES for certain modern processors with built-in AES support, it is much faster than AES on computers without AES speedups. This is important for low power and IoT devices. The nature of ChaCha20's design also makes it less vulnerable to certain classes of attacks called *timing attacks*. These kinds of attacks figure out how to break ciphers by measuring how long it takes to do something. ChaCha20 is more resistant to this than AES [192].

Finally, some people want ChaCha20 available if for no other reason than having another option to AES. This is basically following the old proverb to not have all your eggs in one basket. If the world's communications systems only used AES (and for a while, that really was the only option for a lot of Internet traffic other than older less secure algorithms), then if a vulnerability is discovered in AES the result will be that everything is broken and there are no options to switch to. By having ChaCha20 as a standard that is widely deployed if something bad happens to AES, there is something else to use as an alternative.

Message Authentication Codes and Combined Modes of Operation

Finally, before moving on to asymmetric cryptography, it is time to talk about Message Authentication Codes or MACs. As alluded to several times in this chapter, encryption is designed for *confidentiality*, not *data integrity*. Now, however, MACs can help with data integrity by enabling *data origin authentication*.

As a concept, a Message Authentication Code is somewhat similar to a hash and, as we will see, can be implemented using hashing. Like a hash, the Message Authentication Code is usually a fixed-size cryptographic code that is a kind of fingerprint on some designated data. The MAC makes it possible to tell if the data has been changed. If it has, the MAC will not be verified.

What makes a MAC different from a hash is that it must be *keyed*. Hashes require no keys. And the output of a hash for a given input is always the same. This makes it impossible for the raw hash to be a MAC because an attacker that can change the data can also change the hash to match.

On the other hand, a MAC involves a secret key known only to the authorized parties. A correct MAC can only be generated (or verified) with possession of the secret key. So, if an attacker intercepts and alters the data, they are unable to generate a matching MAC because they are missing the key required to do so. Thus, when an authorized recipient receives the data and the MAC, they can use their copy of the secret key to verify that the MAC matches the data. A valid result tells them two things:

1. The MAC was generated by an authorized party (assuming only authorized parties have the key).
2. The data protected by the MAC has not been altered.

Together, these two properties provide the data origin authentication. If the validity check fails, either the author is not legitimate or someone else has altered the data.

In case the comparison to the hash function is not clear, here is how a MAC is generated. The author or sender of some data inputs both the data and the secret key into a MAC function. This produces a MAC code. The data and the MAC code are sent together (or must otherwise be made available to the recipient together). Once the recipient has both the data and the MAC code, the recipient inserts the data, the MAC, and the secret key into a verification function. This function will return a true or false value depending on whether or not the MAC code can be verified for the data.

There are a number of different ways of generating a MAC code. As mentioned, there is a Hash-Based Message Authentication Code conveniently called HMAC. HMAC combines a hashing algorithm with a key. The intuition behind this is pretty simple. Imagine if someone wanted to send the message “attack at dawn” and wants to ensure that it cannot be altered. Let us also imagine that the sender and the recipient share a secret key in the form of a human-readable four-word password: “JawColdFilmZilch”.

To send an overly simplified hash-based password, the sender could simply *prepend* the password to the message: “JawColdFilmZilchattack at dawn”. The hash of this message would serve as a rudimentary MAC. This MAC code would be sent with the data to the recipient.

When the recipient received the data (“attack at dawn”) and the corresponding MAC, they would generate their own hash on the data prepended with the password: “JawColdFilmZilchattack at dawn”. The two hashes would match, and the recipient would have some assurance that the message came from an authorized party and had not been altered.

Had the message been intercepted, an attacker could have altered the message (e.g., “throw down your arms and hail your new overlords!”), but without the password it would be impossible to generate the correct MAC.

Please note, however, that this is the overly simplified version. HMAC is actually a more complicated algorithm that involves two hashes, some padding, and some XOR operations. But conceptually, it is still hashing data mixed in with a key. To be clear, the overly simplified version in the preceding example is not secure and should not be used as a real MAC. Its sole purpose is to help provide a little bit of intuition about how HMAC works.

Perhaps it also bears pointing out that in HMAC, the key does not have to be a human-readable password. That was also just for simplicity and convenience in the example. HMAC can be used on any kind of arbitrary data (videos, pictures, documents, text, etc.), and the keys are generally just random binary data or binary data derived from a password.

I included HMAC in the section on symmetric ciphers because the process is symmetric. The same key generates the MAC as is used to verify it. At the same time, however, ciphers that can perform encryption can also be used for creating MACs. Recall the CBC mode described earlier for AES encryption. Each of the blocks of data was fed into the next block of data to influence its encrypted output. The CBC-MAC algorithm uses CBC on the data but throws away everything except the last block. By convention, the IV is set to all zeros. In contrast to encryption, when the same message should *not* encrypt to the same output every time, a MAC should be the same for the same message with the same key. Fixing the IV to zero is perfectly fine for this MAC algorithm even though it would be a terrible idea for encryption.

One word of warning: If CBC is used for both encryption and the MAC, different keys must be used for each one.

Speaking of which, I have talked about using MAC for integrity and encryption for confidentiality. What if you need both? In the preceding example with the transmitted message "attack at dawn," there was no encryption applied. Even if an attacker could not change the message, it might not be optimal to have them read it either.

Of course, one can apply both encryption and MAC to the data. But which should be done first? One approach is MAC-Then-Encrypt. In this version, the MAC is applied to the *plaintext*, and then both the plaintext and the MAC are encrypted together. Alternatively, there is also Encrypt-Then-MAC. To take this approach, the MAC is again computed over the ciphertext, but the MAC itself is not encrypted. It is sent (unencrypted) along with the ciphertext.

Is one better than the other? As it happens, it is strongly recommended by most cryptographers to use Encrypt-Then-MAC. The general reason for this is the attackers should not be able to undetectably alter the ciphertext, not just the plaintext. This can be unintuitive because the end goal is protecting the *plaintext*. But it turns out that bad things happen when the attackers can undetectably change the ciphertext without us being able to detect it. The Encrypt-Then-MAC approach should protect the ciphertext against modification.

If all the various rules for MAC seem complicated and really easy to do wrong, you are correct. One of the real challenges of cryptography is that there are so many ways to do it wrong. When it is wrong, it is not always obvious. While this challenge will probably never be fully eliminated, one approach to solving the problem is to develop algorithms and techniques that are harder to do wrong and easier to get right.

One such approach are modes of operation called “combined” modes of operations. A combined mode of operation is a mode for symmetric ciphers (either block or stream) that natively include a MAC generation as part of the operation. The more technical term for these modes is AEAD (Authenticated Encryption with Additional Data). To reiterate, these new modes of operation provide *both* confidentiality and authenticity for the plaintext. AEAD can also provide authenticity over data that does not need to be encrypted (“additional data”). This is more important than it might sound. There are many times when certain public pieces of information need to be tied together with the encrypted data as well. Both the encrypted data and the associated unencrypted data must be protected and protected together.

Basically, AEAD converts an encryption operation from a two-input, one-output function into a three-input, two-output function. Normally, the encryption function is simply

```
 encrypt(key, plaintext) = ciphertext
```

With AEAD, it becomes

```
encrypt(key, plaintext, additional_data) = ciphertext, tag
```

To repeat, the encrypt function does *not* encrypt the additional data. The additional data is not included in the ciphertext. But the *tag* value is computed as a MAC on both the ciphertext and the additional data. For various reasons, AEAD functions use the term “tag” instead of MAC, but it is more or less equivalent.

The decrypt function is similarly changed, whereas the original decryption function looks like this:

```
decrypt(key, ciphertext) = plaintext
```

With AEAD, it becomes

```
decrypt(key, ciphertext, additional_data, tag) = plaintext, validity
```

These function definitions are conceptual, of course. For example, some implementations may not produce *any* plaintext until the tag is verified. After all, the ciphertext need not be decrypted first because the tag is computed on the ciphertext, not the plaintext. But in many practical applications, this is impossible. Often, the ciphertext will be received by a recipient in parts, and it is considered inefficient to not start decrypting while awaiting the remaining ciphertext. In this case, the tag cannot yet be verified because the verification can only be performed when all of the ciphertext has been received.

In any event, the concept is the same. The plaintext is not to be trusted until the tag is verified.

Some of the common AEAD ciphers in use today are AES-GCM, AES-CCM, and ChaCha20-Poly1305. AES-GCM and AES-CCM both use AES counter mode internally for the generation of the ciphertext. AES-GCM uses Galois Message Authentication Code (GMAC) to generate the tag.

AES-CCM, on the other hand, uses the CBC-MAC algorithm described in the previous section. As you can probably guess, ChaCha20-Poly1305 uses ChaCha20 to perform the encryption. The MAC algorithm known as Poly1305 was also created by Daniel J. Bernstein. Even though all three of these algorithms can be described as two individual components for the encryption and the tagging, the combined mode of operation describes how the two are to be performed together so as to be integrated.

As a final note, AEAD algorithms are relatively newer. Nevertheless, AEAD is already used in many systems and generally recommended for new systems going forward [115].

Asymmetric Cryptography

Switching gears, the rest of this chapter focuses on asymmetric cryptography. This technology is one of the most important advances in cryptographic security ever made. It is used in many applications, not the least of which is the security protocols used by almost every major website on the Internet today. Another name this technology goes by is "public key cryptography."

Asymmetric cryptography is about systems that involve *two* keys. All of our examples in symmetric cryptography involve just one key for both sides of an operation: encryption/decryption and MAC generation/verification. But in the operations that follow, there is always a public key that can be given to everyone and a private key that should never be disclosed to anyone. This is very powerful.

Consider, for example, the MAC operations from the previous section. A recipient can validate a MAC with the shared key. But this key can also be used to *generate* MACs. If the recipient is untrustworthy, the recipient could generate fake messages and create MACs for them. It would be impossible to tell which party created the fake messages because both of them have the key. This also prevents the sender from being able to authenticate a message to a wide group of participants.

On the other hand, asymmetric operations mean that a public key can be widely distributed without any risk to the private key. This enables some really neat operations. Three of the most common asymmetric operations are encryption, signatures, and key agreement. Notably, not every asymmetric algorithm can perform every asymmetric operation (e.g., not every asymmetric algorithm can do both encryption and signatures).

This book is not meant to catalog all possible algorithms and all possible operations. Instead, you will learn about some common approaches to asymmetric encryption, signatures, and key agreement as an introduction to the technology.

Asymmetric Encryption

Unlike symmetric ciphers, where there are many algorithms that perform encryption, most asymmetric algorithms do not. In fact, asymmetric encryption is not widely used and becoming less common as time goes on. One widely used asymmetric algorithm that does do encryption is *RSA*.

RSA, named for its three authors Rivest, Shamir, and Adleman, is one of the earliest asymmetric algorithms. Although it is aging and many systems are moving to newer algorithms, it is still very widely used. Perhaps more importantly for the purposes of this section, it provides some easy-to-understand operations.

Even still, RSA encryption is not used for what is called “bulk” encryption. Bulk encryption is the encryption of arbitrarily large amounts of data such as a large file or network communication traffic. RSA cannot do this because

RSA's encryption operation can only encrypt relatively small messages. And while a mode of operation like CBC could, in theory, make it possible to encrypt larger amounts of data, RSA is just too slow to be used for that purpose. RSA encryption is slow. Really, really slow.

Instead, RSA encryption is primarily used for two purposes: authenticated transport of secrets and digital signatures. There are other ways of doing digital signatures, so I will cover how RSA uses encryption for digital signatures in the next section. For now, I will talk about how asymmetric encryption allows two parties to share a secret without having a shared key.

The first concept to understand is that the public key and private key of RSA both can encrypt data. What is important is that data encrypted by one *can only be decrypted by the other*. Data encrypted by the public key can only be decrypted by the private key and vice versa. In the section on digital signatures, RSA will use encryption with the private key.

For sharing secrets, however, data is encrypted with the *public* key. Remember that a public key can be shared with everyone. If someone with the public key encrypts some data, only the party with the private key can decrypt it. *Assuming* that only one party is in possession of the private key, then only that one party can decrypt it.

The purpose of this kind of algorithm is a kind of "secure drop box." If you have ever dropped off books at a library, this is basically the same operation. You can drop the books into the slot at the library, but you (nor anyone other than the library staff) do not have access to it afterward. Similarly, anyone with the public key can encrypt a message with the public key, but thereafter only the party with the private key can retrieve the data. So, if you generated an RSA public and private key, you could give the public key to the whole world (e.g., publish it on your website), and anyone in the world could send you a message that only you could decrypt (i.e., using your private key).

Story Time: Better Late Than Never
It turns out that Rivest, Shamir, and Adleman were not the first to come up with the mathematics behind what we now know as RSA. A British team did it first. Their names were Clifford Cocks, James Ellis, and Malcolm Williamson. They figured out their formulation of the problem in the early 1970s.

Unfortunately for these three individuals, they worked for the British Government Communications Headquarters (GCHQ). In the worst of all possible worlds, the GCHQ not only could not find a use for their ideas but also classified it, preventing them from publishing about it. Their work was not declassified until 1997.

Despite the delay in recognition, in October of 2010, the IEEE (Institute of Electrical and Electronics Engineers) presented its 100th milestone award to the three Britons (James Ellis had already passed away, but his widow was able to attend on his behalf). Although somewhat late, it was cathartic for pioneers in technology to be recognized for their ideas that are ubiquitous now. Even though they were not able to publish about it at the time, they appear to be the first people to have figured out the math that every one of us depends on every single day for protected Internet communications [209].

As stated, however, RSA can only encrypt small messages and is very, very slow. It would be more or less impossible to exchange data of any real size. So what good is it?

Conveniently, a *symmetric* algorithm is very fast for bulk encryption, and a symmetric key is a very small amount of data. The most common way RSA encryption of this form is used is for an initiator to transmit a *symmetric key* to the private key owner and then switch over to symmetric encryption and MACs as well. This is sometimes called *hybrid* cryptography [66, Chapter 2].

To walk through this in a little more detail, I will name our fictitious parties “Alice” and “Bob.” This is commonly done in cryptography to talk about “party A” and “party B.”

The power of asymmetric cryptography is that Alice and Bob can start to communicate over a secured channel without ever having shared a key together in the past. In all of the symmetric examples in the first part of this chapter, it was just assumed that somehow the sender and recipient of messages shared a key. But how did they get that key in the first place? Did they meet in person? How was the key transmitted to them securely? Did they make sure they got the right key? Did they get the key from an authorized individual? All of those issues were ignored.

Now, armed with asymmetric cryptography, Alice and Bob are going to do what is called “key transport.” For this example, which is also visualized in Figure 5-11 I will assume that Alice is the party with the private and public key pair. Alice has published her public key to all the world through some means. Bob wants to communicate with Alice even though they have never met and never previously shared a key.

Bob starts by generating a new symmetric key. As described in the previous sections of the chapter, symmetric keys are usually just random data. This is relatively easy and fast to generate. Because Bob is generating this key on the fly, and it will only be used for this particular communication with Alice, it is often called a “session key” and an “ephemeral key.” It is a session key because it is only for this session, and it is an ephemeral key because it was created “out of thin air” and will be discarded after use.

After generating this symmetric key (say for AES encryption), Bob takes Alice’s public key and encrypts the symmetric key. Bob sends this encrypted message to Alice along with some other data such as an introduction and configuration information. Once Alice receives the encrypted message, she (and only she) can decrypt it with the private key. This decrypts the session key. Now, both Alice and Bob have the same shared key.

Alice and Bob now begin to exchange messages encrypted and MACed using the symmetric key (it is often the case that different keys are needed for encryption and MAC, but there are algorithms for expanding a single key into multiple keys for exactly this purpose). Because the messages have MACs, both sides know that only the parties with the symmetric key could be sending the messages. Bob knows that only Alice could have decrypted the message with the symmetric key, so only Alice could be sending messages MACed with this key. Thus, Bob is assured he is speaking with Alice and that the communications from Alice have not been tampered with.

For her part, Alice does not know Bob’s identity. But she does know that the communications are all tied to the same party. If Alice is actually a website (remember, Alice and Bob are really just placeholders for party A and party B), Bob might be signing up with a username and a password. Bob, in this case, is establishing his identity. But he wants to be sure of the website before he does so.

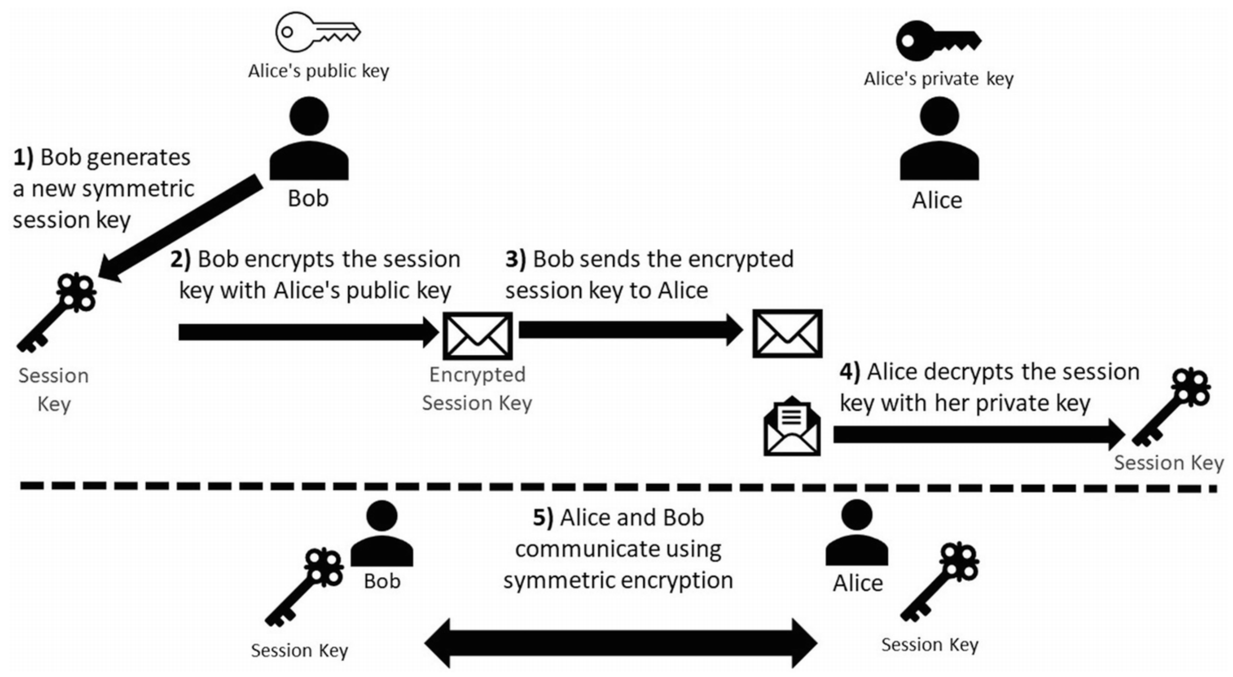

Alice's public key
Alice's private key
1) Bob generates a new symmetric session key
Bob
Alice
2) Bob encrypts the session key with Alice's public key
3) Bob sends the encrypted session key to Alice
Session Key
Encrypted Session Key
4) Alice decrypts the session key with her private key
Session Key
5) Alice and Bob communicate using symmetric encryption
Bob
Alice
Session Key
Session Key

Figure 5-11 Key transport between Alice and Bob using asymmetric cryptography

Alice can also verify Bob's identity if Bob has his own private and public key pair. Alice can transmit her own messages to Bob encrypted by Bob's public key. Data can either be combined such that the key shared between them was equally created by both of them, or data could even be split into two channels (i.e., a channel from Bob to Alice and another channel from Alice to Bob). There are many ways these systems can be constructed.

With that said, the standard warnings always apply. What is described here is a simplification for instruction purposes. In practice, there are a number of ways this can go wrong, and practical systems, which are the subject of the next chapter, have had to go through a lot of trial and error to get things right. Always remember to get a cryptography subject-matter expert if you are in an environment where cryptography is needed.

Two additional comments before moving on. First, unlike symmetric keys, RSA keys are not just random bits. They are mathematical numbers that have certain properties that enable the RSA operations to work. RSA keys are also much larger than symmetric keys. The very smallest size is 1024 bits, and these are considered obsolete and insecure. The minimum size for RSA keys now is 2048 bits. RSA keys are also *slow* to generate. Usually, this is not a problem because RSA keys are meant to be used *long term*. They are *not* ephemeral keys and are intended to be used and reused. This will become important in the section on key agreement.

The other comment is that our example just assumes that Bob gets the correct public key for Alice. This is harder than it sounds. How does Bob know he got the right one? What if an attacker manages to get Bob a different public key and convince him that it is Alice's? I will address this subject in a later chapter, but the preview is that this is why a "public key infrastructure" (PKI) is necessary.

Digital Signatures

RSA encryption can also be used for digital signatures. There are other methods for performing a signature, but I will start with RSA to follow up on the previous section.

In introducing RSA, I pointed out that it is possible to encrypt with the *private* key and decrypt with the public key. Recall that for key transport, Bob encrypted the session key using Alice's public key. This version of things goes the other way around.

A good question to ask yourself here is *why* would anyone want to encrypt with the private key? After all, *everyone* (potentially) has the public key, so anyone can decrypt it. That is true. If this were encryption for the purpose of confidentiality, it would be a terrible system.

But this encryption is not used for confidentiality. It is used for *data origin authentication*. The goal is to prove authorship and that data remains unchanged. In other words, the encrypted data is *intended* to be decrypted by everyone else to prove to them the source and correctness of the data. This is the concept of a *digital signature*.

Using the beloved Alice and Bob duo again, I will walk through a digital signature example that is also illustrated in Figure 5-12. This example will start with Alice holding the private key. As before, she has also published her public key to the whole world. Bob, for example, is in possession of the public key and knows (believes?) it belongs to Alice.

Alice would like to publish data and *prove* to the world that she is the author and that the data she wrote has not been altered. To create an RSA signature, Alice starts by hashing the data she has authored. Next, Alice encrypts the hash of her data with the RSA private key. The RSA-encrypted hash of her data is the RSA digital signature.

Alice now publishes to the world the data she has authored along with the digital signature. Bob, for example, can take the data and validate it. For this example, I will assume it is unencrypted. To verify the published data, Bob takes the data and hashes it himself. The hashing algorithm must be known, of course, but anyone can generate an unkeyed hash. Now Bob uses Alice’s public key to decrypt the encrypted hash. Bob compares the decrypted hash with the hash he generated himself. If they are identical, the signature is validated, and Bob is assured that Alice is the author and that the data has not been modified.

Alice must be the author because only Alice has the private key. Therefore, only Alice could have encrypted the hash of the data. The data must not have been altered because Bob generated his own hash and found it to be identical to the decrypted hash. An attacker could not have generated a fake signature because it would have required the private key.

RSA signatures were the most common way of signing data for a relatively long time (in computer years, anyway) that some sources began to describe all digital signatures as an “encrypted hash.” This is not true because, as stated, most asymmetric algorithms do not even have an encryption operation. For example, algorithms like Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) do not have encryption algorithms.

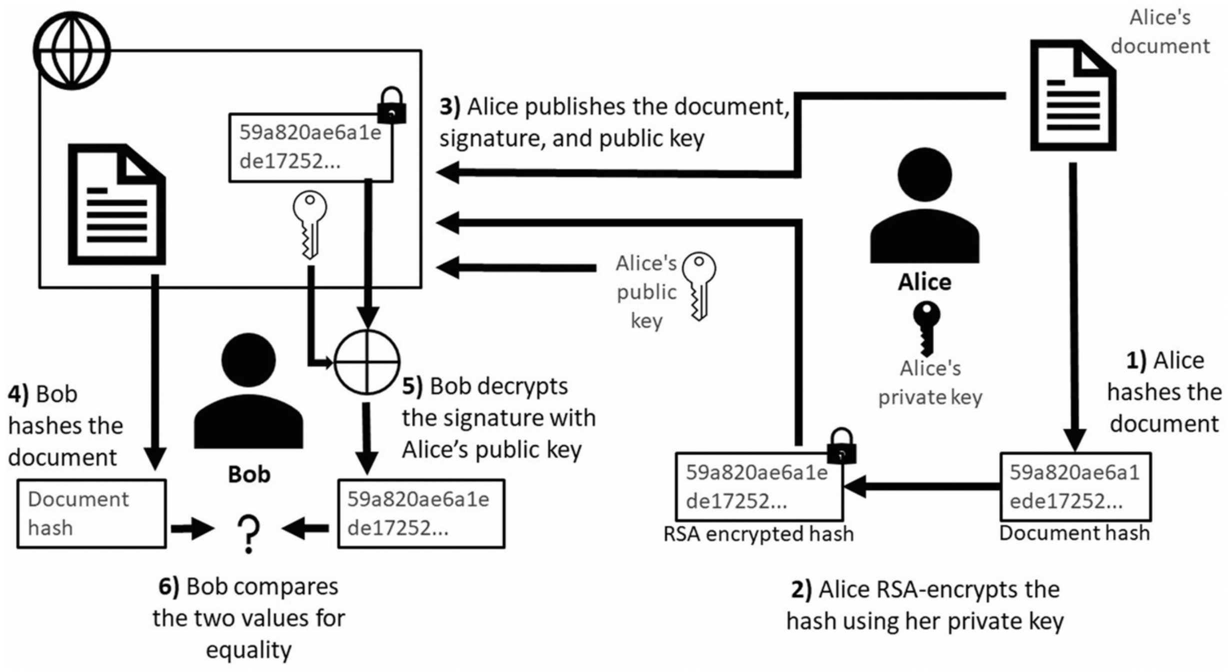

Alice's document
3) Alice publishes the document, signature, and public key
59a820ae6a1e de17252...
Alice's public key
Alice
Alice's private key
1) Alice hashes the document
4) Bob hashes the document
5) Bob decrypts the signature with Alice's public key
Bob
Document hash
?
59a820ae6a1e de17252...
59a820ae6a1e de17252...
RSA encrypted hash
59a820ae6a1 ede17252...
Document hash
6) Bob compares the two values for equality
2) Alice RSA-encrypts the hash using her private key

Figure 5-12 Alice’s digital signature proves that Alice authored the document

Like all asymmetric algorithms, both DSA and ECDSA have private and public key pairs. Like RSA, these are not random numbers but special mathematical pairs. For both algorithms, the signature is a *pair* of numbers unimaginatively labeled *r* and *s*. The *s* value is calculated from the data to be signed and the private key. The verification algorithm involves a mathematical algorithm applied to the data, the *s* and *r* values, and the public key. If the output of this function is equal to *r*, the signature is verified. The mathematics are too complicated for any real treatment, but I emphasize the *r* and *s* values simply to illustrate how the outputs are different than RSA.

It is worth pointing out, however, that both RSA and ECDSA also use the hash of the data rather than the data itself. Pretty much all signature schemes will use hashing or something like it in order to limit the asymmetric calculations on the data. It is much easier to do these mathematical operations on 64 bytes of data rather than gigabytes of data.

Key Agreement

Our final asymmetric operation to discuss in this chapter is called *key agreement*. Key agreement, like the key transport discussed earlier, is a way of enabling two parties to establish a shared secret (i.e., a shared symmetric key).[10] But whereas key transport actually involves encrypting a secret and transmitting it for decryption, key agreement enables the two parties to *simultaneously create the same key at the same time* without transmitting any part of it! Like I said, cryptography is like magic!

There are two algorithms widely used for key agreement: Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH). Both algorithms work along similar lines. For the rest of the chapter, I will just refer to DH unless I need to identify ECDH specifically.

The basic idea is this. Alice and Bob would like to generate a symmetric key that they will use for bulk transport. To do this, Alice and Bob first agree on some mathematical parameters for this operation. This information can be public, and there are no issues exchanging this information in the open. From these parameters, they both generate a DH public and private key pair. In particular, the public key is derived from the private key in combination with the public parameters.

Next, Alice and Bob exchange their DH public keys. At this point, Alice combines her private key with Bob's public key. At the same time, Bob combines his private key with Alice's public key. Because of some cool math properties of the calculations, Alice and Bob both compute the same number from these two different operations. Or, in other words, Alice's private key mixed with Bob's public key results in the same number as Bob's private key mixed with Alice's public key. So they get the same secret number (that can then be used to create a symmetric key), while an attacker is unable to generate it. Remember, the only thing exchanged in the open are some public parameters and the public keys! Without at least one of the private keys, an attacker gets nothing.

A nonmathematical explanation in A. J. Han Vinck's course "Introduction to Public Key Cryptography" [277] is depicted in Figure 5-13. In this figure, the "private key" is a secret paint color. Alice and Bob each have their own secret color. The public parameters are represented by a public paint color. Alice and Bob can mix their secret colors with the public color to create a new color for each of them. This new color is their respective public keys.

Alice and Bob can now exchange the public key. Because it is their secret color mixed with the public color, an attacker cannot get their original, secret color (this example presumes that it is difficult, if not impossible, to extract the original color from the mixed paints). But when Alice and Bob receive the other side's public key, they now have all three colors. Alice has her secret color, plus Bob's color mixed with the public color ($A + (B + P)$). Bob, on the other hand, has his secret color plus Alice's secret color mixed with the public color ($B + (A + P)$). Both Alice and Bob can now produce the exact same color ($A + B + P$). But an attacker, who has only ever seen the public colors $B + P$ and $A + P$, cannot correctly get the combined color.

Figure 5-14 shows the more realistic Diffie-Hellman key agreement.

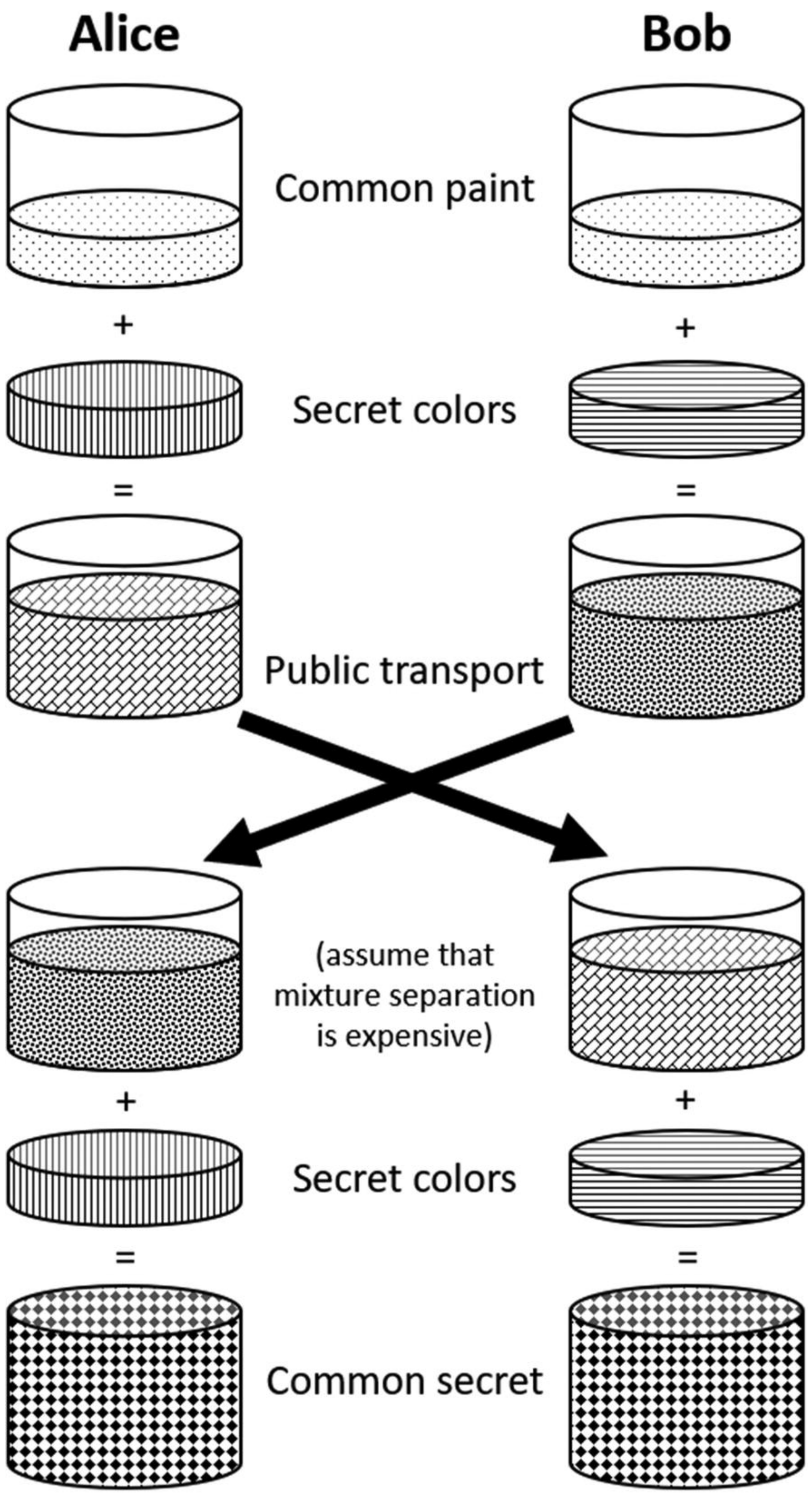


Figure 5-13 Intuition behind Diffie-Hellman

There are multiple reasons why Diffie-Hellman is preferred to RSA key transport. For one thing, in DH, both sides contribute to the key, whereas in RSA key transport, the key is completely generated by one side. This can be fixed for RSA using various mechanisms, but all of these fixes add to the time and complexity of the algorithm.

The much bigger reason is the DH can be done with ephemeral keys, just like the session keys it creates. Instead of using the same DH key pair over and over, Alice and Bob can generate *new* DH keys for every single key exchange. This is very valuable because it enables *forward secrecy*. The concept of forward secrecy, also called *perfect forward secrecy*, is that even if an attacker breaks *one* session, they should not be able to break *all sessions*. Or, alternatively, if there is a breakdown in security for a single key, it should, at most, result in the loss of confidentiality for a single session.

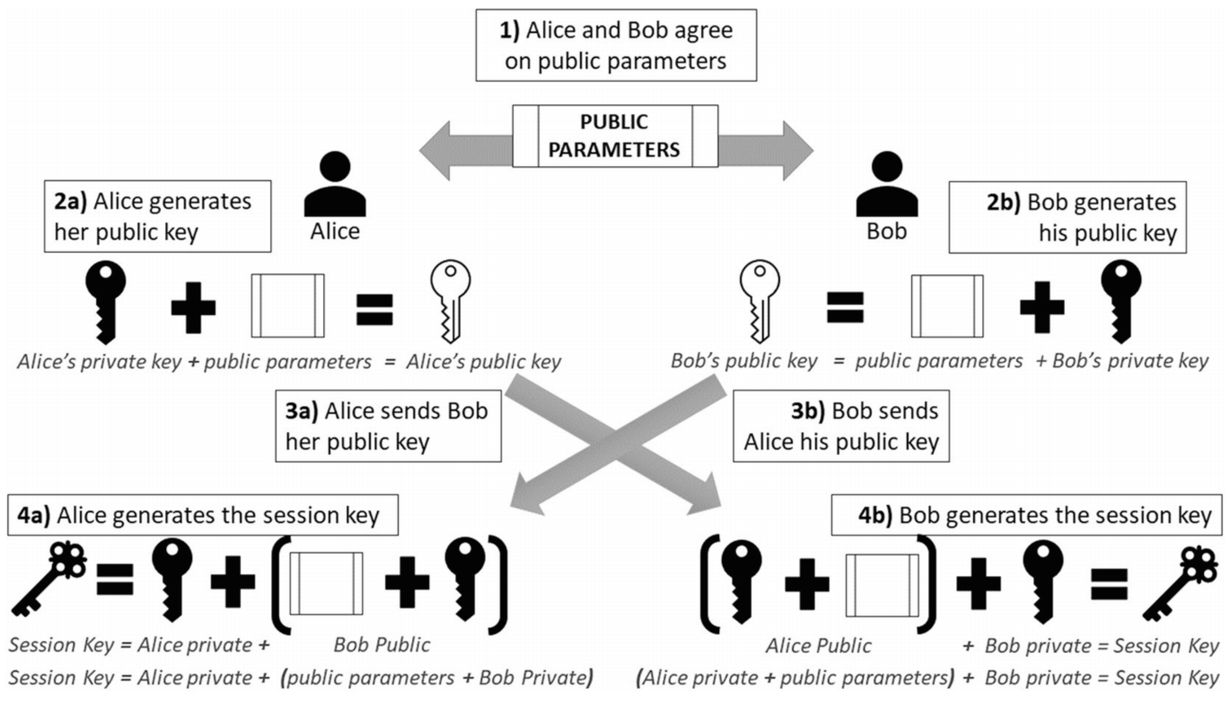

1) Alice and Bob agree on public parameters
PUBLIC PARAMETERS
2a) Alice generates her public key
Alice
Bob
2b) Bob generates his public key
Alice's private key + public parameters = Alice's public key
Bob's public key = public parameters + Bob's private key
3a) Alice sends Bob her public key
3b) Bob sends Alice his public key
4a) Alice generates the session key
4b) Bob generates the session key
Session Key = Alice private + Bob Public
Session Key = Alice private + (public parameters + Bob Private)
Alice Public + Bob private = Session Key
(Alice private + public parameters) + Bob private = Session Key

Figure 5-14 Alice and Bob creating a session key using Diffie-Hellman. Remember that each public key is the participant's private key combined with the public parameters. The private key cannot be extracted from this

combination, but it can be combined with the recipient's private key. Both Alice and Bob combine all three of the same values to produce the same session key

Remember that RSA keys are meant to be long-term keys. RSA key transport will use the same public key to encrypt, and the same private key to decrypt, potentially many session keys. If an attacker were to record these transmissions, if the RSA private key was ever compromised at any point in the future, the attacker could decrypt every recorded session key and all of the data subsequently encrypted by those session keys. This kind of scenario is not unrealistic. Imagine if Alice stores her private key on her computer and then after five years sells it (or recycles it) and buys a new one. A determined attacker could acquire the discarded computer and potentially retrieve the private key. A government looking to build a case against a whistle blower, or persecute a target with blackmail, could record traffic for years while waiting for an opportune time to buy, steal, or otherwise obtain the computer with the public key necessary to decrypt all of the encrypted traffic.

On the other hand, when DH Ephemeral[11] is used, a *new* DH key pair is used for each key agreement *and then discarded*. There is no long-term key to compromise. Even if a single DHE private key is compromised, the attacker gets, at most, a single session key. No additional sessions can be broken from this single compromise.

The downside to Diffie-Hellman Ephemeral is that it is impossible (in practice) to use the public key to identify the other side of the exchange. If it were a long-term key, then Alice or Bob could use various methods to establish the identity of the other and associate that identity with the long-term key. But because a new key pair is generated for every single key exchange, it is no longer possible to do so. What this means in practice is that Alice and Bob can create a key "out of thin air" using DHE, but *either one* *cannot be certain they have created that key with the other*. An attacker could have substituted their public key instead, and there would be no way to tell.

To fix this problem, DHE keys are usually *signed* by a long-term key such as an ECDSA, DSA, or RSA signature. In other words, Alice and Bob are actually using *two* asymmetric key pairs for a key exchange operation. The first key pair is long term and used to authenticate the data origin. The second key pair is an ephemeral DH key that is generated on the spot for a one-time use for generating just one session key. Alice and Bob each sign their respective DH public keys with their respective long-term private keys (e.g., an RSA, DSA, or ECDSA private key). When they each transmit the DH public key to the other, they can verify the signature on the DH public key to be sure that it came from the correct party.

In theory, it would be possible to do an RSA ephemeral key transport. In this version, there would be a long-term RSA key that is used for signing and an ephemeral RSA key that is only used once to transport a single key. After transmitting this single key, the ephemeral RSA key pair would be discarded. Forward secrecy would be maintained, and the compromise of a single RSA key reveals, at most, the data from a single transmission.

The problem is, however, that RSA keys take a long time to generate, whereas DH keys can be generated very quickly. When I say “a long time,” I mean a long time to a computer. RSA keys can be generated within seconds. But seconds is too long for a computer that needs to create hundreds of connections in a second for downloading a web page. This RSA ephemeral mode (which does not really exist and which I have just completely made up to teach the point) would mean that some websites (which may require hundreds of keys) could take many minutes to load.

The point of this exercise is to point out how there are many factors that determine what asymmetric algorithm should be used. Of course, the algorithm must be able to support the operation at a theoretical level, but it also must be appropriate to the relevant problem in practice.

You should also be figuring out from these examples and explanations how it requires a combination of techniques to obtain desired results. For Alice and Bob to communicate in the most recent example, they needed one asymmetric algorithm and keys to sign data, proving that the data originated with the correct party, another asymmetric key pair for DH or ECDH to generate a symmetric session key, and the symmetric algorithms to communicate once the key was established. The symmetric algorithms must support both confidentiality and data integrity such as an AEAD algorithm or a traditional encryption algorithm combined with a MAC.

That is a lot of moving parts. Moreover, I still have not addressed the issue of how Bob knows that Alice’s public key is really hers. That requires more machinery that will be discussed later in the next chapter.

A Word About Quantum Cryptography

There is some bad news about asymmetric cryptography. It is estimated that all of the current algorithms including RSA, DSA, ECDSA, DH, and ECDH will be obsoleted in the next 20 years or so. On the horizon is a new type of computing called “quantum computing.”

There is neither the time nor space in this book for a discussion of a technology that is not yet practical. Instead, this brief section is a placeholder for things to come. You should be aware that at some point in the relatively near future, there will be computers that can break these algorithms.

This is a problem primarily for asymmetric algorithms. It is possible that quantum computing will impact symmetric cryptography as well, but even if it does, the impact will only affect key sizes. The possible concern is a quantum algorithm called *Grover's algorithm* that will theoretically reduce the difficulty of using brute force to find a symmetric key. In the worst case, an AES 128-bit key will be weakened to the equivalent of a 64-bit key, which is not considered sufficiently strong. A 256-bit key, in the worst case, will have the same strength as a 128-bit key does now. For this reason, moving toward AES-256 is recommended by some [17].

However, NIST's post-quantum recommendations, at the time of this writing, do not require an increase and note that it is highly likely that quantum computing will have much impact on AES. In an FAQ, they note a number of expected limitations and, as a result, that "...it is quite likely that Grover's [quantum] algorithm will provide little or no advantage in attacking AES, and AES 128 will remain secure for decades to come" [193].

On the other hand, the asymmetric algorithms discussed in this chapter all rely on certain types of mathematical problems that are "hard" to solve. Without defining what "hard" means, it is enough to simply say that it is impractical to invert the mathematical operations necessary to undo the security of these asymmetric algorithms. That will all change with quantum computing because all of the current types of "hard" problems are "easy" with quantum computing. That cannot be fixed with bigger key sizes; the underlying mathematics simply will not work anymore.

The good news is that there are *other* types of hard mathematical problems that are known to be hard even for quantum computing. Algorithms based on these ideas are called *quantum resistant*. Research teams from all over the world are working, right now, to find effective and efficient quantum-resistant algorithms. The various algorithms, their variations, and competing implementations are tested and vetted in public, peer-reviewed forums. In a future much closer than quantum computing, these algorithms will be in place and fully operational. It is believed by most that when quantum computing comes around, the world will have already migrated to quantum-resistant cryptography.

This does mean, however, that at some point soon-ish, much of the information about asymmetric cryptography in this chapter will be obsoleted to some degree or another. Because the new quantum-resistant algorithms are not yet finalized and in production, however, I really cannot begin to discuss them. So the information in this chapter is the best I can give you for the time being. But you might need to pay attention to the topic and keep your eyes open. There will likely be a lot of changes in the coming years.

Summary

In this chapter, you learned about the cryptography that is commonly used to protect data whether stored on a computer or being transmitted over the Internet. Most cryptography tries to solve the problems of *confidentiality*, *data integrity*, *identity authentication*, and *data origin authentication*. The examples in this chapter dealt with everything except identity authentication. That problem will be addressed in the next chapter.

The cryptographic technologies in this chapter are sometimes called *primitives*. A cryptographic primitive is like a basic building block that can be reused in many different operations. The other topic of the next chapter is how these primitives can be put together into practical systems.

The primitives covered included three basic divisions: unkeyed primitives, symmetric cryptography, and asymmetric cryptography. Hashing is an unkeyed primitive and is used for generating “fingerprint” codes for arbitrary data. Symmetric cryptography uses algorithms with a single shared key used by all participants to encrypt and decrypt data and to create and verify Message Authentication Codes (MACs) on data. Finally, asymmetric algorithms use key pairs to perform various operations. The key pair includes a private key that should not be shared with anyone and a public key that can be shared with everyone. Using public key cryptography, it is possible to encrypt small messages, create digital signatures, and have two parties generate a symmetric session key “out of thin air.”

Asymmetric cryptography is often used in combination with symmetric cryptography. The asymmetric algorithms are typically used to establish an initial origin of the data and establish a shared secret in the form of a one-time use, symmetric session key. Using this session key, the parties can transmit bulk data between one another using symmetric algorithms for both confidentiality and data integrity.

In addition to this high-level overview, you also learned about some of the basic parameters and configurations of the various algorithms. This includes learning about key space (the total number of keys that can be used in an algorithm) and block size for block ciphers. You learned about the need for unique key-IV pairs for symmetric algorithms. And you learned about basic rules of the road for stream ciphers.

The one-time pad (OTP) is a stream cipher that combines the plaintext with a random key that must be the same size as the plaintext. This makes the one-time pad impractical for most applications, but it is also what makes it information-theoretically secure. This means that there is no information an attacker can get about the plaintext from the ciphertext, and it is also impossible to attack with brute force.

Nevertheless, OTP is not some silver bullet for perfect security of data. You learned about various rules that have to be observed, such as not reusing the OTP key. You also learned that stream ciphers, including OTP, cannot provide data integrity. Instead, ciphers have to be combined with something like a Message Authentication Code or MAC. MACs conceptually are like keyed hashes. They provide fingerprints of data that can only be generated and verified using a key. Newer modern modes of operation, such as AES-GCM and AES-CCM, simultaneously produce a ciphertext and a tag (which is like a MAC).

Because OTP is not practical for most applications, other types of stream ciphers are more common such as binary additive stream ciphers. These stream ciphers are kind of like OTP impostors. Instead of having a key of the same size as the plaintext, they use algorithms to turn relatively small keys into comparatively much larger key streams. AES in counter mode (AES-CTR) generates the key stream by encrypting a successive counter. Each encryption of a counter value generates an additional 128 bits (16 bytes) of key stream. On the other hand, the Salsa20 and ChaCha20 algorithms perform more of a hashing-like function on a counter to produce key stream in chunks of 512 bits (64 bytes).

Salsa20 and ChaCha20 were also used to illustrate other considerations in symmetric cryptography, such as performance (with and without hardware acceleration) and matters of transparency in the cryptography design process.

In the study of the asymmetric algorithms, similar issues emerged. For example, RSA keys are slow to generate, while DH keys are fast. This makes DH more suitable for “ephemeral” operations that are done once, and then the keys can be discarded. This is crucial for the creation and sharing of session keys, for example, because ephemeral operations are required to maintain “forward secrecy.”

Forward secrecy is a kind of protection wherein the compromise of a single key results in the compromise of at most one session.

You also learned that digital signatures can be created using a variety of techniques. RSA, for example, encrypts the hash of the data. ECDSA and DSA, on the other hand, produces output pairs such that a specific one of the two values when operated on with both the data and the public key should equal the other value in the pair. One of the important lessons is that there are many mechanisms for achieving the same effective result (i.e., a digital signature).

Finally, I briefly introduced you to the concept of quantum computing, a nascent computing technology that will eventually obsolete the asymmetric algorithms described in this section and require larger key sizes for the symmetric algorithms. Fortunately, new algorithms are already on the horizon that are quantum resistant. Most of the cryptography community believe these algorithms will be operational and well tested by the time quantum computers are ready to break the old asymmetric algorithms.

Further Reading

This chapter does not even begin to scratch the surface of cryptography at any kind of technical level. It is meant to give the nontechnical reader an introduction to the kinds of problems, solutions, and problems with the solutions that are in contemporary core cryptography technology.

For a much deeper dive into the technical aspects of cryptography, I recommend the Handbook of Applied Cryptography, which is freely available online [179].

Probably one of the best sources of up-to-date information is blogs from security/cryptography researchers like Matt Green. He focuses more on practical deployment and usage issues. He also talks about where things go wrong and where cryptography systems have been “cracked,” including his own work finding vulnerabilities in systems [116].

Another interesting source is *Applied Cryptography* by Bruce Schneier. Even though this book was written a long time ago (in computer years) and some of the content is obsolete, there is still a lot of useful guidance and design principles [66]. Bruce Schneier also maintains a blog that covers all kinds of security issues including cryptography [235].

Finally, I would like to give a personal plug for Crypto Done Right (CDR), which is a not-for-profit project that I am personally involved with. At the time of this writing, CDR is just barely getting off the ground, having originally started as a project at Johns Hopkins University based on a grant from Cisco. Unfortunately, the project went through upheaval when I left the university, coinciding with the ravages of the Covid-19 pandemic. But we are getting restarted. The stated goal of CDR is to provide a place for noncryptographers to get the best cryptography advice for dealing with their practical projects. Whether software developers without cryptography experience, IT workers that need to deploy servers, or managers that need to figure out what a news story about crypto being “broken” means, the CDR project is meant to put practical and useful information into your hands [2].

References

2. Crypto done right!

17. The quantum computer and its implications for public-key crypto systems. Technical report, Entrust Datacard, 2019.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing.

[Crossref]

57. Bernstein, D.J. 2005. Salsa20 design. *Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago, Chicago*.

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

66. Bruce, S. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.-2nd*. Wiley.
[zbMATH]

89. Dame-Boyle, A. 2015. EFF at 25: Remembering the case that established code as speech.

97. Duckett, C. 2020. Zoom concedes custom encryption is substandard as citizen lab pokes holes in it.

115. Green, M. 2011. How (not) to use symmetric encryption.

116. Green, M. 2023. A few thoughts on cryptographic engineering.

145. Junod, P. 2001. On the complexity of Matsui’s attack. In *Selected Areas in Cryptography*, ed. S. Vaudenay and A.M. Youssef, 199–211. Berlin/Heidelberg: Springer.
[Crossref]

152. Knudsen, L.R., and J.E. Mathiassen. 2001. A chosen-plaintext linear attack on des. In *Fast Software Encryption*, ed. G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, 262–272. Berlin/Heidelberg: Springer.
[Crossref]

172. Marczak, B., and J. Scott-Railton. 2020. Move fast and roll your own crypto: A quick look at the confidentiality of zoom meetings. Technical Report 126, University of Toronto.

175. Matsui, M. 1994. Linear cryptanalysis method for des cipher. In *Advances in Cryptology—EUROCRYPT’93*, ed. T. Helleseth, 386–397. Berlin/Heidelberg: Springer.

179. Menezes, A.J., S.A. Vanstone, and P.C.V. Oorschot. 1996. *Handbook of Applied Cryptography*, 1st ed. Boca Raton: CRC Press, Inc.
[zbMATH]

192. Najm, Z., D. Jap, B. Jungk, S. Picek, and S. Bhasin. 2018. On comparing side-channel properties of AES and chacha20 on microcontrollers. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 552–555.

193. National Institute of Standards and Technology. 2023. Post quantum cryptography FAQs: To protect against the threat of quantum computers, should we double the key length for AES now?

194. Nir, Y., and A. Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols (7539).
[Crossref]

209. Prodhan, G. 2010. Secret coding inventors finally win recognition.

235. Schneier, B. Schneier on security.

277. Vinck, A.J.H. 2012. Introduction to public key cryptography. Accessed 08 Oct 2018.

278. Vopson, M.M. 2021. The world's data explained: how much we're producing and where it's all stored.

Footnotes

1 Note: This acronym will mean different things in different chapters, so pay close attention to context.

2 Actually, Rijndael is a family of algorithms and had a range of configurations. Only a subset are certified as AES.

3 Note, however, it does not work in the reverse direction. A change made to the last block only changes the last block.

4 Although, in practice, block ciphers must do the same thing. As you saw in the previous section, if each block is encrypted exactly the same, patterns emerge. Modes of operation like Cipher Block Chaining mode carry over the output from one block's encryption into the input of the next. In short, how the cipher encrypts the block is changed or influenced by the previous block's encryption. For this reason, CBC mode may be considered a form of stream cipher even though it is not usually referred to as such.

5 Some machines cannot generate that much random data quickly.

6 Even though this is true from a purely theoretical perspective, an attacker may have some contextual knowledge that would permit them to recognize the “correct” plaintext. In practice, however, for a message of even a relatively short length, the odds of getting a zero key (key of all zeros) are very unlikely. For the short 14-character message I used as an example, the odds of getting the zero key are $2^{14*8} = 2^{112} = 5{,}192{,}296{,}858{,}534{,}827{,}628{,}530{,}496{,}329{,}220{,}096$.

7 In order to make the image displayable, the header portion of the BMP file is not modified. But all of the data that comprises the visible image is XORed.

8 There are other modes of operation that produce a stream cipher, but counter mode is the most common and probably the easiest to understand.

9 Although AES-CTR mode and other modes like it do have limits on how large the key stream can safely get, these details are outside the scope of the book.

10 Sometimes, the term *key exchange* is used as a synonym for *key agreement*. However, I am going to follow the *Handbook of Applied Cryptography*’s nomenclature. *Key establishment* is getting the parties to share a key. *Key transport* and *key agreement* are two forms of key establishment. The term *key exchange* can refer to various algorithms including the Diffie-Hellman algorithm discussed in this section, but it is not a synonym.

11 Diffie-Hellman Ephemeral is abbreviated DHE. However, it is such a common mode for DH that even when it is not expressly stated, or even when the DH abbreviation is used, it may very well be DHE. You may need to ask for clarification if it is not clear from context. This is also true for ECDHE.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_6

6. Cryptographic Systems Technologies

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
Using the building blocks from the previous chapter, this chapter puts them together in two example systems. One of these, Transport Layer Security (TLS), is used to secure Internet communications. One key consideration for the design of a cryptographic system is protection against a *man-in-the-middle*, a model wherein the attacker can intercept, and potentially modify or generate, messages.

Key Concepts

1. *Data at rest* and *data in motion* are examples of the states of data (analogous to states of matter).
2. *Man-in-the-middle* is an attack model where the attacker can get in between communications and impersonate either side to the other.
3. *Transport Layer Security*, or TLS, provides an end-to-end encrypted channel; TLS is used to secure HTTPS.

4. A *Public Key Infrastructure* is used to provide a basis for being able to *trust* a certificate by chaining it to already-trusted entities.
5. *Certificate transparency* is a relatively new technology designed to prevent compromise of roots of trust in PKI.
6. Key management is one of the most challenging problems in cryptography; keys have a life cycle including phases such as creation, distribution, management, archival, and deletion.

Common Pitfalls and Misunderstandings

1. Reminder from the last chapter, YANAC (You Are Not A Cryptographer); please do not try to put together your own cryptographic system.
2. Keeping the private key secure is of utmost importance. Remember, as long as the key is secure, anything else about the system can be known (encryption algorithms, key exchange protocols, even the encrypted data itself), and the system is still secure.

Useful Vocabulary

- **TLS**: Transport Layer Security; provides confidentiality, authenticity, and integrity on an Internet connection
- **Certificate Authority**: A trusted party responsible for signing certificates used to prove identity
- **Secure Authenticated Channel**: A communication channel that provides both confidentiality and authenticity

- **DH**: Diffie-Hellman; an algorithm for negotiating a session key between two parties using public and private keys
- **IV**: Initialization vector; a public piece of data used to randomize inputs to AES modes like CBC
- **MAC**: Message Authentication Code[1]

In Chapters 4 and 5, I introduced a number of building blocks of cryptography. These building blocks are often called cryptographic *primitives*. These primitives included

- Hashing functions
- Symmetric key encryption and decryption
- Symmetric key Message Authentication Codes (MACs)
- Asymmetric key signatures
- Asymmetric key methods for key exchange

These primitives are generally not useful by themselves but have to be combined into a functional cryptographic system. Different types of systems have different purposes and can be used in different environments. But generally, a cryptographic system will provide some combination of data protections, authentication functions, and authorization functions.

Cryptographic systems are designed to protect data in various states. For example, a cryptographic system might be designed to protect data in storage. Another system might be designed to protect the data traveling across the Internet. Other systems might be appropriate for specialized applications that deal with both stored and transmitted data. In every case, however, the cryptographic system has to prevent attackers from violating the expected data permissions, defeating the authentication, and/or obtaining unauthorized access.

In particular, the data protections enforced by a given system are often one or more of the security goals discussed in Chapter 4: confidentiality, data integrity, entity authentication, and message origin authentication. For example, a cryptographic system that is tasked with ensuring confidentiality must enable the authorized parties to read the data while at the same time preventing unauthorized parties from doing so.

In this chapter, I review a cryptographic system designed for protecting data at rest in the form of a file encryption system. I also review the ubiquitous TLS (Transport Layer Security) protocol that is used to provide the security for HTTPS connections on the Internet. This is an example of protecting data in motion. The purpose is not just to introduce the technologies; it is also to illustrate some of the common solutions and pitfalls present in these kinds of systems. If you can understand these issues, you will be better positioned to understand any cryptographic system.

On that note, there are four other cryptographic systems discussed in this book in other chapters. In Chapter 7, I discuss how ransomware uses a cryptographic system as a means to attack users, rather than as a means for users to defend themselves. In Chapter 9, I review the OAuth protocol, which is used to provide Single Sign-On (SSO) capabilities on the Web. And finally, in Chapter 10, I discuss two cryptographic systems used for protecting email communications. Again, however, use this chapter to learn about important concepts applicable to any of these systems.

The rest of this chapter is organized as follows. First, I introduce a very simplified and basic model of a generalized attacker by discussing the concept of a *man-in-the-middle* (MITM). Next, I will introduce how different cryptographic primitives are commonly used in a cryptographic system and how these thwart attackers. After that, I introduce the first example system: TLS, which is used by HTTPS and is, therefore, a key component to much of the security of the Internet. Within this discussion is an overview of the *public key infrastructure* that is used to make TLS work. The last section of the chapter presents a summary of the IEEE 1619.1 storage encryption protocol.[2]

The Attacker: Man-in-the-Middle

Before diving into this section, I acknowledge that the term “man-in-the-middle” is not considered inclusive. There are a number of alternatives being considered such as “person-in-the-middle.” While there is no standardized form yet accepted, I will continue to use “man-in-the-middle” while acknowledging that women, nonbinary people, and others may be just as thieving, scheming, and deceptive as men.

One of the common objectives for an attacker is to get in between two communicating parties. If an attacker can get in between Alice and the bank, the attacker can read any messages, modify any messages, and even insert completely new messages. This kind of attacker, one that is in between communicating parties and can read and alter messages, is a *man-in-the-middle* (MITM).

How would an attacker ever get “in the middle”? One way is to intercept communications.

The Internet’s default communications protocols do not have any security guarantees. For example, they do not provide any *confidentiality*. None of these protocols do any kind of encryption. Internet packets, when transmitted, are handled by a series of intermediaries such as devices known as routers and proxies. If an attacker controlled one of these devices, perhaps by being an insider or through an attack on the device, they could easily read any data transmitted between parties communicating over the Internet.

At least as bad, if not worse, is the fact that default Internet protocols do not provide authenticity. If an attacker took control of a communication node, they could intercept any communications passing through those nodes. Instead of letting them pass on to the real destination, they could respond themselves and pretend to be the real destination.

One way this happens in practice is attackers that take advantage of a public WiFi, such as the WiFi available at a cafe, airport, library, or so forth. Suppose you sat down to use the Internet at such a location. You decide to browse over to your bank to check your balance. The packets that you transmit over WiFi are visible to other individuals using the same WiFi. So, even if they did nothing else, without some cryptographic system to encrypt your data, the attacker could simply observe all of your traffic, read your balance, and otherwise spy on your actions. But worse, the attacker could pretend to be the bank and respond to your laptop as if they were the bank. So long as their packets got to you before the real ones, your laptop would accept them without question. Now the attacker could lie to you about your balance or, worse, alter any payments or other transactions you might make.

What about data in storage? If you are just encrypting the data on your hard drive, can there even be an attacker in the middle? If your hard drive were stolen (e.g., your laptop was stolen at an airport), any of the people that access your drive become MITM attackers. If there were no cryptographic protections, they could certainly read your data. But they could also alter the data and then “return” your laptop. In the case of storage, you might be both parties, and a thief would be an attacker in the middle between you and your later self.

Cryptographic systems have to be designed to deal with more specific threat models than this, but this general description of an attacker that can read, write, and modify messages is a good starting point for why cryptographic systems are needed and why we use the components we do to create them.

Putting Together a Cryptographic System

Suppose that a person sits down to their computer and wants to connect to their online banking service. As I did in Chapter 5, I will call this person Alice. Perhaps Alice would like to check her balance or maybe pay a bill. How should the data (i.e., Alice's balance) be protected?

Two common issues are protecting Alice's balance while that data is *at rest* and protecting Alice's balance while the data is *in motion*. Based on the concept that matter has states (e.g., solid, liquid, and gas), data is described as having states. The two most commonly dealt with are *data at rest*, such as data stored on a disk, and *data in motion*, such as data being transmitted from one computer to another.[3] Protecting data in these different forms, or states, has some similarities but also some important differences.

The goals, of course, are generally the same. I already referenced confidentiality, data integrity, entity authentication, and message origin authentication in the introduction. For simplicity, however, I am going to reduce these four goals to three slightly more generic properties:

- Confidentiality
- Authenticity
- Integrity

Confidentiality remains the same in both lists. For the purposes of our cryptographic systems, confidentiality means that only authorized parties can *read* the data.

Integrity is more or less the same as “data integrity” and is the flip side of confidentiality. For the purposes of our cryptographic systems, integrity means that only authorized parties can *write* (or modify) the data.

Authenticity is a combination of entity authentication and origin authentication. Within a cryptographic system, this property means that the publisher of the data (e.g., the author or sender) is verifiable. This includes entity authentication because the publisher’s identity is verified. It also includes origin authentication because the message is verified as coming from the identified publisher.

It should also be noted that authenticity must include integrity. If the data has been altered, then authenticity would no longer verify (because the data publisher cannot be verified). However, integrity can be had without authenticity. In certain cryptographic systems, two communicating parties do not verify each other’s identity but do ensure that the data transmitted between them is not altered. For this and other reasons, I have simplified the terms to focus integrity on the modification of the data itself and authenticity to refer to knowing the source.

Also, it is important to understand what is meant by *read* and *write* in this context. When I say that only authorized parties can *read* the data, I mean *understand* the data. An unauthorized party might be able to get the encrypted data, but so long as they cannot decrypt the data, they cannot “read” it from a security perspective. Similarly, cryptographic systems often protect data against unauthorized modification by using verification codes. *Message Authentication Codes* and *signatures*, both of which were discussed in Chapter 5, attach a code to the data being protected. When I say that these kinds of codes ensure that only authorized parties can *write* the data, I mean that any unauthorized write can be detected. That is, an attacker may, in fact, change the data, but the verification code would prove that the changes were unauthorized.

How do these various goals help Alice and her balance? Confidentiality is probably the most obvious. Only authorized employees of the bank and Alice should ever be able to read Alice’s balance. Most people can understand wanting to keep that information private. This confidentiality should apply whether the data is sitting on the bank’s server or whether it is in transit across the Internet.

Integrity is equally important. Clearly, nobody should be able to alter Alice’s balance, either on disk or on its way to Alice’s browser. Given that much of our currency and financial data is simply data recorded electronically, data integrity prevents Alice’s money from being stolen or inflated.

The authenticity property is the one most commonly overlooked by those unfamiliar with cryptography. This property is most easily understood from the perspective of data in motion between Alice and the bank. How does Alice know that she is actually communicating with the bank? When communicating over the Internet, messages are passed along a chain of intermediate devices. What has prevented one of these devices, either because of a malicious operator or intrusion from an attacker, from *pretending* to be Alice’s bank? Authenticity ensures that Alice can verify that she is, in fact, speaking to the real bank and not an impostor.

Cryptographic systems also deal with authentication concepts, such as those discussed in Chapter 2. For example, data stored on disk is often protected with a password that is converted into a key for the encryption algorithm. I discuss this in more detail later in the chapter. The two examples in this chapter do not do much with authorization, though. As you will see, you either get access or you do not. But the other cryptographic systems described in later sections will deal with more fine-grained permissions and a more clear authorization model.

Putting the Pieces Together

Now that I have walked through some of the goals of a cryptographic system, it is time to look at how cryptographic primitives can be put together to enforce or enable those goals. In order to do this, it is necessary to break down our security goals into more concrete tasks.

Confidentiality Components

Let’s start with confidentiality. Both of the example systems in this chapter provide confidentiality. One system provides confidentiality at rest, and one provides confidentiality in motion. In both cases, symmetric encryption will be used to provide the encryption of the data. Why symmetric encryption and not asymmetric encryption? As discussed in Chapter 5, symmetric encryption is designed to work on any amount of data and is fast, while asymmetric encryption is designed for small messages and is very slow. Symmetric is almost always the right choice for what is called “bulk” encryption.

Of course, encryption is not useful unless the data can also be decrypted. Based on what you learned in Chapter 5, can you see what is required for authorized parties to be able to both encrypt and decrypt? In general, both the encrypting party and the decrypting party will need to agree on the following:

1. The encryption algorithm, such as AES or ChaCha20
2. The algorithm parameters, such as CBC or CTR mode
3. The mode parameters, such as IV or nonce

4. The key

Of these, the key presents an interesting challenge because it *must* be kept secret between the two parties. This is a nontrivial challenge.

The obvious challenge is how do both parties share the key? Especially for data in motion, the parties may be separated by geographic distance. For data at rest, the two parties may be the same party (i.e., the same person that encrypted the data is now decrypting the data), but that person is separated from themselves *in time*. How is the key kept or maintained by the person over time? The key must be kept secret, so it generally cannot be transmitted unencrypted or stored insecurely. What good would it do to encrypt data on disk and store the key to decrypt the data on the same disk?

But there are less obvious challenges. For example, in the cases where the key is shared between two parties, how long should the key be shared between them? One thing that is hopefully obvious is that the same key cannot be shared across multiple pairs. If Alice and Bob share a key, and Alice and Charlie share the same key, Charlie has access to all of Alice and Bob’s secrets.

For this and many other reasons, it is common in data-in-motion communications to use a *session key*, or a key that is only used for a specific communication period (the “session”).

Data at rest generally does not have the same kind of sessions as data in motion does. But it must also use multiple keys. In some cases, using a different key for each encrypted chunk (such as a fixed-size chunk of a file) is recommended by the system. In all cases, risks increase as more ciphertexts are created by the same key.[4] Thus, keys have a *key life cycle* that includes creation, distribution, storage, archival, and destruction. All of this must be managed by the cryptographic system.

Moreover, there must be a way for the parties in the system (e.g., Alice and Bob) to *synchronize* their keys. For symmetric key cryptography, they must *share* the same keys. If Alice destroys a key and creates a new one, Bob must mirror the process in order to continue being able to read Alice’s encrypted data.

Cryptographic systems tend to solve these problems with two operations. First, Alice and Bob (or whatever parties are involved) will have a means for exchanging a master key with each other. As you learned in Chapter 5, algorithms like Diffie-Hellman and RSA encryption enable these kinds of exchanges. Second, Alice and Bob will use a protocol that will enable the creation of all other keys. Different approaches to this will be discussed in each example cryptographic system.

As alluded to earlier in this section, it is also necessary for Alice and Bob to agree on IV or nonce parameters and modes of operation. Although this is somewhat less complicated because it does not have to be secret, there are still a number of issues that must be addressed. For example, IVs should generally not be predictable even though they can be public information once created. In other cases, data exchanged must not be modifiable by attackers for various reasons. And that leads us to our next set of requirements.

Integrity Components

Integrity requirements take on many different forms, but two general categories are the integrity requirements that are required for ensuring there have been no unauthorized modifications to protected data and the integrity requirements for establishing authenticity. I will handle the latter in the next section.

For protecting bulk data, MACs (Message Authentication Codes) are a common solution. A MAC is typically kept with the protected (possibly encrypted) data, either in transmissions or in storage. Either way, the MAC must be checked before determining that the data is valid. For example, a chunk of encrypted data may be transported over the Internet. A MAC generated for the encrypted data is sent with it. Suppose that an attacker intercepts either, or both. If the attacker makes any attempt to alter either one, the cryptographic system at the receiving end can verify the MAC as the first step. If the MAC fails, there is no need to proceed with the decryption process.

MACs, as discussed in Chapter 5, require a key. Most MACs are symmetric key based, so all involved parties must share the same key to be able to generate or verify the data. All of the same key life cycle issues discussed in the previous section apply here as well. However, the solutions can be, and usually are, combined. That is, whatever process is used to generate additional keys for confidentiality can also be used to generate keys for integrity. Thus, Alice and Bob can exchange a single master key and then derive a nearly limitless number of keys for confidentiality and a nearly unlimited number of keys for integrity.

In modern best practices, however, this entire set of requirements for integrity is completely integrated into the requirements for confidentiality by the use of Authenticated Encryption with Additional Data (AEAD). As discussed in Chapter 5, these algorithms perform both encryption and message authentication (integrity checking) with a single key as part of a single operation. For many reasons, this is the preferred approach whenever possible.

Authenticity Components

Many cryptographic systems require some kind of assurance that Alice is indeed talking to Bob or that Bob is indeed talking to Alice. Asymmetric cryptography is often used for these operations.

What makes asymmetric cryptography so effective for proving identity is the ability to assign one private key to a single party. In symmetric cryptography, the parties have to share a key. The same shared key can never be used to prove identity between more than two people. Suppose Alice, Bob, and Charlie all share a key. If a message arrives claiming to be from Alice with a MAC attached to it, Bob cannot be sure if Alice sent it. After all, Charlie could have forged it and pretended to be Alice. The only solution to this would be for Alice to have a uniquely shared key with every party she wishes to prove her identity to. That is a significant key management challenge. Even if such a system were constructed, Bob could still forge messages from Alice to himself.[5]

But with asymmetric cryptography, Alice can have a private key that no other party in the world has ever had access to. Alice can issue her public key far and wide. Using her private key, Alice can sign messages and distribute them. Anyone with Alice’s public key can verify that Alice published the message because nobody else has (or should have!) the private key necessary to generate the signature.

Signatures, like MACs, are typically sent with the message they sign. The recipient receives the data and the signature, verifies the signature for the data with the public key, and knows *both* that the message is unaltered and who the publisher was. Notice that this ensures that the message has integrity as hinted at in the previous section. The message must be unaltered because if the message was altered, the signature would not verify.

Authenticity components are often combined with other components, such as the key exchange required for confidentiality and integrity of the bulk data. For example, even if Alice and Bob are using Diffie-Hellman to create a shared symmetric key between them, they need to know that they are, in fact, performing the Diffie- Hellman operation with each other. For example, let us bring back our MITM attacker from earlier in the chapter. If Alice begins a Diffie-Hellman exchange with Bob, what prevents an MITM attacker from intercepting this message and responding to it themselves? Alice would have no idea that the messages coming back were from the attacker instead of Bob. Alice would still derive a shared key using DH, but the key would be shared with the MITM attacker instead of Bob! Figure 6-1 illustrates this attack.

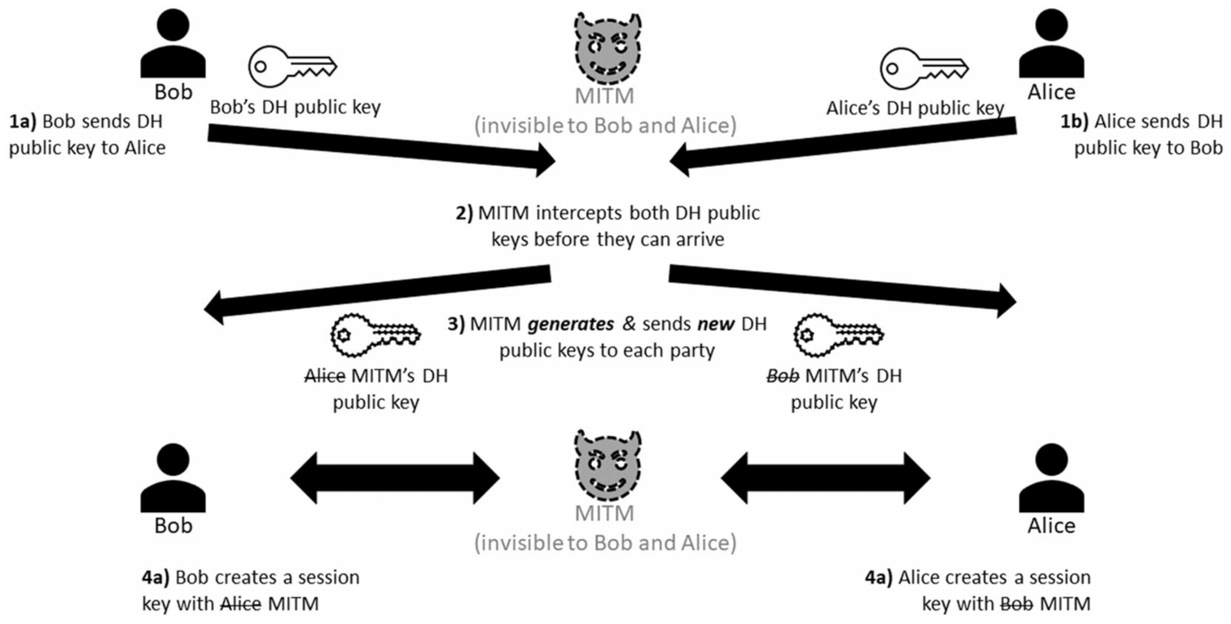

Bob
Bob's DH public key
MITM
(invisible to Bob and Alice)
Alice's DH public key
Alice
1a) Bob sends DH public key to Alice
1b) Alice sends DH public key to Bob
2) MITM intercepts both DH public keys before they can arrive
3) MITM ***generates*** & sends ***new*** DH public keys to each party
~~Alice~~ MITM's DH public key
~~Bob~~ MITM's DH public key
Bob
MITM
(invisible to Bob and Alice)
Alice
4a) Bob creates a session key with ~~Alice~~ MITM
4a) Alice creates a session key with ~~Bob~~ MITM

Figure 6-1 Diffie-Hellman exchange subverted by an invisible MITM

However, using signatures, Alice and Bob can validate that the data they are receiving for the DH operation is from the other. If the signatures on the DH data do not match, they will reject the operation. If the signatures do match, they can be assured that the DH exchange is taking place with the correct party. Once assured of the identity of their partner, they can complete DH to produce a master key and subsequently derive all of the necessary additional keys. The protected DH exchange is shown in Figure 6-2.

Securing Web Communications: HTTPS and TLS

Our first cryptographic system that puts together these concepts is a system that secures data in motion. Specifically, it protects network communications, including communications over the worldwide Internet, from an MITM attacker. I will continue with the example of Alice and her bank. In this specific example, Alice is communicating with her bank over the Internet (e.g., online banking).

To be protected, Alice needs a security protocol that solves at least three problems. In order to be safe from MITM attackers, Alice requires that even if her packets, or the bank's packets, are intercepted, the attacker cannot read the data. Also, if these packets are intercepted, the MITM attacker should not be able to modify them (undetectably). And finally, the attacker should not be ble to generate their own packets and then lie about them, pretending that they are from the bank.

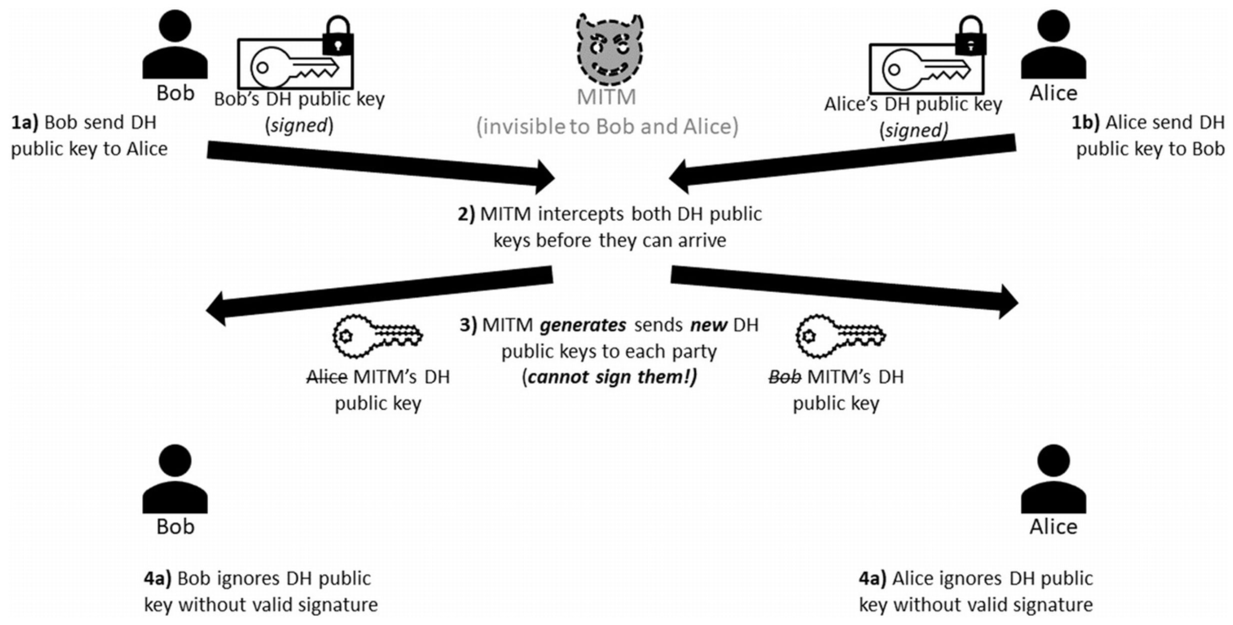

Bob
Bob's DH public key (*signed*)
1a) Bob send DH public key to Alice
MITM (invisible to Bob and Alice)
Alice's DH public key (*signed*)
Alice
1b) Alice send DH public key to Bob
2) MITM intercepts both DH public keys before they can arrive
3) MITM ***generates*** sends ***new*** DH public keys to each party (***cannot sign them!***)
~~Alice~~ MITM's DH public key
~~Bob~~ MITM's DH public key
Bob
Alice
4a) Bob ignores DH public key without valid signature
4a) Alice ignores DH public key without valid signature

Figure 6-2 Diffie-Hellman protected from MITM attacks by signing the DH public key. The DH public key must, itself, be signed by another private key (not shown)

As explained earlier in this chapter, the original, default protocols of the Internet do not provide any of these capabilities or protections. However, the TLS protocol, which is layered on top of existing Internet capabilities, does.

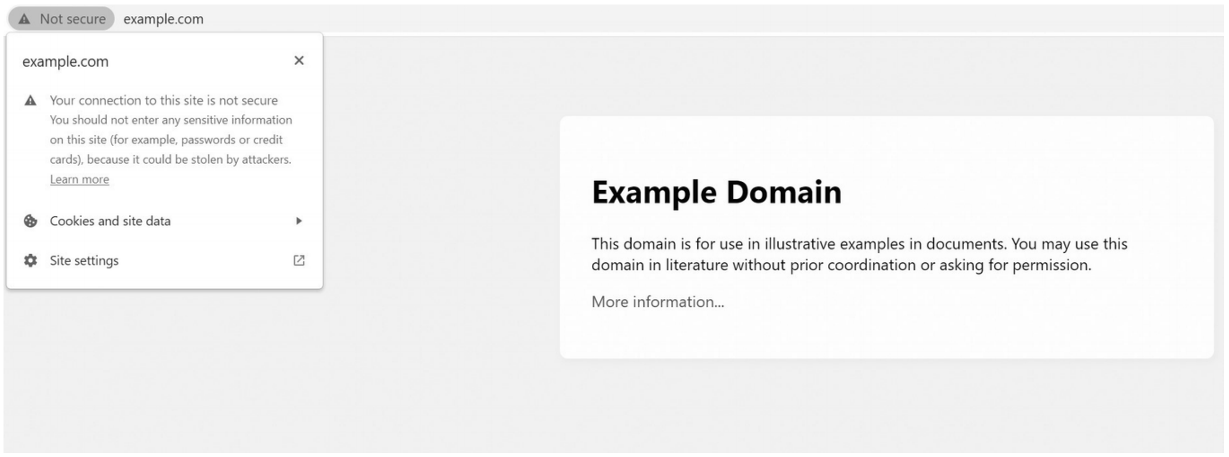
The TLS Protocol

This section does require understanding computer networks a little bit. If you are unfamiliar with how computer networks operate, you should review Appendix C. It would be helpful to understand "protocol," "protocol stack," and the "OSI model." Understanding the concepts behind TCP/IP would also be helpful.

The goal of the TLS protocol, which stands for "Transport Layer Security," is to provide confidentiality, authenticity, and message integrity for both ends of a connection on the Internet. This is also called *end-to-end* encryption, meaning that there are no decryptions along the path between the communicating parties. Another common term for describing this kind of connection with both confidentiality and authenticity between two parties is a *secure authenticated channel* or SAC.

In terms of the network stack, TLS operates above the TCP layer (layer 4) and below the application layer (layer 7). The exact identification of what layer of the OSI model TLS fits into is debated. Nevertheless, TLS operates *like* a transport layer from the perspective of applications.

This is the case with HTTPS. HTTPS is the secure version of HTTP, but it is, in fact, just HTTP (a layer-7 protocol) using TLS for transport instead of TCP. Most browsers used to always show the HTTP or HTTPS protocol in the URL bar. However, it is now more common to just show the URL and a lock icon if HTTPs is used. Still, if you click to edit the URL, many times the HTTPS will still be displayed. Figures 6-3 and 6-4 show the website example.com over an insecure HTTP connection and a secure HTTP connection, respectively.

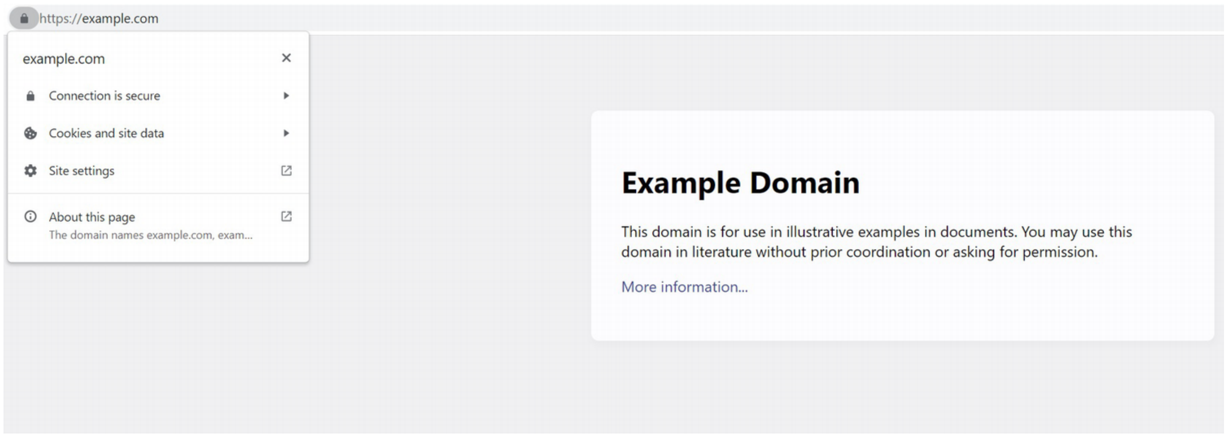


Your connection to this site is not secure
You should not enter any sensitive information on this site (for example, passwords or credit cards), because it could be stolen by attackers.
Learn more

Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

Figure 6-3 The URL bar for Google Chrome when navigating to http://example.com. This is the insecure version of the website, and Chrome shows the padlock in its unlocked state



Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

More information...

Figure 6-4 The URL bar for Google Chrome when navigating to https://example.com. This is the secure version of the website. All data is sent over a TLS secure authenticated connection. Chrome shows the padlock in its unlocked state. The URL bar has been clicked for editing causing Chrome to show the https

By way of background, TLS was originally named the "Secure Socket Layer" (SSL) protocol and was created by Netscape in the mid-1990s. Netscape was an early Internet company that developed one of the first web browsers called Netscape Navigator. SSL Version 2 was the first public release of the protocol, followed by version 3 shortly thereafter. Subsequently, it received a few changes and became an open standard renamed TLS 1.0.[6] The updated versions since that time have been released to update cryptography and alleviate problems with the cryptographic protocol. Version 1.2 has been around for a number of years and is still considered current. Version 1.3 was also released more recently, but is not currently being described as a replacement to 1.2 (both versions are considered current).

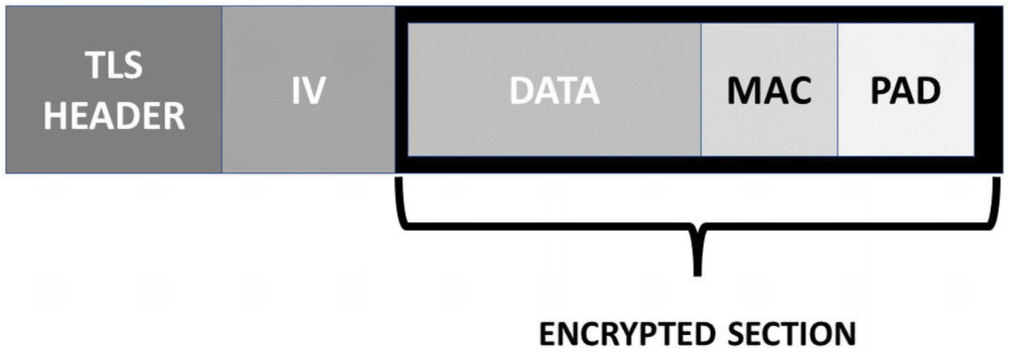


Figure 6-5 An example TLS data packet. The encrypted section includes the data as well as the Message Authentication Code

The TLS protocol implements a cryptographic system with the components I described earlier in this chapter. It makes use of the following primitives:

- Asymmetric cryptography for digital signatures (e.g., RSA signatures)
- Asymmetric cryptography for key agreements (e.g., Diffie-Hellman)
- Symmetric encryption (e.g., AES)
- Message Authentication Codes (e.g., HMAC)
- Modes of operation (e.g., CTR, GCM)

As illustrated by this list of concepts, there is a *lot* of cryptography that goes into making TLS work. To help walk through how it works, I am actually going to start in the *middle* and work backward toward the *beginning*. The middle of TLS is when regular data transfer is happening. When secure messages are being sent over a protected TLS channel, the data is usually encrypted with a symmetric key algorithm to provide confidentiality and protected with a Message Authentication Code to prove authenticity and message integrity.

TLS Data Transfer

Data transferred in TLS is protected by wrapping data chunks (called "fragments") in TLS packets that are protected with encryption and message authentication. Figure 6-5 illustrates one example of a TLS data packet.

On the sending side of the TLS process for this kind of data packet, the plaintext data is first protected with a MAC code.[7] Then, the plaintext, MAC code, and padding are appended together. The combined data (plaintext, MAC, and padding) are then encrypted with AES and a randomly generated IV. A TLS header and the IV are prepended to create the full TLS packet. Once constructed, the TLS packet is sent over the network.

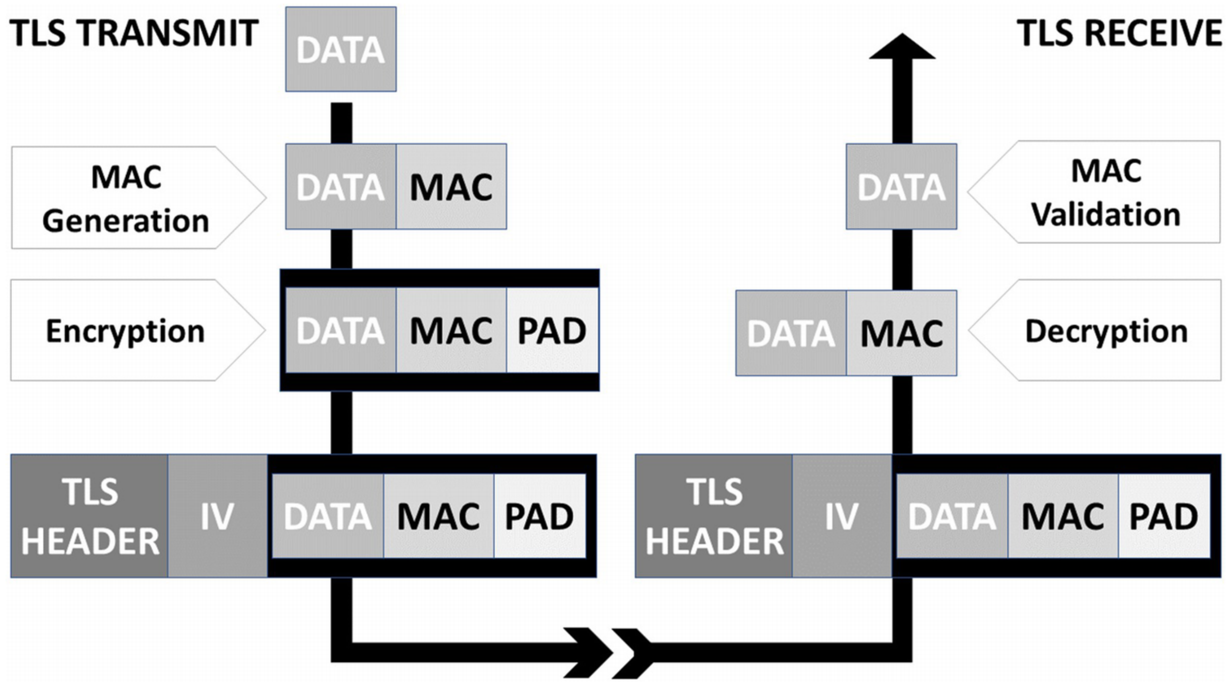


Figure 6-6 A transmission and reception of TLS data for an appropriate configuration. On the sending side, the data first has a MAC generated for it, and then the data and the MAC are encrypted. When the TLS packet is received, the data and MAC are decrypted. If the MAC verifies, the data is accepted as authentic

Once the TLS packet arrives, the TLS receiver first decrypts the packet, making use of the IV to do so. The decrypted data includes plaintext, MAC, and padding. The padding is stripped off, and the MAC code is verified. If the MAC verifies correctly, the data is accepted as correct and passed on to the application process expecting it. This process, on both the sending and the receiving side, is illustrated in Figure 6-6.

Please note that TLS is extremely configurable (especially in version 1.2). This is just one example of how a data packet can be formed. In this example, the data is protected by encryption and a MAC. The MAC is actually included in the encrypted segment. The receiving TLS system must first decrypt the segment, then use the MAC to determine if the contents have been altered. It is sometimes helpful to think of the encrypted segment as an "envelope." Decrypting the section is like unpacking data out of the envelope. So when this segment is decrypted, the plaintext data and MAC are extracted. The padding is just data inserted to make the total length a multiple of 16, as some modes of AES (e.g., CBC mode) only work on multiples of 16 bytes.

This example is useful, however, to illustrate confidentiality, authenticity, and message integrity. For example, suppose that an attacker that has managed to get between the sender and receiver attempts to read the data. Because the attacker *does not have the required symmetric AES key*, it is impossible to decrypt the record fragment. So this TLS data packet has preserved confidentiality.

Second, the MAC is used to ensure that the data came from the authorized source. Remember that there are ways to *change* an encrypted message that cannot be read. But if the attacker was somehow able to change the data inside the encrypted envelope, the MAC code should[8] fail verification on the receiving side, and the receiver will know that the data has been altered. The attacker cannot generate a fake MAC code because the attacker does not have the key.

A verified MAC proves both authenticity and integrity. Only the party with the shared key for MAC generation could have generated the MAC. Any changes by an unauthorized party should result in a MAC verification failure. The man-in-the-middle is thwarted by the security on the data packet.

But wait! We said that the attacker could not read the data, or forge the data, because the attacker did not have keys. *Where* did the TLS sender and receiver get their keys from? As I said, I started in the middle. The middle is where both sender and receiver have symmetric keys that enable them to send data packets. *If* they have valid keys, the data packets have the required security properties.

This is where the working backward part comes in. Before the data transfer part of TLS began, both sides needed to establish keys. These keys are *ephemeral* keys that will only be used for a single session (data sent over TCP until the connection is closed). The question to answer is, *where do these session keys come from*?

The TLS Handshake

The TLS protocol begins with an initialization process called the *TLS handshake*. It is an exchange of data between the TLS endpoints in order to do two very important things. First, the TLS handshake must authenticate identity.[9] The second goal is to mutually derive the ephemeral session keys.

The handshake is also where the various TLS versions are most different from one another. For the purposes of illustration, I will review the TLS 1.2 handshake. TLS 1.3 simplifies the handshake and speeds it up considerably. So if you can wrap your head around how 1.2 works, 1.3 will be much easier.

In the TLS 1.2 handshake, the client starts out by sending what is called a *Client Hello* message. The Client Hello message communicates to the server that the client wishes to start a TLS session. It also passes along configuration information necessary for setup. Note that this message is not encrypted. It cannot be encrypted yet because neither side has any keys of the other.

Once the server receives the client’s hello message, it responds with a *Server Hello* message with configuration information. But the server sends several more messages right after. The first is a *Certificate* message, which includes the server’s certificate. I will explain certificates in detail in the next section. For now, just know that a certificate contains a public key and data about the public key such as whose public key it is. The server distributes its public key to the client in the certificate so that the client can use the public key to validate anything signed by the server’s private key.

But if you stop to think about it, this is really problematic. It is somewhat like when a shopper at a retail store uses their credit card, and the store clerk notices the credit card is not signed. The clerk sometimes asks shoppers (this has happened to me) to sign the card in front of them *and then compares the signature on the card to the signature on the receipt*. The clerk literally saw the shopper sign both the card and the receipt in front of them. Why would they be different?

Similarly, the client got a public key from the server and then uses it to verify signatures from the server. Of course, they are going to match. The server can hand out any public key it wants, so it is not difficult to hand out a public key that will match.

To make this really work, the client has to have some assurance that the public key it receives *is actually the true public key of the server*. In the previous example of Alice, she (or rather, her browser) needs to know that the public key in the certificate in the TLS message is truly the public key belonging to her bank.

In the next section, when I discuss certificates, I will explain in detail how they are verified and authenticated. For now, accept on faith that the certificate is verified, and Alice's browser trusts that the certificate it received actually belongs to Alice's bank.

In addition to the certificate, the server also transmits a Diffie-Hellman message. As explained in Chapter 5, DH is used for creating a new shared key "out of thin air" between two parties. The server transmits its part of the DH exchange in a message called a *Server Key Exchange* message. This message is *signed by the server's private key* to ensure that the DH parameters cannot be altered by a man-in-the-middle attacker. This information will be verified by the client. (Note: TLS can be configured to use RSA key transport instead of Diffie-Hellman, but I will use the DH configuration for this example.)

Finally, the server sends a conclusory message called a *Server Hello Done* message. The server has, therefore, sent *four* messages together: Server Hello, Certificate, Server Key Exchange, Server Hello Done.

Once the client (e.g., Alice's browser) receives all four of these messages, it knows that it is talking to the true server (as will be explained in the next section), and it has the server's DH data. The client uses the public key in the certificate to validate the server's DH data and make sure it was not sabotaged or altered by an unauthorized party. Once the DH information from the server is validated, the client now generates its own DH data and combines both to derive a session key.

The client is now ready to send its own data back to the server. First, it sends its own DH data to the server in a *Client Key Exchange* message. The client knows that the server will use this message to generate the same session key. The client knows that it and the server can now exchange encrypted messages. It sends a special message called a *Change Cipher Spec*, which tells the server it is now switching to encrypted messages, and sends an encrypted *Finished* message.

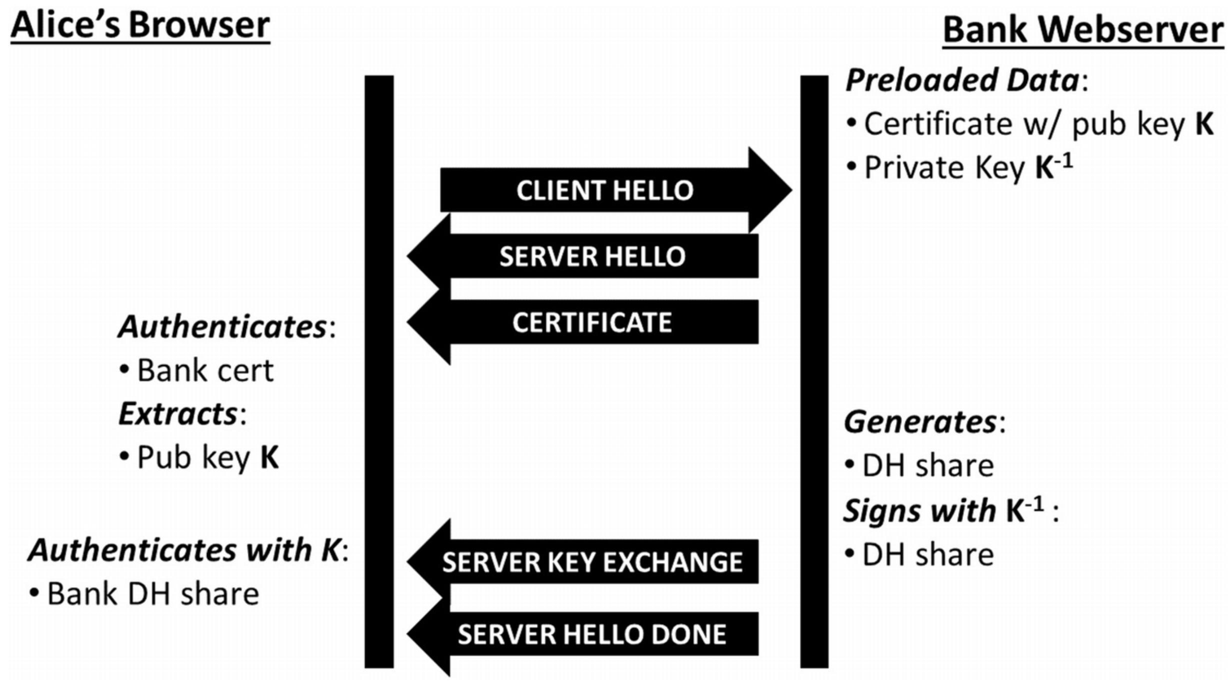


Figure 6-7 The first half of the TLS handshake between Alice's browser and the bank's web server

The server receives the client's key exchange data, change cipher spec, and finished messages. It uses the key exchange data to derive the same session key, which it then uses to decrypt and verify the finished message. If all of this data verifies, the server knows that it and the client share a key. It will send its own *Change Cipher Spec* message followed by an encrypted *Finished* message.

To make this more clear, I will walk through the handshake using Alice and the bank web server as an example. As shown in Figure 6-7, the bank's server starts out with a certificate and a corresponding private key. Remember, certificates contain public keys! The private key is the inverse of the public key, which is why it is written K^{-1} . What is signed with the private key is verified with the public key in the certificate.

Alice’s browser begins the handshake by sending the Client Hello. The server responds with the four messages: Server Hello, Certificate, Server Key Exchange, and Server Hello Done. As shown in the figure, Alice’s browser receives the bank’s certificate from the Certificate message and authenticates it. I will detail the authentication process in the next section. For now, just assume that the certificate authenticates correctly.

Once the certificate is authenticated, Alice’s browser *knows* (or has confidence that) the certificate is from the bank’s web server, and the public key in the certificate is the authentic public key for transactions with that server. The public key could be RSA, DSA, or ECDSA. For this example, I will say it is an RSA public key.

Alice’s browser also receives the bank’s Diffie-Hellman share in the Server Key Exchange message. Although the more correct term and the term used in Chapter 5 is Diffie-Hellman public key, the term “key share” is often used instead, possibly to avoid confusion with the RSA public key in the certificate. Either way, it is the server’s contribution to the Diffie-Hellman exchange. Refer back to Chapter 5 if you need to review how DH works.

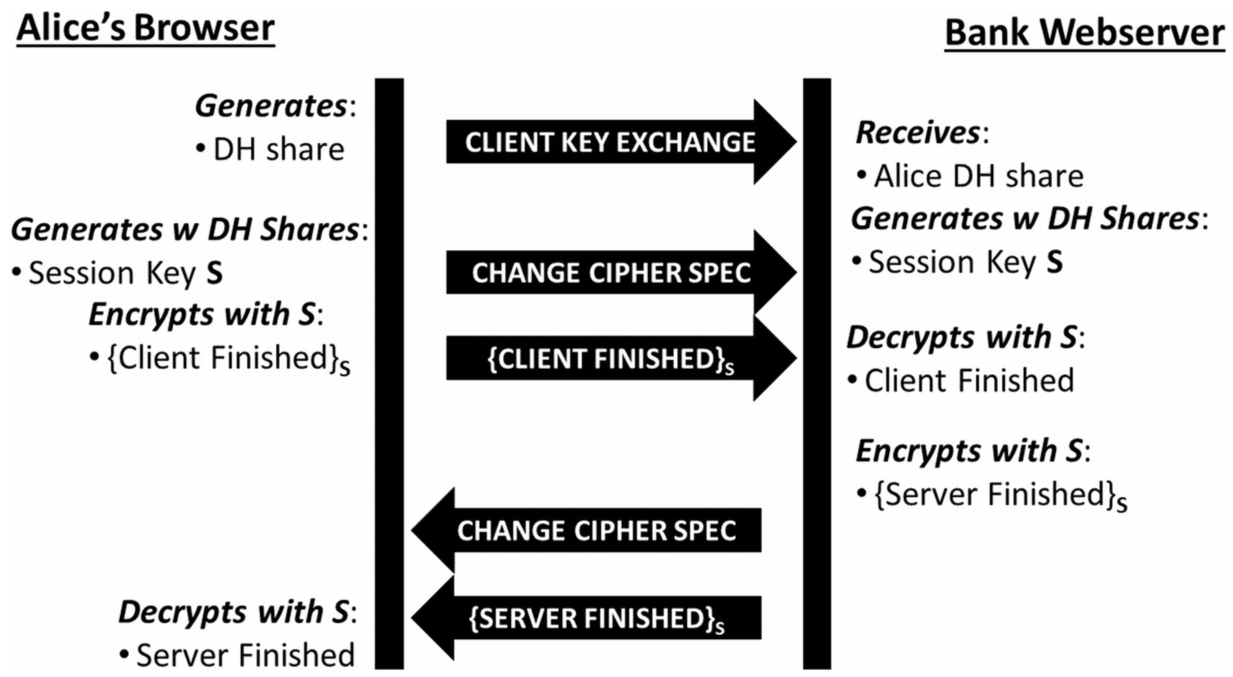


Figure 6-8 The second half of the TLS handshake between Alice's browser and the bank's web server. The browser and the server now share a session key they can use to exchange encrypted information

But Alice needs to be sure that this DH key share is *really* from the bank web server. What if a man-in-the-middle *intercepted the real DH key share and sent a fake one in its place*? Fortunately for Alice and her browser, the server *signed* the DH key share using its private key. Alice's browser uses the public key from the certificate to validate the signature on the DH key share.

So what does Alice's browser know now? It knows that it has an authentic certificate for the bank's web server. And it knows that the DH key share was signed by the private key associated with the public key in that authentic certificate. *Assuming* the private key has not been compromised, and assuming the authentication of the certificate was correct, the DH key share received *must* be from the bank's web server. It cannot be a forgery or modified by an unauthorized party.

Moving on to Figure 6-8, the next phase of TLS begins. Armed with the information gained in Figure 6-7, Alice's browser trusts the DH key share. The browser now uses its own key share (which it generated at some point during this exchange) and combines the two key shares together according to the Diffie-Hellman algorithm. This produces the session key I have labeled **S**. Alice's browser transmits the client key share in the Client Key Exchange message and then sends a Change Cipher Spec message. The Change Cipher Spec message informs the server that the next message will be encrypted.

Alice's browser generates a Finished message for itself and encrypts it using the session key **S**. This encrypted Finished message is transmitted to the server.

For the server's part, it receives the client key share in the Client Key Exchange message and accepts it without authentication. Using this DH key exchange, and its own key exchange that it sent earlier, it will derive the same session key **S**. What does the server know now? It knows it is sharing a session key with *someone*. Unlike the client, the server has done no validation yet. The server moves forward anyway. After the alerting Change Cipher Spec message, the server receives the client's encrypted Finished message. It is able to decrypt and validate this message using session key **S**.

Why does Alice's bank not authenticate Alice? Once the TLS exchange is finished, the bank still will not know that Alice is Alice. Only Alice knows (through her browser) that she is talking to the bank. The reason for this is because most servers have their own way of authenticating the client. For example, Alice will authenticate herself to the bank using a username and password and perhaps two-factor authentication. So the bank will authenticate Alice without using any authentication mechanism in TLS.

But Alice needs to *trust the bank* before she goes putting her secret password into their system, and she needs to have an encrypted channel before she sends her password over the Internet. TLS’s one-sided authentication enables a bootstrapping process whereby the bank is trusted and a secure channel is set up first. Once this is completed, the client and the server can find additional ways of authenticating the client over the secure channel.

By the end of the handshake, Alice’s browser and the bank’s web server have the keys needed to exchange the secure messages described in the previous section. I have simplified the description of the handshake to just talk about a session key **S**, but in reality, Alice’s browser and the bank’s web server use a process called *key expansion* to turn a single secret into multiple keys where one key can be used for encryption and another key can be used for the MACs. So you can think of **S** as a set of keys rather than a single key, and this set of keys is used to secure data exchange for confidentiality, authenticity, and message integrity.

All of this security, however, depends on the *authentication of the certificate*. If the certificate is not authentic, then the public key is not authentic. And if the public key is not authentic, the signature on the DH share from the server is not verifiable. And if the DH share is not verifiable, then the session key (or set of session keys) derived from the DH key exchange is not verifiable. And if those keys are not verifiable, then none of the other data can be trusted. In the next section, I will walk through how browsers authenticate the certificate in a TLS handshake.

Before moving on, one quick caution. This walk-through was meant to teach principles, not a thorough technical description of TLS. I have simplified some steps and, as previously stated, used a single configuration. It is more important to use this example to understand the concepts of building trust and sharing keys and not as a comprehensive guide to the protocol.

Certificate Authentication and PKI

What is a certificate? A certificate, like anything else stored in a computer, is just data. It generally includes a public key, the metadata related to ownership of the key, and a signature over all of the contents by a known "issuer." The metadata includes information such as the identity of the owner, the identity of the issuer, an expiration date, a serial number, and so forth. The concept is to bind the metadata, especially for identity, to the public key. The identity can be a name, an email address, a URL, or any other agreed-upon identifier.

There are various formats that can be used for certificates, but the X.509 certificate is one of the most common type of certificate used by websites. X.509's format is a collection of key/value pairs. The keys can be hierarchical. This means that a key can have subkeys. Here is the list of all keys, in their hierarchy:

1. Certificate
    - (a) Version Number
    - (b) Serial Number
    - (c) Signature Algorithm ID
    - (d) Issuer Name
    - (e) Validity period
        - i. Not Before
        - ii. Not After
    - (f) Subject name

      Subject Public Key Info

    (g) Subject Public Key Info
        i. Public Key Algorithm
        ii. Subject Public Key
    (h) Issuer Unique Identifier (optional)
    (i) Subject Unique Identifier (optional)
    (j) Extensions (optional)
2. Certificate Signature Algorithm
3. Certificate Signature

The primary purpose of a certificate is to tie a party's identity to a public key. The party associated with the public key is called the *Subject*. But how did the subject and the public key become associated together? The *Issuer* is another party that issues the certificate and asserts that the public key belongs to the Subject. The entire certificate is protected by a digital signature that is attached at the end of the certificate. The signature is created by the Issuer to prove the Issuer's approval. The X.509 fields that identify the subject, the public key, and the issuer are the most critical, but the other fields provide contextual information necessary to understand and interpret the data. For the purposes of this chapter, however, there are only two other fields that need to be discussed.

The first of these two fields is the validity period. This field is used to determine when a certificate should be considered valid. While the "Not Before" field is important and must be checked, in practice the "Not After" period usually gets the most attention. The certificate is considered invalid after the date identified by the “Not After” field, enforcing a lifespan on the public key contained therein. Certificates that are a higher risk of theft or compromise (i.e., the *private key* associated with the certificate is stolen) should have a shorter duration.

The other field I will draw your attention to is the serial number. This is a unique number (per issuer) that identifies the certificate uniquely. This serial number is useful for something called “revocation,” which is a process for invalidating a certificate after compromise (i.e., of the private key).

Returning to the more crucial fields, “Issuer Name” and “Subject Name” describe the identities claimed by the issuer and the subject, respectively. These fields have a structure and subcomponents. Called the “Distinguished Name,” these two identity fields typically have the following subfields:

1. CN: CommonName
2. OU: OrganizationalUnit
3. O: Organization
4. L: Locality
5. S: StateOrProvinceName
6. C: CountryName

So, for example, a “Subject Name” or an “Issuer Name” might look like this:

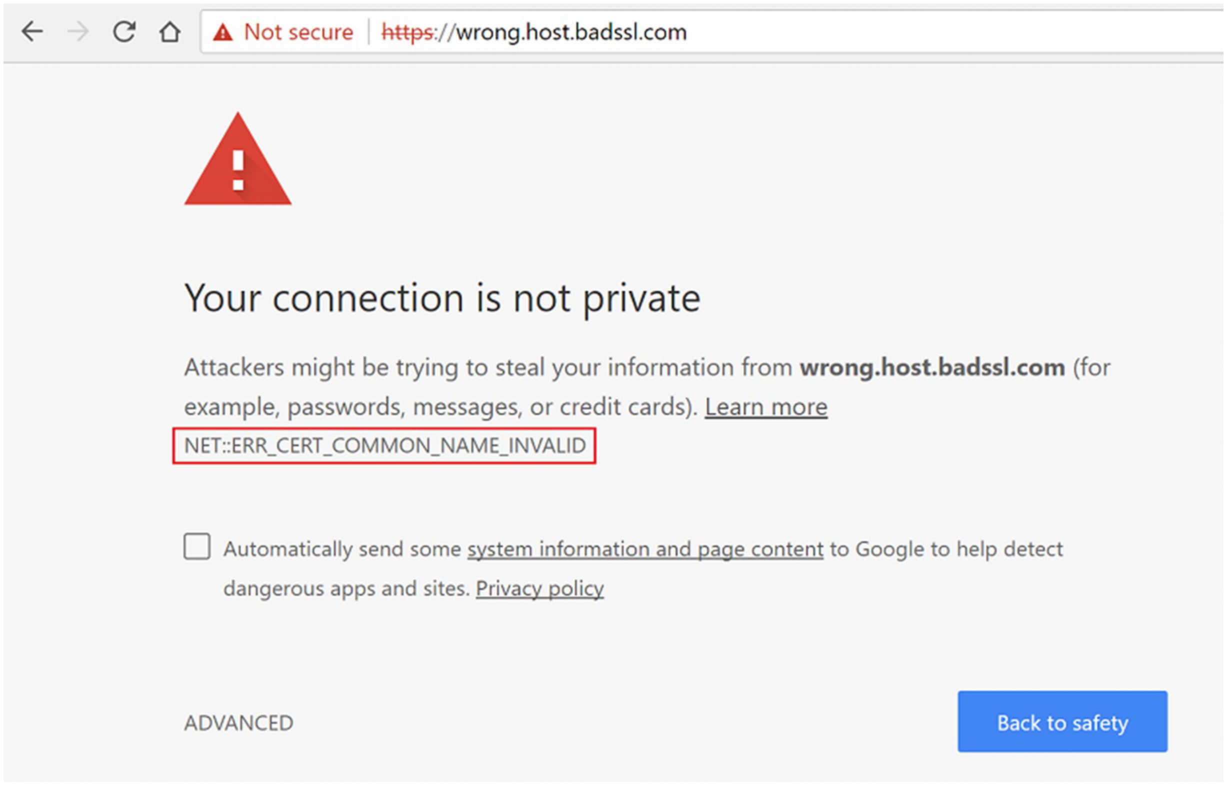
```
CN=Charlie, OU=Espionage, O=EA, L=Room 110, S=HQ, C=EA
```

Not all of these subfields have to be filled in, but CN (Common Name) is generally required and the most important. And this is a good place to start talking about how browsers validate certificates.

When Alice's browser, for example, receives the bank's certificate, the first check that it needs to make is that the certificate matches the *URL of the website*. The way that a certificate is used to prove that a website is the real website is currently done by proving that the website has a right to claim the URL that identifies it. So, if Alice had to type in "www.examplebank.com" to get to her bank's online site, the server must have a certificate for "www.examplebank.com". The browser performs this part of the validation by checking that the Common Name of the Subject Name is precisely the expected URL: "www.examplebank.com". There are a few variations of this process as some certificates put the URL in a different field called the "Subject Alternative Name," and some certificates can use wildcards like "*.examplebank.com" for any website that ends with "examplebank.com" (e.g., "www.examplebank.com," "mortgages.examplebank.com," etc.). But checking the subject's common name for an exact match is one of the more common configurations.

If the subject's name does not match the URL in the browser, the certificate is invalidated, and the browser will not show the web page. Usually, the browser indicates some kind of error like the one shown in Figure 6-9.

Additionally, the browser then checks that the current date is within the validity period (i.e., later than the "not before" date and earlier than the "not after" date). It will also check that the serial number is not on any *revocation lists*, which are published lists of revoked certificates. If either of these two checks fail, the browser will report an error. See Figures 6-10 and 6-11.


Not secure | ~~https~~://wrong.host.badssl.com

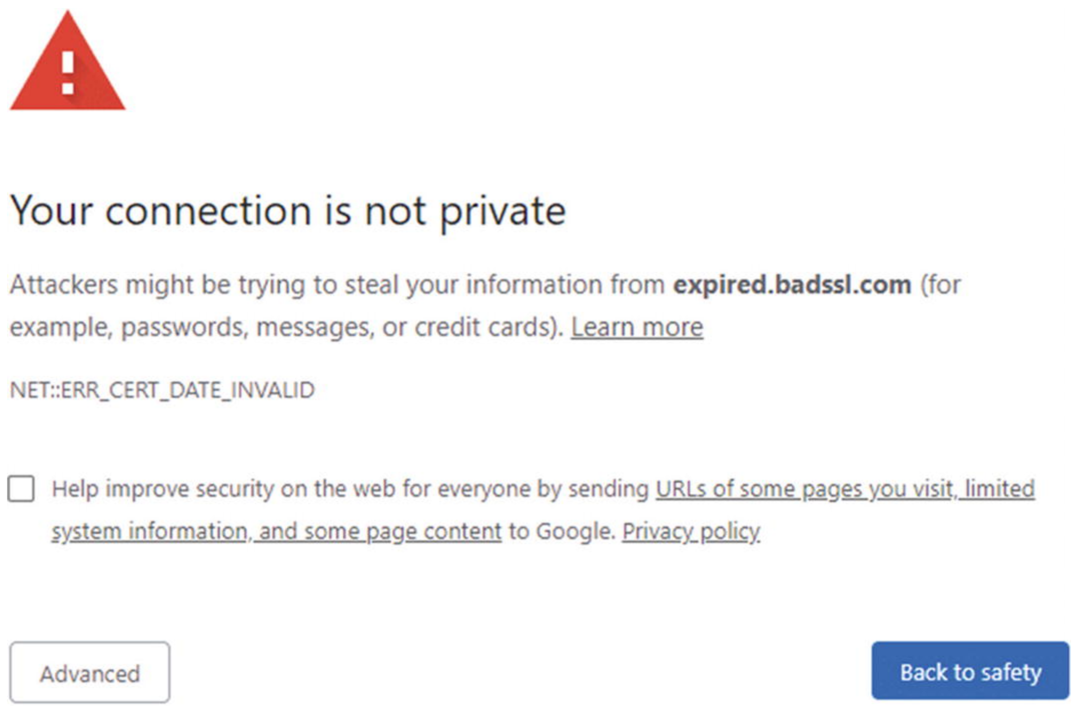
Your connection is not private

Attackers might be trying to steal your information from **wrong.host.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_COMMON_NAME_INVALID

Automatically send some system information and page content to Google to help detect dangerous apps and sites. Privacy policy

Figure 6-9 Example error message from a browser when the domain name of the certificate does not match the URL


Your connection is not private

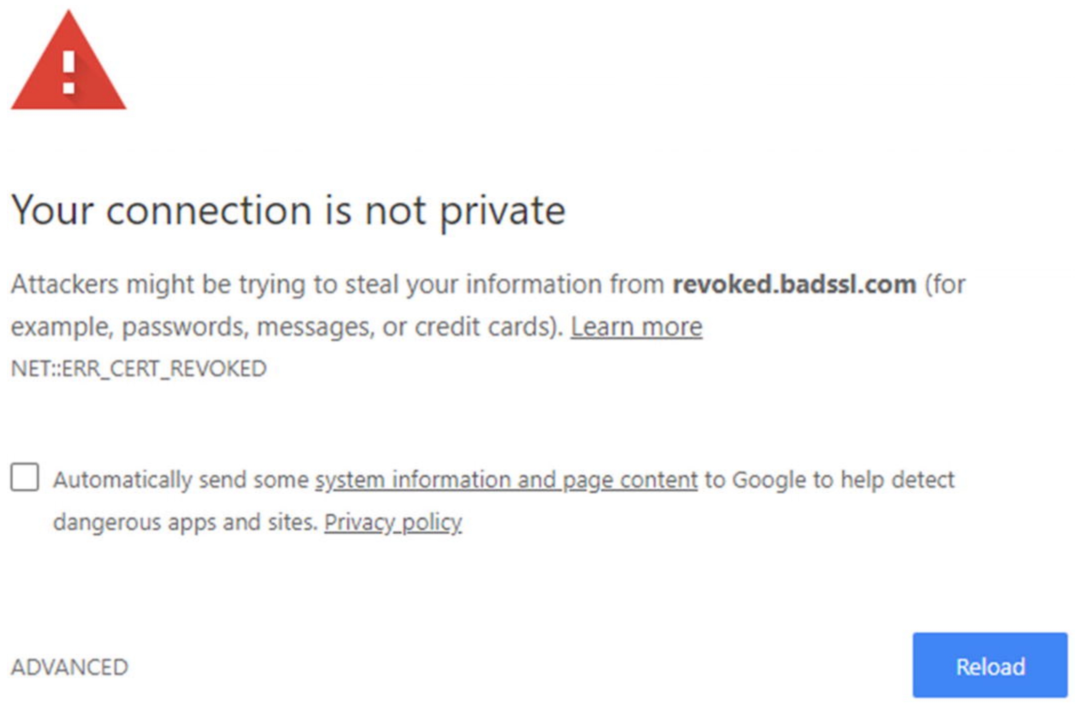
Attackers might be trying to steal your information from **expired.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_DATE_INVALID

Help improve security on the web for everyone by sending URLs of some pages you visit, limited system information, and some page content to Google. Privacy policy

Figure 6-10 Example error message from a browser when the current date is not within the validity period

All of these checks, so far, are simply to make sure that the certificate is internally consistent. At the risk of being too repetitive, anyone, however, can create a certificate. Anyone can create a certificate with a subject common name of “www.examplebank.com,” the right validity period, and a random serial number not likely to be on a revocation list. How does the browser determine that the certificate it just received over the network is the true certificate?



Your connection is not private

Attackers might be trying to steal your information from **revoked.badssl.com** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_REVOKED

☐ Automatically send some system information and page content to Google to help detect dangerous apps and sites. Privacy policy

Figure 6-11 Example error message from a browser when the certificate has been revoked

The solution is to have what is known as a *trusted third party*. The concept of a trusted third party is to have an already trusted party prove the identity of some other party. The trusted party will, in the example of Alice’s bank, prove to Alice’s browser that the certificate is valid.

Imagine backing up in time to when Alice’s bank first set up the web server. The bank wanted to be able to set up secure TLS communications for its customers. As you have seen, it would need to be able to *prove* that it was the real bank, and not some fake bank set up by thieves and ne’er-do-wells. It needed to prove that it was the real bank authorized to use the URL “www.examplebank.com.”

So, Alice’s bank set about finding an organization known as a *Certificate Authority*, or CA. A CA is a trusted party for proving the authenticity of certificates. Two of the largest CAs are *IdenTrust* and *DigiCert*. After flipping a coin (because it really does not matter which one is used), Alice’s bank chose DigiCert. Alice’s bank next generated a private and public key pair. *Keeping the private key protected and safe*, Alice’s bank sent the public key to DigiCert in what is called a *Certificate Signing Request*, or CSR. A CSR is just like a certificate, but without the signature, issuer, or serial number.

When DigiCert gets the CSR from Alice’s bank, they will do some kind of investigation to ensure that Alice’s bank is who they say they are. Depending on a number of factors, they may need to meet with a representative in person, see authorization that the individual actually sending the request is authorized to speak on the bank’s behalf, and so forth. Once satisfied that they are creating a certificate for the real bank, DigiCert adds their own identity as the Issuer to the certificate, inserts a unique serial number, and then signs the certificate request with their own private key.

To repeat, Alice’s bank *never sent the private key.* Private keys should never be shared. *EVER*. There have been stories in the news where CAs or other certificate vendors have asked for and received private keys instead of just the public key. At least one of these stories resulted in the compromise of the private keys of the customers.

Story Time: Never Share Your Private Key. No Really

In 2018, a fascinating story emerged regarding Trustico, a reseller of TLS certificates. Originally, Trustico's certificates were all based on a Certificate Authority (CA) provided by Symantec. Symantec certificates were the root of trust, but Trustico managed the customer certificate creation (signed by Symantec root certificates). Trustico wanted to *revoke* the old Symantec certificates and *reissue* all certificates under a different CA named Comodo. For security reasons, Symantec was about to lose its status as a CA because of security issues and had transferred that business to a different entity called DigiCert.

Trustico reached out to DigiCert, now in charge of the old Symantec PKI, and asked for mass revocation. DigiCert refused, arguing that only the actual certificate holder, not the intermediary that created it, could request revocation. DigiCert stated that they would only revoke the certificates in cases where the certificates were known to be compromised.

Accordingly, Trustico followed up this exchange by ***sending all their customers' private keys in an email to DigiCert***. By sending these private keys over an insecure channel (email), they effectively caused a mass compromise to force DigiCert to revoke the certificates. But this was equivalent to blowing up a building to have cause to fire the contracted security. Essentially, when they sent out the private keys of these companies, they were putting each and every one of their customers, some of whom were high security operations, in incredible risk.

More importantly, *why did they have their customers' private keys?* Only the customer should have their private key. The private key is not needed to create the certificate, and having a copy stored with Trustico increased the risk of compromise. In the end, it enabled

Trustico themselves to purposefully expose them in order to get their way in a business deal. It was noted that the private keys were not even password protected.

Moral of the story: Only the certificate's subject should have access to the private key.

Also note that Alice's bank has a certificate signed with DigiCert's private key, not with the bank's private key. The bank's private key will be used for things like signing the DH key share, and the public key in the certificate will be used to validate those signatures. But the signature on the certificate is generated with DigiCert's private key. It must be validated with DigiCert's public key.

How would a browser even have DigiCert's public key?

Operating systems are shipped with the certificates of the widely used CAs already stored inside them. Some browsers, such as Firefox, have their own store, while other browsers, such as Chrome for Windows, just use the certificates provided by the OS. These CA certificates are sometimes called *root* certificates because they are the "ground truth" for the authentication.

So, when Alice's browser receives the certificate from the bank, it identifies the Issuer as DigiCert from the Issuer field. Then, it looks in its store (either its own personal store or the operating system's store) for root certificates. If it has DigiCert's certificate, it extracts it and uses the public key in it to validate the signature on the bank certificate. If the signature matches, the certificate is presumed to be authentic.

So, in summary, Alice's browser would walk through the following steps for authenticating the browser:

- The subject's common name (or subject alternative name) matches the URL.
- The current date is within the validity period.
- The serial number is not on any revocation list.

- The Issuer’s name is found in the browser’s or OS’s root CA store.
- The Issuer’s public key (loaded from the CA store) validates the signature on the certificate.

As with many examples in this book, this example is a little simplified. Real authentication sometimes involves a *chain* of authorities. The certificate is not issued by a root authority, but by an intermediate authority. The intermediate authority is a CA that may not be in the browser’s store. But the intermediate authority is signed by a higher authority. The chain continues until it reaches a root authority. Each link in the chain must be issued and signed by the next, and the ultimate root authority must be in the browser’s list of trusted root CAs.

It is worth noting that all of the root CAs in a browser or OS’s store are *equals*. Every CA can validate every certificate. This has some interesting security implications. From a purely idealistic design perspective, this is suboptimal. This means that *any* CA compromise can result in the generation of forged certificates for *any* website.

This is *not* theoretical. CAs are compromised unfortunately often. One of the biggest compromises was the intrusion into a company called DigiNotar in 2011. An apparently Iranian hacker infiltrated the DigiNotar servers and generated fake certificates for Google and other such organizations. The intrusion was serious. It appears that the Iranian government may have been involved and used these certificates to spy on Iranians trying to use these websites.

Remember! If the certificates cannot be trusted, then *any* information received over the “secure” TLS channel can be intercepted, changed, and modified by man-in-the-middle attacks. By compromising a root CA, the hacker (potentially at the behest of the government), enabled man-in-the-middle attacks *for any website on the Internet*.

Hopefully, this is clearly describing the scope and gravity of the problem.

As bad as the DigiNotar hack was, it could have been much worse. Google had *special*, nonstandard checks in their own browser (Chrome) when connecting to Google websites. Google knows which certificates should be used for Google, and Chrome in 2011 would not accept fake Google certificates no matter who they were from. This is an example of a defensive measure called *certificate pinning*, which I will describe in just a moment. As soon as Google became aware there were fake certificates floating about, they were able to analyze them and determine they were issued by DigiNotar. This all happened relatively shortly after the hack, and it resulted in revocation of the DigiNotar root certificates. The process was not clean or easy, and it took some time to fully resolve, but it brought an end to the compromised certificates much faster than might have otherwise happened.

Because CAs are so vulnerable, and the damage from them can be so extensive, a number of additional mechanisms have been proposed to secure them. I already mentioned pinning. Pinning comes in a couple of different forms. A certificate can be pinned through a trusted distribution system. Chrome, for example, comes with Google certificates pinned within it. *If* you trusted Chrome at the time it was downloaded, the pinned Google certificates should mean that connections to Google are always authentic (or blocked). Other examples of this kind of pinning are when apps for phones or mobile devices are distributed through an app store. Some mobile apps have pinned certificates for their “home base” company servers. So long as the app store was trustworthy at the time of the download, the pinned certificates make it difficult to forge certificates for those particular servers.

There have also been more general certificate pinning solutions proposed. However, those approaches have generally been found lacking and are not widely used or supported.

Instead, the concept of *certificate transparency* (CT) has gained more interest and traction. The basic idea is in some ways similar to blockchain and distributed ledgers. Whenever a certificate is issued, it is also submitted to a public log. The public log is hosted by a third party, perhaps even the CA that issued the certificate, but it is verifiable so that the third party does not have to be trusted.

The purpose of the log is transparency: CAs are thus essentially audited for the certificates they produce. The goal is to have all issued certificates publicly available for inspection in a cryptographically verifiable way.[10] Browsers are now beginning to support being configured to not accept any certificate that is not found in such a log.

What do we get from using CT logs? It’s deceptively simple but surprisingly helpful. Suppose that an attacker attempts to create a fake certificate to Alice’s bank’s web server. If browsers will not accept the cert unless it is published, the attacker will have to submit it to one of the public logs. If that happens, Alice’s bank can immediately detect that a forged certificate has been generated. While this does require that the bank monitor the logs, it is easy to deploy an automated system that checks to see if any new certificates have been issued that shouldn’t have been. The bank knows (or should know) which certs it has legitimately issued and can flag ones that aren’t.

Even if the attacker is so clever as to somehow interfere with the bank’s auditing system and does manage to get away with some subterfuge, once the attack is detected, the public logs will enable a thorough investigation of the problem and an accurate assessment of the damage. It is terrifying that in the DigiNotar hack, investigators were unable to even fully identify all the certificates that had been generated! To this day, *nobody knows* exactly how many certificates the attacker created. That is one reason why DigiNotar had to completely shut down. It was impossible to identify all of the certificates that needed to be revoked.

CT is still relatively new, just having reached version 2.0 at the time of this writing. But it is growing in support and seen as a valuable solution to an otherwise thorny problem.

Securing Storage: IEEE Standard 1619.1

There are a number of different technologies that provide storage encryption functionality. Both Windows and MacOS offer full-disk encryption as a built-in option. There are various third-party systems, including TrueCrypt and VeraCrypt. Each has different emphases and focuses on different benefits.

However, my goal in this section is to discuss the cryptographic system. For that, I chose to use the standard created by IEEE numbered 1619.1 [15]. I chose this standard because I felt it was generic and illustrated a lot of key ideas about cryptographic systems. Most other storage encryption technologies will do something like this although each may take a slightly different approach. For example, the 1619.1 standard ensures the integrity of the data, while VeraCrypt explicitly does not [273].

Managing the encryption and decryption of data at rest introduces a number of interesting challenges. Unlike data being transmitted, data at rest may be modified. This introduces performance problems as well as security concerns.

With respect to performance, the crucial question is how much data must be reencrypted when a change is made. This depends on many parameters. But suppose we performed a naive storage encryption system that encrypted whole files with AES-GCM mode using a single key and nonce. Any change to the file, no matter how small, would require a complete reencryption of the entire file. Such operations would be hardly scalable for a large file subject to frequent updates.

Security concerns can also happen when encrypted data is being mutated. As discussed in Chapter 5, a key and IV pair (or key and nonce pair) should never be “reused.” But what does “reused” even mean? It means literally any use on different data, but it may not be obvious to everyone that changed data *is* different data. So, suppose a system used the same key and IV to encrypt a file and then encrypt changes to the file. That would constitute reusing the same key and IV pair. This could potentially leak data or possibly break the security of the system altogether.

Another interesting question is whether or not the MITM attacker is just an eavesdropper or can alter data. That is, if a laptop is stolen, chances are very low that the attacker will send it back in order to trick the laptop’s owner into accepting fake data. In that case, integrity is much less important than confidentiality (and this is why VeraCrypt just does confidentiality).

On the other hand, if data is stored in an environment where it may pass through various hands before being accessed again, integrity is much more important.

The IEEE 1619.1 standard is entitled *IEEE Standard for Authenticated Encryption with Length Expansion for Storage Devices*. This standard specifies an approach to encrypting storage data combining both confidentiality and integrity. In fact, one particular scenario was *tape* drives, which are usually used for backup and storage. As you can imagine, a tape could be intercepted and the data altered, so integrity is important.

The words “Length Expansion” are also important with respect to the encryption of data at rest. This standard is explicit that it is only useful in environments where the encrypted length can be greater than the plaintext length. If you think about it, storage devices have to be careful with encryption that takes up more space. The 1619.1 specification explicitly requires that the protected data be allowed to take up more space than the original. This is not because of the encryption algorithm per se, as all of the encryption algorithms used produce the same size ciphertext as the plaintext. However, there is additional space required for metadata and the authentication data used for providing integrity.

In the following sections, I will walk through the cryptographic operations of the 1619.1 standard. As with my explanation for TLS, I will start in the middle with the actual encryption of data and then work backward.

Bulk Encryption of Storage Data

The 1619.1 standard performs bulk encryption on storage data chunks called “host records.” The standard does not require that the host record be any specific type or amount of storage data, but it suggests that it could be a fixed-size chunk of a file. However, the encryption component can be implemented to define an upper bound for the size of the host record. A host record can be split into one or more “plaintext records” that are subsequently encrypted to produce “ciphertext records” which are part of “encrypted records.” Figure 6-12 illustrates these relationships.

The encryption algorithms supported by the 1619.1 standard include

1. AES-GCM

2. AES-CCM
3. AES-CBC with HMAC
4. AES-XTS with HMAC

You should recognize both AES-GCM and AES-CBC from Chapter 5. As a reminder, AES-GCM is AEAD and has its own built-in MAC. AES-CCM is another AEAD algorithm. On the other hand, AES-CBC and AES-XTS are not AEAD and HMAC is added.

The support for multiple algorithms is similar to what TLS does, although in TLS this is complicated by requiring a client and server to mutually agree on an implementation that they both support. When performing the encryption of data in storage, this is less of an issue. Accordingly, an implementation can be designed to support one or more algorithms based on how well the algorithms support the particular operational context. For example, the XTS mode of operation is supposed to be particularly high performance for storage encryption. A vendor may decide to use that algorithm because of software or hardware support already available in the target systems. Another reason for enabling multiple modes of operation is that if a vulnerability is discovered in a particular mode, the product remains useful using another mode (although all the data would have to be reencrypted).

The 1619.1 standard, as mentioned previously, requires that the target system permit the output encrypted records to be longer than the plaintext. One reason for this is that the encrypted records may include additional unencrypted data called “additional authenticated data” (AAD). Within this standard, AAD is more or less like the “additional data” in Authenticated Encryption with Additional Data. Because only two of the supported algorithms are AEAD, it identifies this data explicitly. The 1619.1 standard uses the AAD, which only needs to be authenticated but not encrypted, for metadata such as information about the original host record and where this particular chunk fits into it.

When a plaintext record is encrypted, the inputs to the algorithm are

1. A secret cipher key
2. An initialization vector (IV)
3. Length of the IV or nonce
4. Plaintext record
5. Length of the plaintext record
6. Additional authenticated data (AAD)
7. Length of the AAD

The outputs of the algorithm are

1. A ciphertext record
2. A Message Authentication Code (MAC)
3. Optionally, the IV or enough information to reconstruct the IV
4. Optionally, the AAD or enough information to reconstruct the AAD

The standard specifies that the ciphertext record must be the same size as the plaintext record. The encrypted record contains the ciphertext record and MAC. It also contains the IV and AAD or enough information to reconstruct them. This option is made available because sometimes an IV or additional metadata (i.e., in the AAD) can be determined partially from context. There is no reason to store information that can be reconstructed during the decryption process.

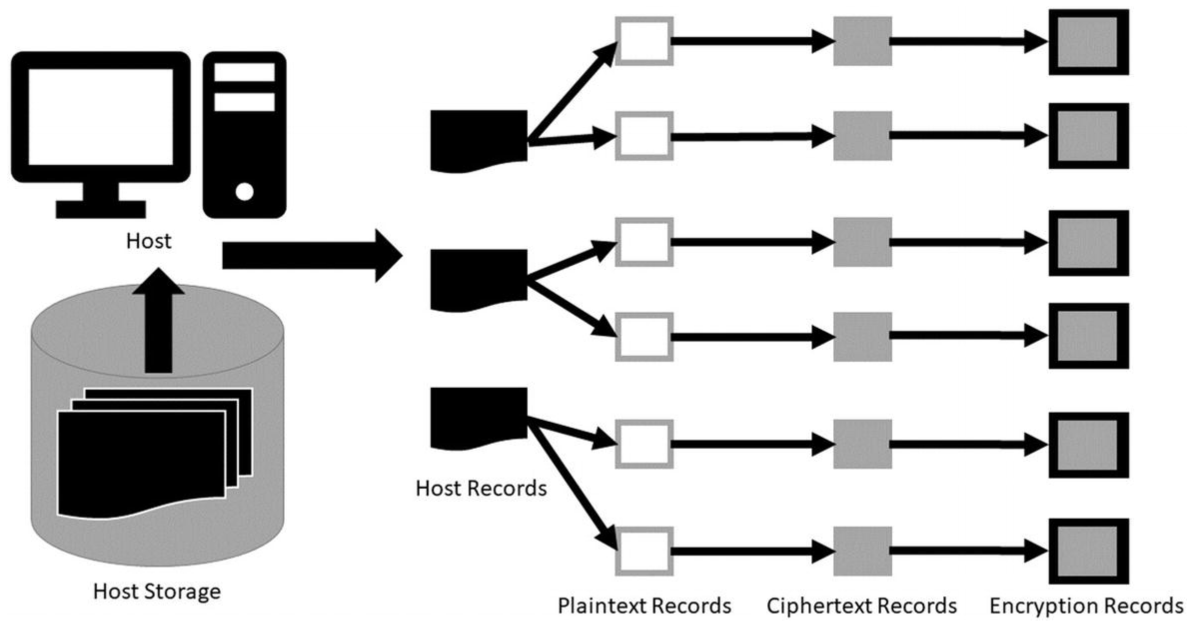


Figure 6-12 In the IEEE 1619.1 protocol, data is extracted from the host as host records. Each host record can be split into multiple plaintext records. Each of these is encrypted into same-size ciphertext records. Metadata is combined with the ciphertext record to become encryption records

Notably, the standard explicitly forbids the *cryptographic key* or any of the *plaintext* from being written to the storage medium. However, the key may be stored if it is, itself, encrypted by another key. A key used to encrypt a key is unimaginatively called a *Key Encrypting Key* or KEK. This will be discussed in greater detail in the next section.

In another notable requirement, the standard requires that the system associate a key with precisely *one* cryptographic mode, meaning one of the four supported modes of operation like AES-GCM. That is, the same key can be used with multiple encipherments using the same mode, but must *never* be used for multiple encipherments using multiple modes (e.g., one encipherment using AES-GCM and one using AES-CCM). This requirement prevents data leakage that sometimes happens when a key gets used in different algorithms. Sometimes, information can be exposed from the key being used in two different ways.

As also explained in Chapter 5, the same key and IV pair can never be used for encrypting two different inputs. An entire section of the standard is dedicated to preventing a key-IV pair from being used twice, which it calls an *IV collision*. One of the approaches is simply to use a completely random IV each time, wherein the IV is generated by a sufficiently secure random number generator. As a completely random number, the odds of being used twice with the same key are vanishingly small.

Another option pointed out by the specification is to use a different, random *key* for every encryption operation. This would typically require extra storage space because the key would have to be stored (encrypted by the KEK) with every encrypted record. On the other hand, a completely random IV would also have to be stored with every record but would not need to be encrypted.

The standard describes other approaches to these kinds of collisions as well, and the list is not exhaustive. Vendors are free to pick a solution that works for their particular needs, but they do need to make sure they get it right. Repeated use of key-IV pairs is a quick way to a compromise.

Story Time: That's... Not Good

Basic cryptography mistakes such as reusing keys and IVs are super basic. Every student in an intro to cryptography class learns this. It is probably in the top ten most commonly discussed requirements for symmetric cryptography. Maybe even one of the top five.

But that does not prevent mistakes, even from really bright people. In 2022, it was revealed that Samsung's implementation of *TrustZone*, which is the part of the phone that is supposed to be the most secure and impenetrable, had a major flaw. Matt Green, the well-known cryptographer, referred to it as "embarrassing" [270].

The flaw was, in fact, using a key repeatedly while permitting IV reuse. Security researchers demonstrated that the system could be attacked in practice. In the overview of their findings, they said:

> We present an IV reuse attack on AESGCM that allows an attacker to extract hardware-protected key material, and a downgrade attack that makes even the latest Samsung devices vulnerable to the IV reuse attack. We demonstrate working key extraction attacks on the latest devices. We also show the implications of our attacks on two higher-level cryptographic protocols between the TrustZone and a remote server: we demonstrate a working FIDO2 WebAuthn login bypass and a compromise of Google's Secure Key Import. [240]

According to reports, this affected more than 100 million phones [270]. Yikes.

So how does something like this happen? Apparently, the system permitted application-level code outside of TrustZone to pick the IV. This meant that TrustZone was using a single key, then allowing applications, which are *not trusted* to pick IVs [270].

The decryption operation is, of course, reversed. The ciphertext record is decrypted and the MAC checked. The

AAD data may be needed to figure out details of putting together the decrypted data back into the original host record expected by the host system. From the perspective of the host system, host records go in, host records come out.

Key Life Cycle Management

Much like TLS, the actual encryption and decryption process is not too complicated. Yes, certain rules must be followed like a unique key-IV pair, but at the end of the day, data gets encrypted and data gets decrypted. The hard part is getting the keys. In TLS, there is an entire process of having both client and server agree on a master key and having both be able to derive all of the necessary component keys. Diffie-Hellman is a common approach, supported by PKI, to ensure the authenticity of the key agreement.

For the 1619.1 standard, there is no client and server. However, although the person encrypting the stored data and the person decrypting the stored data may be the same person and in the same location, the two operations are separated by *time*. Where has the key been stored during this period? Who has had access to it? If the key has been *lost*, the data is completely unrecoverable!

Also like TLS, the 1619.1 standard has to deal with multiple keys. It may be desirable, as mentioned in the previous section, to use a different key for every encryption. There could be millions of keys at any given time. It is also true that no one key should be used to encrypt more than a certain amount of data. The reasons for this are beyond the scope of this book, but suffice it to say that after the same key has been used to encrypt more than a certain amount of data, there are risks to data leakage and exposure. For example, AES-GCM limits a key to being used more than 4,294,967,296 times when paired with completely random IVs. That number may sound like a lot, but imagine if the 1619.1 standard were being used to encrypt data in an on-the-fly encryption system where files were encrypted and decrypted as they were being used. Every operation would result in a new encryption and that could happen thousands of times a day.

Unlike TLS, the 1619.1 standard does not fully specify how all key management works and instead defers details to implementations. However, it does define two components that work together with respect to keys: the cryptographic unit and the key manager. The cryptographic unit is the component that performs the encryption of the plaintext records as discussed in the previous section. The key manager is tasked with managing the life cycle of keys used by the cryptographic unit, where life cycle includes generation, archiving, and destruction.

Recall that the cryptographic unit requires a cipher key for the encryption of a given record. The standard indicates two[11] different ways that it can obtain a key for this operation. First, it can generate a random key internally (without the key manager) using a sufficiently secure random number generator. Alternatively, it can receive a key from the key manager and ensure that this key is paired with a unique IV.

If it generates its own key for encrypting a record, the key must be preserved. The standard requires that the cryptographic unit encrypt this cipher key using a KEK. It can then either store the wrapped (encrypted) key with the encrypted data in the storage medium, or it can send the wrapped (encrypted) key to the key manager for storage. In the latter case, it would also have to pass along enough metadata that the key can be looked up and retrieved when decryption is needed. It is worth noting that the KEK is also meant to come from the key manager, so even when generating a cipher key the cryptographic unit relies on the key manager for storing them.

On the other hand, if the cryptographic unit receives the key from the key manager, it is expected that it will not store the key on the storage medium. Instead, it should receive the key from the key manager again when performing the decrypt operation.

The standard does not describe any specific expectations for the key manager or how it should manage the key life cycle. While this might seem like punting the hard problems to some other component, it is not unreasonable. There are a wide range of key managers and key management techniques. Some use hardware, such as hardware security modules (HSMs), while others use a straightforward software module.

Because the standard does not address the construction of or requirements for the key manager, it also punts on the other major problem: the master key. It identifies that keys may be generated by the cryptographic unit, but the KEKs that protect them (i.e., for storage until subsequent decryption) come from the key manager. And where does the key manager get those? That is left up to the implementation of the key manager and could include any of the authentication techniques discussed in Chapter 2.

The 1619.1 standard does take a dim view of deriving keys from passwords. This is an approach taken by a number of different storage encryption systems such as VeraCrypt [274]. This concept is that a key can be derived from a password through a process kind of like a hash. That is, the password can be converted using a one-way (irreversible) function into the proper number of bits for a symmetric key. This could be used in a 1619.1 key manager, for example, to create a KEK (or a master key for a series of KEKs). Thus, the security of the data stored on disk would all be tied to the password. The cryptographic unit could generate its own keys for each record, then store these on disk wrapped using the KEK key derived from the password. To decrypt, the user would put in the same password, regenerate the same KEKs, and decrypt the keys stored on the storage medium to decrypt the encrypted records. This process is visualized in Figure 6-13.

But, I repeat, the 1619.1 standard does not look on this kind of solution with approval. According to the standard, it is “relatively easy for an attacker to launch an off-line dictionary attack” [15]. While that may be true, this problem is, of course, solved by using a sufficiently challenging password.

But whether a password is used or some other authentication, there is an increasing risk of the data being permanently unrecoverable as time goes on. The password can be lost or forgotten, but a rarely used system (e.g., tape backup) that relies on biometrics might find that, when needed, the employees with the biometric access are no longer with the company. As always, keys are the most crucial part of a cryptographic system but also the most fragile.

Summary

Designing a successful cryptographic system depends on properly combining many different components to achieve the goals of the system.

The guarantees we want from a cryptographic system usually center around keeping our data away from prying eyes of those who shouldn’t read it (confidentiality), knowing that our data hasn’t been modified after the sender encrypted it (integrity), and trusting that the party we’re communicating with is who we expect (authenticity). We need to know that our communications have not been intercepted (i.e., if an attacker reads them, they only see encrypted data and can’t learn the plaintext) or modified (i.e., if an attacker attempts to modify the data, we can recognize the tampering and deem it invalid). And we need to know that we can trust our counterparty—both that they are who they claim to be and that they are who we expect them to be.

This book surveys many cryptographic systems (e.g., ransomware in Chapter 7, OAuth in Chapter 9, and email security in Chapter 10), and this chapter serves as both an introduction and a point of reference for you later when we get to them. Here, we discussed two example designs for data-in-motion (HTTP and TLS for secure web communication) and data-at-rest (IEEE 1619.1 for encrypted data storage) cryptographic systems.

When Alice and her bank began their secure web communication session, they needed to negotiate shared keys to encrypt the contents of their communications, which is the central focus of the TLS handshake. Since the process occurs over the open Internet, the bank needs a way of publicly proving its identity. To do this, Alice can rely on the certificate supplied by the bank. Alice can verify that the certificate is legitimate through the public key infrastructure, a chain of trust starting from a Certificate Authority and proceeding down to the certificate from her bank's website. Once the encryption keys are properly created, both parties can rely on the protocol to provide confidentiality and integrity.

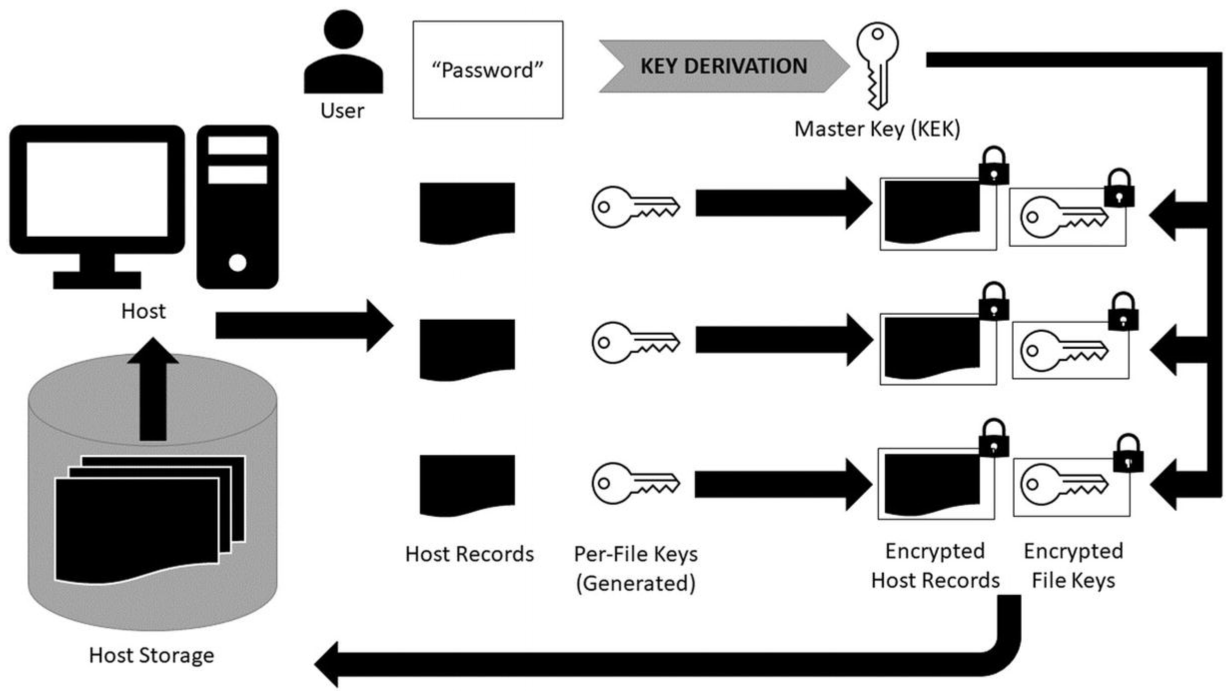


Figure 6-13 A basic approach to file encryption using a password. The password is derived into a master key that serves as a Key Encrypting Key (KEK). Each file or chunk is encrypted with a generated key, and each generated key is encrypted by the KEK. The encrypted file and the encrypted key are stored on disk

For file storage, the two parties may be the same entity, separated by time, and we want to ensure that any data stored has the properties of confidentiality, integrity, and authenticity. We need to know that the data has remained secret, even if an attacker obtained the stored data. We need to know that the data has not been modified since we stored it. And we need to know that the person storing the data and the person retrieving the data are authorized to do so (e.g., through properly storing and protecting the encryption keys, perhaps with a key manager).

When these components are properly combined, we get cryptographic systems that are robust to man-in-the-middle attacks and provide the assurances we need to safely and securely store and transmit data.

Further Reading

The TLS standard is defined by the Internet Engineering Task Force (IETF) in documents called Request for Comments (RFCs). You can read the core TLS 1.2 specification in RFC 5246 [216] and the TLS 1.3 specification in RFC 8446 [215]. If nothing else, each one of these two RFCs has a security overview of the protocol. You can also go back in time to the mid-1990s and read one of the first specifications for the original SSL protocol [101].

Anderson's book *Security Engineering* dedicates several chapters that might be helpful to the reader. Chapter 4 discusses protocols at a conceptual level and some of the basic concepts behind a number of actual systems as well [40, Chapter 4]. One famous crypto system not covered in this book is called Kerberos, and this chapter of Anderson covers it. Chapter 20 goes into many systems including Signal (and similar competitors such as WhatsApp), The Onion Router (TOR), blockchain, and more [40, Chapter 20].

Another reference cited often in this book is Bishop's textbook. Chapter 12 includes a number of example protocols including TLS. It gets into some aspects of TLS that I did not cover, such as the extension for a heartbeat that led to the so-called *HeartBleed* vulnerability. Bishop also covered design principles extensively and explicitly [60, Chapter 12].

TLS 1.2 is discussed in significant detail by Stallings' book on cryptography. Written in 2013, it predates 1.3 [250].

With the change to cloud computing, plus other technologies such as containers (e.g., Kubernetes), there is also a lot of discussion around securing machine-to-machine connections. Many references about cloud security will discuss using TLS, and even mutual TLS (where both sides confirm the identity of the other), within these contexts [95, 217, 272].

TLS emerged in the mid-1990s. TLS, as discussed in this chapter, relies on Certificate Authorities for its PKI. But before TLS, there were other proposed PKI designs. Many of these centered around the concept of an *online* repository of public keys. A computer would contact the online repository to get the key for any party it wanted to talk to. The repository's public key would be published *everywhere*, such as newspapers, books, magazines, advertisements, etc., so that it would be difficult to steal. Davies's introduction to network security (written in the 1980s) is built around this concept [91].

References

15. IEEE standard for authenticated encryption with length expansion for storage devices, 2019.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

91. Davies, D.W., and W.L. Price. 1984. *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer*. New York: Wiley.

95. Dotzon, C. 2019. *Practical Cloud Security: A Guide for Secure Design and Deployment*. Sebastopol: O'Reilly Media.

101. Elgamal, D.T., and K.E. Hickman. 1995. The SSL Protocol. Internet-Draft draft-hickman-netscape-ssl-00, Internet Engineering Task Force, Work in Progress.

215. Rescorla, E. 2018. The Transport Layer Security (TLS) Protocol Version 1.3 (8446).

216. Rescorla, E., and T. Dierks. 2008. The Transport Layer Security (TLS)

Protocol Version 1.2 (5246).

217. Rice, L. 2020. *Container Security: Fundamental Technology Concepts That Protect Containerized Applications*. Sebastopol: O'Reilly Media.

240. Shakevsky, A., E. Ronen, and A. Wool. 2022. Trust dies in darkness: Shedding light on Samsung's TrustZone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, 251–268. USENIX Association.

250. Stallings, W. 2013. *Cryptography and Network Security: Principles and Practice*, 6th ed. Prentice Hall Press.

270. Vaas, L. 2022. Samsung shattered encryption on 100m phones.

272. Vehent, J. 2018. *Securing DevOps: Security in the Cloud*. Shelter Island/New York: Manning Publications Co.

273. VeraCrypt. VeraCrypt documentation: Authenticity and integrity.

274. VeraCrypt. VeraCrypt documentation: Header key derivation, salt, and iteration count.

Footnotes

1 Note: This acronym will mean different things in different chapters, so pay close attention to context.

2 The full specifications of the 1619.1 standard are only available by purchase from IEEE. I will not be providing the specifications but, instead, will talk about certain principles found in their approach to storage encryption.

3 A third state of data is *data in use*, but data in this form, and the protections thereof, are outside the scope of this book.

4 Recall from Chapter 5 that as more ciphertext is created, there are more opportunities for finding patterns. Although modern ciphers are exceptionally strong and can produce a lot of ciphertext before they become at risk, they have limits.

5 There are reasons Bob may wish to do this. It has to do with a concept of *repudiation* for which there is not the space to get into here.

6 Old habits die hard. Many times, the term “SSL” is still used, even when talking about TLS. Certificates, for example, are still often referred to as SSL certificates even if they are only used for TLS.

7 The MAC also protects certain TLS data in the header, but this is a little more complicated and not discussed here.

8 Actually, there are some weaknesses with this approach of encrypting the MAC. This method is called *MAC-Then-Encrypt* because the MAC is generated first over the plaintext and then encrypted with the plaintext. Cryptographers figured out later that it is generally better to encrypt the plaintext and then create an unencrypted MAC over the ciphertext. This approach, called *Encrypt-Then-MAC*, ensures that the attacker cannot be messing around even with the ciphertext.

9 In common practice, only the *server’s* identity is verified, though there are increasing use cases for “mutual TLS” (MTLS), where the client verifies the server, and the server also verifies the client.

10 The name of one of the original projects behind this was called “Sunshine” and was started after the DigiNotar hack.

11 It actually specifies three different methods. However, two of them are similar and are differentiated only in scope. For simplicity, I have reduced the three options to two.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_7

7. Host Security Technology

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
Digging into a more concrete topic, this chapter covers both attacks and defenses on host computer systems. The first half of the chapter focuses on building a robust system using *isolation* and *access control* through operating system design, hardware enforcement and software enforcement. The second half of the chapter digs into attacks on these systems through exploitation of design flaws and/or malicious software.

Key Concepts

1. A program is a set of instructions for computer hardware.
2. An operating system (OS) is a program that controls all other programs' access to the hardware.
3. The OS *isolates* programs from the hardware and each other.
4. Many modern systems have software and hardware that enforce security on the OS.
5. Software defects like buffer overflow can enable an

attacker to take over the program.

6. Attackers also create malicious software like viruses, trojans, ransomware, and so forth.
7. There will never be perfect malware detection/prevention.
8. Anti-malware approaches include *identify and neutralize*, *mitigate*, and *recover and respond*.

Common Pitfalls and Misunderstandings

1. Buffer overflow and related issues can be hard to understand if you did not read the control flow sections carefully.
2. The term “virus” used to mean a specific kind of malware but now is often used to mean any kind of malware.

Useful Vocabulary

- **Trusted Execution Environment**: A processing mode enabled by hardware that allows loading certain sensitive information (e.g., cryptographic keys) or running sensitive operations
- **Buffer Overflow**: Writing data past the end of a specified section of memory, which may overwrite other data and enable an attacker to gain access to the system
- **Return-Oriented Programming**: An attack technique of constructing malicious programs out of segments of an existing program, similar to constructing a ransom note out of magazine clippings
- **Virus**: Malicious software that attaches itself to legitimate programs in order to cause damage or

propagate
- **Worm**: Malicious software, usually self-contained, that seeks to propagate across a network
- **Trojan horse**: Malicious software that masquerades as legitimate or desirable software
- **Rootkit**: Malicious software that installs itself into a user's system and attempts to gain elevated system privileges
- **Ransomware**: Malicious software that encrypts data with an attacker's key and demands a ransom in exchange for decrypting the data
- **Botnet**: A network of compromised computers that can be remotely controlled by an attacker in order to stage large-scale attacks, such as distributed denial of service (DDOS)

Although the topics covered in the previous chapters are technologies in their own right, they are primarily components that can be put together in larger or more specialized systems. In this chapter, I will introduce technologies designed to protect host systems. Many of these technologies depend, at least in part, on these more foundational technologies.

The term "host" is a generic name for any kind of individual system. Other synonyms for host, or system, include "node," "endpoint," or even "device." No matter the term, the idea is basically some kind of independent processing and operations. It would include the desktop computers someone might use at home or the office, IoT devices, or large server machines. For a large part of computer history, these were almost always some kind of physical system, but in modern systems, it is as likely as not that a "device" is *virtual*, that is, a *pretend* computer that is simulated on a real (physical) machine.

One reason for the various terms (host, node, etc.) is because these systems are used in so many different contexts and configurations. *Node*, for example, is a term more commonly used where each component is more or less equal to all the others either in function, behavior, capacity, or some other relevant metric. On the other hand, *endpoint* almost always refers to a final destination of data or function such as a user's personal computing device. It differentiates the endpoint system from all the other systems that provide services to, enable, or otherwise assist it.

Although there may be subtle differences between these various terms, and there may be circumstances where distinctions matter, for the purposes of this chapter, I will assume that they are all equivalent. And, for simplicity, I will use the term "host" throughout. It may be helpful to review some of the basics of a computer's major components and how they work together in Appendix B.

Host security technology is focused on protecting individual computing systems from subversion. Common security goals include *confidentiality*, *integrity*, and *availability*. I first introduced these terms in Chapter 3 about authorization technologies. Those technologies are focused on the proper *assignment* and *management* of permissions related to these goals. In this chapter, the technology is designed to *enforce* the permissions (or lack thereof) assigned to parties accessing the systems. These technologies are sometimes called *controls*. For example, NIST refers to both *security controls* and *privacy controls* this way:

> Security controls are the safeguards or countermeasures employed within a system or an organization to protect the confidentiality, integrity, and availability of the system and its information and

> to manage information security risk. Privacy controls are the administrative, technical, and physical safeguards employed within a system or an organization to manage privacy risks and to ensure compliance with applicable privacy requirements. [121]

For simplicity, I will simply refer to “controls” and not distinguish them as to whether they are security or privacy focused.

Of the many ways they try to achieve their nefarious goals, attackers look for ways to bypass, disable, or abuse security controls in order to obtain unauthorized permissions. Unfortunately, they are successful far too often. Because of this, host security controls are layered so that a failure in one control is mitigated in whole or in part.

Host Security Fundamentals

Effective computer security is almost always based on a foundational design concept or principle that has demonstrated wide application and solid security outcomes. For example, in Chapter 2 I introduced the design principle of *defense in depth*. Or, in Chapter 3, you learned about *least privilege* in security design. Chapter 4 discussed *the principle of open design*. Learning security design principles is crucial to understanding security technologies and being able to evaluate their effectiveness.

One of the most important design concepts for host security is *isolation*. As applied to hosts, isolation generally refers to ensuring that every resource is separated and protected from all other resources with all accesses mediated.[1] The term resource includes, of course, things like files, hardware, printers, cameras, WiFi, and video devices. But it also includes running computer programs. In fact, because computer programs are active resources that change system state, access other resources, make decisions, and act on behalf of system users, many isolation concepts and mechanisms are directed specifically to running programs. Because I will be talking about running programs throughout this chapter, I will use the term *process* to describe the running instructions with its associated state as opposed to the program in some kind of stored and inert form. A big part of host security comes from isolating processes from each other and from other resources on the system.

Most people without technical computer training know operating systems, such as Windows and MacOS, as *user interfaces*. Operating systems have an outward appearance, a look and feel that is the most visible (and most marketed) part of the system. However important and useful all of these features are, they are not actually the core of the operating system (OS), nor are they the most important. Beyond “looking pretty,” operating systems provide crucial management of system resources, including processes, to enable stability and performance. One key function of the OS is to isolate processes from each other and from any direct access to system resources.

Operating Systems and Isolation

The technical details of how a computer CPU runs a program discussed in Appendix B are important to understanding how an operating system isolates a program and why this matters. Again, a review of that appendix may be useful to you before starting this section. What is described in that appendix is how a process would work if it was running “raw” on the CPU with no intermediary operating system. But if processes could have direct access to the CPU and the RAM, it would cause problems with performance and stability.

The first major problem is that a CPU can only execute instructions from one process at a time. Computers *appear* to run multiple processes at the same time by doing *time slicing*. The concept is to execute instructions from one process for a slice of time, then switch and execute instructions from another process in the next slice of time. There is no way to do time slicing without some kind of controlling mechanism. And that is where the operating system comes in.

An operating system is a special kind of computer program that manages the overall operations of the entire computer including the CPU and the memory. No process can get started without the operating system's permission, and the execution of the process's instructions is managed by the operating system. The operating system, for example, controls the time slicing that permits multiple processes to execute (what appears to the user to be) simultaneously.

The operating system *forces* a process to only run for the time slice. The process cannot prevent the OS from interrupting it. This is primarily good for performance, but it also has computer security properties. As discussed at the beginning of the chapter, one of the goals for host security is *availability*. This means that resources *should* be available to authorized parties. Unauthorized parties should not be able to deny the authorized parties access to their resources.

By *isolating* the processor from the running process, there is no way for a process controlled by an attacker to take over the processor. Imagine, for example, if an attacker started a program that did nothing useful but *ran forever*. If there were no operating system to forcibly interrupt the process, no other process could run until the computer was reset and restarted. An attacker could make a computer completely unusable by simply starting a program that would never exit.

To be able to enforce these kinds of controls, software and hardware use *privilege levels*. There can be multiple levels, and some common models use four levels. But for the purposes of this chapter, I will only focus on two levels and will refer to them as privileged and unprivileged. Processors, for example, have certain hardware and operations that can only be used in a privileged mode. The privileged hardware includes special, privileged registers (as a reminder, review Appendix B for an explanation of registers). The privileged registers hold configuration data that is very sensitive to system operation and security. The privileged instructions configure or enable security-sensitive operations including controls related to isolating processes.

When the operating system starts after the computer is turned on, it starts in privileged mode and configures these security settings. Once the sensitive parts of the processor/system are configured, the operating system can enable normal programs to start in unprivileged mode. Once the program is running in unprivileged mode, it is unable to change the security settings.

In fact, some operations that many people would consider “basic” require privileged operations. One example is file access. A process in unprivileged mode cannot directly access hardware including storage drives. Because it cannot access the storage hardware, it cannot read from or write to files. This is true for all the usual programs that run on a computer such as a word processor like Microsoft Word. So how does Microsoft Word open files if it is running in an unprivileged mode?

Unprivileged programs can ask the operating system to perform privileged operations on its behalf. With file access, for example, Microsoft Word, unable to access the files directly, can ask the operating system to access the file on its behalf. The operating system can decide whether the process has permission to access the file or not. The operating system can exercise any number of checks or filters to ensure that the file access is safe for the system to permit. These requests to the operating system are called *system calls*.

This illustrates another form of isolation: *hardware isolation*. By isolating the hardware, including file storage, from running processes, the operating system can enforce access controls. Even if malicious programs start running, there is a limit to the amount of damage they can do. As bad as it is that an evil program could delete all of a user’s files, it would be even worse if they could delete system files. Or, if the evil program could change operating system files, the attacker could take over the entire operating system, with all of the privileged controls that come with it.

Operating systems also provide *process isolation*. Processes are not allowed to interfere with, or even spy on, each other. One of the ways an OS enforces this is using *address virtualization*. Every process that is running has to have its instructions in RAM. If multiple processes are running at the same time, they are all using RAM at the same time. If a process could load or store data to any place in memory, it could read another process’s data or even change it. Remember that the instructions are also in memory, so an evil process could change another process’s instructions or use a branch to jump to a completely different set of instructions (in other words, the evil program could reprogram a process to do anything).

With address virtualization, a process does not see the “real” address layout of memory. Instead, the operating system *lies* to the computer program and only shows it the memory allocated to it, and with fake addresses. The fake addresses are called *virtual* addresses. From every process’s perspective, its (virtual) memory starts at address 0. Every process “sees” their allocated (virtual) memory as starting at 0 and going to some maximum value that represents the end of its allocated space. When a process reads or writes to memory, it uses the fake (virtual) address, and the operating system *translates* the virtual address to the real address in memory. It is impossible for a process to interact with the memory of other processes because it cannot even have an address for it. If the process tries to use a virtual address that is out of range, the operating system will report an error.

The use of a virtual address space and mediated access to hardware for every process means that if an attacker manages to start an “evil” unprivileged process, or if they are able to corrupt a “good” unprivileged process as will be discussed later in the chapter, the attacker’s access to the system is unprivileged and isolated. Most of the time, this means the attacker has limited access to files, hardware, and other resources. And, above all, the attacker does not have access to the security controls of the processor and operating system. This assumes, of course, that the attacker was only able to compromise an unprivileged process. Things are much, much worse if the attacker manages to get privileged access, which I will talk about a little later.

It is worth noting that, although I have walked through these isolation approaches from a security perspective, many of these protections were developed at least partially because of *errors*. Computer programs always have bugs (flaws or mistakes) that do something unexpected. They can accidentally corrupt memory or unintentionally change things. By isolating programs, errors have a limited impact. But there is a relationship between accidental errors and malicious actions. And regularly the solutions are similar or identical.

Story Time: Spectre and Meltdown

It is harder than you might think to maintain isolation. In one of the most shocking findings in cybersecurity, researchers disclosed in 2018 that through *side-channel* information (see Chapter 3), it was possible to read information from anywhere in all of the computer's memory. In other words, all of the isolation provided by the operating system, at least for read access, was completely bypassed. This would mean that one running program could read all of the data out of memory about another running program including secrets, keys, and any other sensitive data. The attack impacted most modern processors regardless of the operating system or programs running. Because no software is responsible for the vulnerability, no software patch would fix it. This attack, called *meltdown*, was so serious because it affected so many computers with no immediately easy fixes [163].

At the same time *meltdown* was disclosed, the same researchers also disclosed *spectre*. Both attacks had some similarities and they were related to each other (they are often talked about together as well), but they are different attacks. *Spectre*, unlike *meltdown*, is focused on reading its own memory space, but information it would not normally have access to nonetheless. A good example of this is code executing in a sandbox, like JavaScript, which is discussed later in this chapter. Normally, JavaScript cannot read data outside of its sandbox; *spectre* can enable these scripts to bypass these limitations and read outside of the sandbox [153].

Although I cannot get into the somewhat complicated technical details here, these two attacks "broke" the security of the processor through *inferring* information (i.e., side channel) from very deep, common, and nearly ubiquitous processor operations. This is a terrifying warning about how difficult it is to really enforce isolation and how a low-level attack on the hardware is very difficult to block.

Most, if not all, contemporary operating systems provide this basic level of process isolation and this technique does a fairly good job of security the system against errors. However, the effectiveness of the process isolation for actual security threats and for enforcing security policies varies widely by both operating system and hardware. This broad range of security strength at the lower level has given rise to an entire ecosystem of security products that aim to increase the "hardness" of a given system or otherwise add additional process and resource isolation. Because of this, host system security is usually a "stack" of technologies that start with the operating system and then are built up from there in layers.

Enforcing Access Controls

Process isolation, as discussed earlier, brings together many concepts from previous chapters. A user that accesses a system must first authenticate themselves to the system using the approaches described in Chapter 2. Once authenticated, users interact with the system through processes (running programs) that operate on their behalf. In other words, when a process started by a user requests access to a resource from the operating system, it often carries the user's identity with it. Sometimes, for security reasons, it will use a separate identity, but even these special identities are usually tied to the user in some way or form.

Thus, as described in the previous section, when the process attempts to access a resource, such as a file, it is isolated from it and cannot access it directly. Instead, it must ask the operating system to mediate that access. The operating system examines the access request to determine if it is authorized. If the requested resource is protected by an ACL, the operating system will determine if the process is authorized to perform the operation on the resource. Again, the process is carrying the identity of the user (or a related identity); the operating system will check if that identity is on the ACL and, if so, what operations are permitted to it.

On the other hand, if the resource is protected by capabilities, the process would provide that capability to the operating system. The OS provides access upon validating the capability.

Both DAC and MAC permission technologies depend on, to some degree or another, the operating system to perform these access checks. For example, as discussed in Chapter 3, SELinux (Security-Enhanced Linux) supports mandatory access controls (MAC) on top of the usual discretionary access controls (DAC). The SELinux operating system checks both MAC and DAC permissions when a process makes a system call (a request for a privileged operation). When the system call is made, the OS first checks the basic DAC permissions first. This includes the basic read, write, and execute permissions that Linux ACLs support. If the DAC permissions do not grant access, no further checks are initiated. However, if the DAC permissions do permit access, then the MAC permissions are checked next.

However, in computer systems, there are other components besides the operating systems that enforce ACLs. Some computer programs are not meant to be run independently but, rather, are intended to provide services to *other programs*. These service-oriented programs are sometimes called *middleware*. Middleware includes software such as SQL databases. Middleware runs separately as its own process; when other processes want to use it, they have to access the middleware through communications enabled by the operating systems.

It is common for middleware, like databases, to have their own internal ACLs. When a running process connects to the database process, it may be required to transmit an identity and a password (or other authentication means) to the database. The database will authenticate the identity internally, using its own authentication mechanism, and then use ACLs to determine authorizations within the database.

Browsers are another example of middleware. You may not think of them as middleware because they appear to be their own program. But in actuality, browsers are middleware that run programs *from the Internet*. Every website is, more or less, a computer program you run on your computer through the browser.

I will talk about browsers and browser security in Chapter 9. However, Google's Chrome browser has a security design worth mentioning here. As will be discussed in greater detail later, the Internet is a very dangerous place from a cybersecurity perspective, and every website a user visits carries a certain amount of risk. Similar to concepts of process isolation, Google attempts to limit the amount of damage an attacker can do to your browser by running *each tab as a separate process*. Although I described a process as a running program, that is a simplified explanation. In the case of Google's Chrome browser, which a user perceives as a single program, each and every tab runs as if it were a separate program. The operating system isolates each tab from each other with its own virtual memory access, its own time slicing, and all of the other isolation described earlier.

To understand the value of this, an attacker learned of a browser vulnerability. The attacker then creates a website that is designed to exploit this vulnerability. When a user visits this website with their browser, the attacker is able to compromise and take over the browser. This is an example of a *drive-by download*—an attack that works simply by visiting a malicious website.

If the user's browser is all in a single process for all tabs, the attacker could potentially see and/or alter all other browsing sessions in all the other tabs. If the user has a tab open at the same time to their bank's website, the attacker would most likely be able to intercept, steal, or change any of that data.

On the other hand, with Chrome's approach of one-tab-per-process, when a user visits the attacker's website, the attacker will most likely corrupt just a single tab. It is running in its own process and is subject to all of the isolation enforcement provided by the operating system. Although the attacker can still do a significant amount of damage with just one tab, this architecture helps to reduce it.

Unfortunately, this kind of security comes with a cost. Because each tab is its own process, it uses up a significant amount of memory. Other browsers can typically use less memory because some of the data is shared.

Story Time: Spook.js—More Good News Than Bad

The world of computer vulnerabilities often has to approach everything from a "worst-possible scenario" point of view. Even if an attack seems unlikely, the fact that even an unlikely attack exists is problematic. After all, what is unlikely can change. Moreover, an attack may be statistically unlikely but be catastrophic if it occurs. A risk is often calculated as the product of likelihood and impact, so if the impact is significant enough, the risk is significant even if the likelihood is low.

With all of that said, the *Spook.js* attack, although potentially scary and dangerous, actually shows that Google's defenses, added after the reveal of *Spectre*, are quite effective. This attack demonstrates that in some edge cases, a *Spectre*-based attack can still steal sensitive information. But the upside is that Google's defenses defeat *Spectre* for almost any kind of practical situation.

The basic idea is this: *Spectre* attacks enable evil code to find any information in memory for its own process (i.e., within the same running program). When JavaScript, which is used to provide dynamic functionality for websites, is "running," it is actually just part of the existing web browser program. The web browser program *runs* the script program within a special part of itself called an *interpreter*. But from the perspective of the operating system, it is all running within the same process. Attacks using *Spectre* can enable the JavaScript to read data outside of the interpreter.

However, Google strictly isolates each website into its own process. This strict isolation means that *generally* there is no *Spectre*-style attack wherein attack code from one website could read data from another website. What *Spook.js* showed is that under certain combination of conditions, it was still possible to leak out some data. This attack is not irrelevant nor insignificant, but the conditions required are not likely. On the other hand, *if strict process isolation were not used* and all websites were being handled in the same running program, it would be relatively easy for data theft to happen [31].

The good news is, Google's design decision means that *practical* attacks have generally been defeated, while some very unlikely and impractical attacks can still happen [221].

In addition to the access controls enforced by the operating system and middleware, some programs have their own access controls. A very common example of this is the controls put on documents by PDF readers and word processing systems. In many cases, these systems do not use ACLs internally but rather have the protection of the file tied to cryptography. A file can be encrypted with the AES algorithm discussed in Chapter 5. Although AES keys were described in that chapter as random bits (1s and 0s), it is possible to derive an AES key from a password. Thus, the user selects a password to protect the file, the password is converted into a key, and the file is encrypted using the derived key. Now, the file cannot be recovered without the password, and no ACLs are required.

There are limitations, however, to encrypting a document. It basically reduces the permissions model to all or nothing. If the file is decrypted, all operations are possible. If the file is encrypted, no operations are possible. In Chapter 10, I will talk about how Microsoft uses the cloud to provide more fine-grained access controls on email and documents. But with reference to host security, the subject of this chapter, the key idea discussed in this section is that security controls are often a combination of security controls built into a hardware security component, the operating system, middleware software, and applications (Figure 7-1).

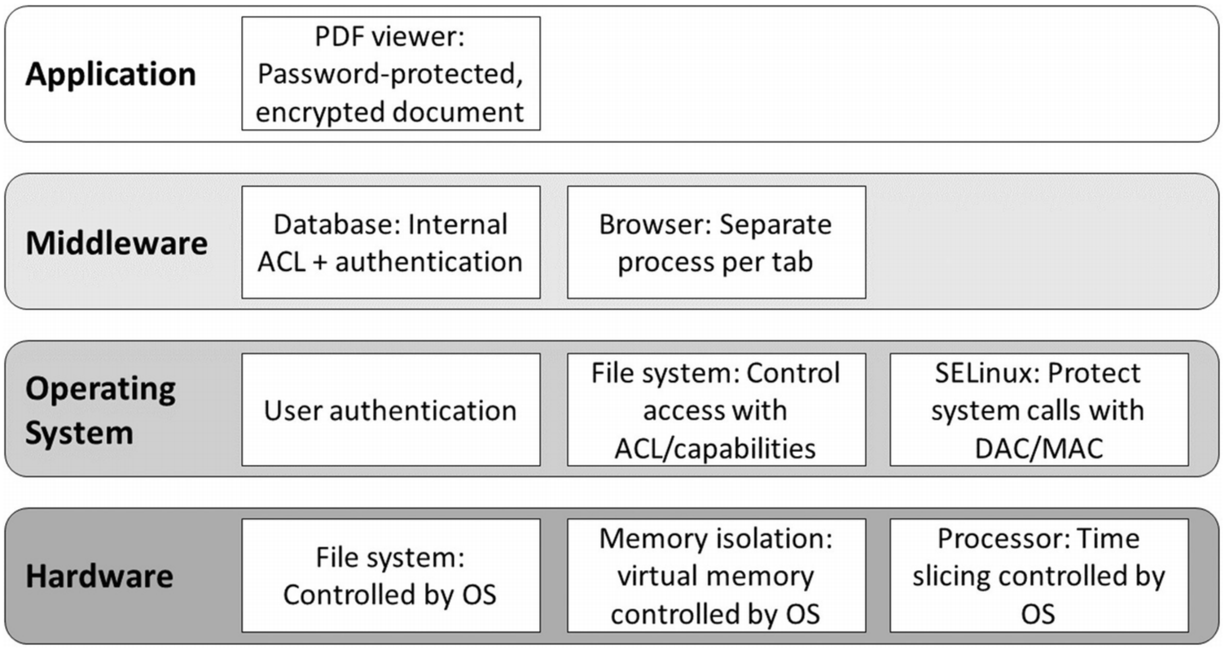

Application
PDF viewer: Password-protected, encrypted document
Middleware
Database: Internal ACL + authentication
Browser: Separate process per tab
Operating System
User authentication
File system: Control access with ACL/capabilities
SELinux: Protect system calls with DAC/MAC
Hardware
File system: Controlled by OS
Memory isolation: virtual memory controlled by OS
Processor: Time slicing controlled by OS

Figure 7-1 Some examples of access controls that may be enforced across the various layers of a system

Stronger Hardware-Based Isolation

Although most processors support some level of privileged operation used by the operating system to protect itself, there are limits to the security of the standard approaches. For example, one significant issue is that whatever program starts first (i.e., the operating system) will be in control of privileged operations. There is no easy way to verify that an authorized operating system is running and that an authorized operating system has not been altered.

Another problem is that even if the operating system is authorized and correct, it has no mechanisms for protecting data that the operating system should not have access to. Copyright owners, for example, would like to ensure that the material on which they hold copyrights can be protected even if rendered on a user's personal device. This means that there must be some mechanism for ensuring that the data copying must be controlled by a security policy that cannot be changed or ignored by the operating system.

One solution to this problem is to create additional security hardware that can perform computing outside of the influence of the normal operating system. One example of this is TrustZone, a security extension that is available for certain Arm processors. These processors often find themselves in mobile devices, like phones. The TrustZone security extensions define a “two-world” model: the normal world and the secure world. A standard operating system runs in the normal world, while a separate operating system, usually a smaller and security-focused OS, runs in the secure world.

Similar to the privileged operations for normal processors, the secure world operations of the TrustZone security extension are controlled by special registers that track which mode of operation the processor is operating in. Certain hardware, processor registers, cryptographic modules, and entire swaths of RAM are only available when operating in secure mode. This means that some information and data, including cryptographic keys, can be loaded into the processor during some initialization phase (e.g., at manufacturing), and that data will remain inaccessible in normal world mode. The normal operating system will not be able to have direct access to it. The secure world is said to offer a *trusted execution environment* or TEE.

Story Time: No More Blu-rays for Your PC

The vast majority of consumers do not watch Blu-ray discs on their PCs. But those that do will not be able to do so on newer computers. Back in 2016, Intel introduced *Software Guard Extensions* (SGX) into their processors to provide Intel-based computers with what is called an *enclave*. Similar to some of the trusted execution environments for Android, SGX was supposed to create an area of the computer that even the operating system (and the computer owner) would not have access to. The movie industry, always concerned about illegal copying, permitted Blu-ray discs to play on PCs only if they were equipped with Intel's SGX system. The goal was to keep the PC from stealing Blu-ray keys and decrypted data.

Sadly, SGX has had many issues and has been compromised over and over. Many reasons have been attributed to the failures of these components, including complexity, lack of third-party audits, and side-channel leaks that enable attacks similar to Spectre (it's back...). So, in January of 2022, Intel has confirmed that they will no longer be building SGX into any of their PC-type chips (although it will still be included in chips for the server market). Without SGX, Blu-ray discs will not play on any of these newer chips, and there appears to be no plans to change these restrictions. While it is true that most people will not notice, as dedicated Blu-ray players will not be affected, if you happen to be an HTPC (Home Theater PC) enthusiast, you will have to stick with old hardware and maybe even Windows 10 [25, 263].

Similar to a system call, when the normal mode OS needs some kind of service from the secure world, it has a means to request certain operations. The secure world, for example, can perform cryptographic operations without revealing any keys and only after validating that such operations are permitted.

This can be useful to phone manufacturers that want only authorized operating systems installed on their handsets. The TrustZone components of the hardware can be configured with cryptographic keys that are used to verify the signature on an operating system installation (called a boot image). When the phone starts up (boots), the TrustZone components can verify that any boot image installed is *signed* by the authorized distributor. This means that any modifications to the operating system will be rejected, as will any operating system images created by an unauthorized third party.

Stronger Software-Based Isolation

On the opposite end of hardware-based isolation, software-based isolation can also be much stronger than the traditional approaches described in the previous sections. Both Android and iOS operating systems for mobile devices lock down processes much more tightly. Both OSs describe these locked down environments as sandboxes. Sandboxes and virtualization are approaches used to increase isolation.

Defining these two terms is difficult because they are used to mean different things in different circumstances and because they have significant overlap. However, a reasonably safe definition of a sandbox is: a restricted software execution environment for processes that reduces their capacity to cause the system harm. Virtualization, on the other hand, is software that simulates some other computer component. The simulated component can be software, hardware, or both. However, the more specific term *virtual machine* almost always refers to the simulation of the computer hardware (at least the CPU but sometimes other components such as memory and storage).

The reason the use of these two terms can be confusing is because some of the most commonly used sandboxing technologies use virtualization. And, conversely, one of the most common motivations for using virtualization (such as a full virtual machine) is to provide a sandboxed environment.

While there are many sandboxing technologies, some of the most widely used are those found in mobile devices. The key idea behind both the Android and iOS sandboxes is that each application that runs on the mobile device is locked into its own isolated environment. Unlike standard computer operating systems, both Android and iOS lock applications into only seeing their own files. Much like how a standard OS provides a virtual address space that isolates a process from other process memory, both Android and iOS provide a virtual storage space that isolates a device application from all other device applications. This is depicted in Figure 7-2.

Mobile phone sandboxes also typically isolate the running programs (processes) from each other. For example, as explained in Chapter 3, Android uses SELinux policies to lock down what each process is allowed to access. Apps are also limited in which system calls they can make to the Android operating system [23].

Most major device resources are also locked down from the application. This means that applications must request permission for using the camera, the microphone, or even the contact list on a phone. By default, apps are denied permissions to any system resource and must have the permission explicitly granted by the operator. Sandboxing and virtualization are approaches that work by increasing software isolation in software. The basic idea behind both approaches is to add another layer of separation between software and the rest of the system.

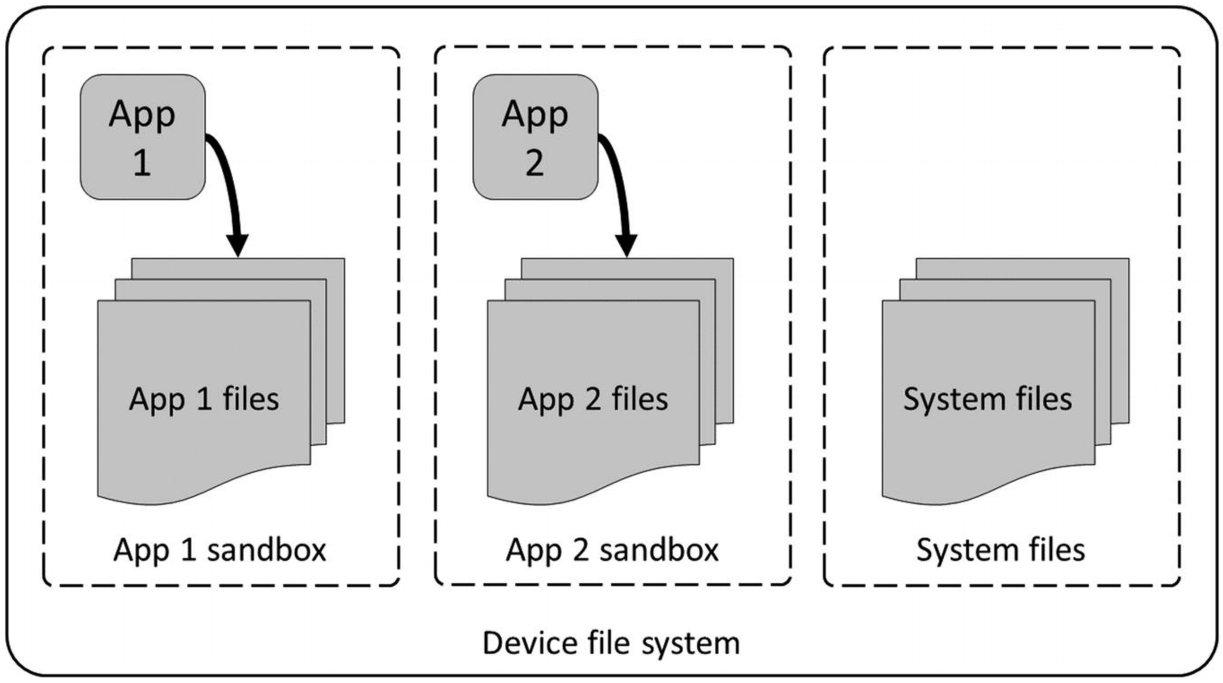


Figure 7-2 Android apps are allowed to access files that belong to them, but not files that belong to other apps

Another widely used sandbox is found in the web browser. Most websites use a type of computer program called JavaScript. When a browser visits a website, it downloads the JavaScript built into the web page and executes it. But because programs from the Internet should not be trusted too much, the browser executes the JavaScript in a protected environment that prevents the JavaScript program from accessing computer resources. For example, JavaScript running in a browser is not allowed to save or load files.

Unlike computer programs that execute directly on the processor, as described earlier in the chapter, JavaScript programs are *interpreted*. This means that JavaScript instructions are *not* CPU instructions. Instead, JavaScript instructions are written in a form that is human readable, even if one has to be a programmer to really understand them. And instead of running the JavaScript program on the processor, it is run *by the browser itself*. That is, the browser *interprets* the JavaScript program and performs the operation. Because the browser is interpreting the JavaScript, it can limit what the program is allowed to do.

Note, however, that because the browser is interpreting the JavaScript, the JavaScript program is running in the same process as the browser interpreter. This is why Chrome's one-tab-per-process helps to create stronger isolation.

Most browsers can also be equipped with a security policy that it can enforce on JavaScript beyond the standard sandbox limitations. The use of a security policy to customize the restrictions of the environment is a common feature in sandboxing technology. Just like the operating system can mediate access to system resources for general processes, the sandbox mediates access to those same resources but at a higher layer of the system that is more appropriate for fine-grained customization. A customizable security policy enables the sandbox to make more careful determinations regarding which accesses should be allowed within a narrower context.

Sandbox technology is never perfect, however, and there have been some instructive failures. Java, for example, is another programming language that was designed to run in a sandbox. Note that Java and JavaScript are not related technologies despite their unfortunately similar naming. But both languages were designed for sandbox execution.

In the case of Java, however, the sandbox is created from the Java Virtual Machine (JVM). Java instructions do not run on a computer's real processor but run, instead, on the simulated processor of the JVM. All system accesses, such as access to files or other system resources, are checked against this security policy while the program is running. Whereas an ACL or a capability enables or denies an operation once, the JVM can check every access as it happens.

While Java technology in general has been successful, Java promoted a special security policy as the Java Sandbox for running Java programs on Internet websites. These Java programs are called *applets*. While I would classify the entire JVM as a sandbox, because it checks every access against a security policy, the Java Sandbox was a specific subset of policies intended for Internet security.

Said another way, all Java programs run using the Java Virtual Machine. And all Java program access to system resources was mediated by a security policy. This, by itself, fits the definition of a sandbox. But Java specifically called the policy created for applets running in a browser the Java Sandbox.

In practice, the Java Sandbox (i.e., the policies for applets) was unsuccessful. There always ended up being too many ways to get around the limitations of the sandbox and there always ended up being too many dangerous bugs. Because of the constant security problems, by 2013 computer security organizations were recommending removing Java altogether from browsers or only permitting "signed" applets from a trusted source [11, 198]. Most contemporary Internet websites do not use applets at all.

Virtualization, on the other hand, is about simulating hardware or software components. In this chapter, you have learned about address virtualization, for example. Operating systems do not let processes see the real memory address space. Instead, it provides the running process with a virtualized address space that confines them to their own allocated memory.

And, as just discussed, the Java Sandbox is implemented in the Java Virtual Machine (JVM). The JVM simulates a processor for the Java program. This enables the JVM to intercept all of the system calls meant for the real processor in order to determine if they are authorized.

More extreme examples of virtualization include containers and virtual machines (VMs). A VM simulates an entire computer. It simulates a processor, storage drive, RAM, and other hardware peripherals. The simulation is so complete that it is possible to install operating systems on the VM and run it as if it were a real computer. When running a virtual machine, it is common to call the computer running the virtual machine as the “host” and the virtual machine the “guest.” The host has its own operating system (the host OS), and the virtual machine has its own separate operating system (guest OS).

There are many advantages to VMs. Medium-to-large companies can use virtual machines in the cloud (e.g., using Azure or Amazon), instead of buying hundreds or thousands of machines. VMs permit scalability and ease of maintenance.

But for security reasons, VMs are also very powerful. VMs can be used to create a strong sandbox. Programs can be run within the virtual machine, and if they cause any damage, the virtual machine can just be turned off, and a new “clean” version can be turned on in its place. There are many security systems that test untrusted software in a VM before letting it be used in any “real” systems. VMs can also be used for isolating different types of servers and programs from each other. That way, if one server is compromised by an attacker, there is much less chance that the attacker will be able to compromise the other.

Many modern processors now offer support for virtual machines that have security implications. Although a virtual machine is simulated, it is generally too slow to fully simulate a processor. Instead, many virtual machine technologies run as much of the guest OS and guest applications as possible directly on the processor. Often, the virtual machine technology needs to only intercept and modify requests like system calls and a few other types of operations. To better support this, many processors now support more modes than just privileged and unprivileged. These processors offer a *hypervisor* mode, a privileged mode, and an unprivileged mode. A hypervisor is a special host operating system that is designed just for operating virtual machines. By having the host OS and the guest OS use different security modes at the hardware level, the processor can help to isolate the guest OS from the real system, even though it is running on the real processor.

Containers are a lighter-weight form of virtualization that does not fully simulate an entire system. A full explanation of the differences between containers and virtual machines is not provided here. In terms of security, containers can provide some of the same isolation as virtual machines, but it is not as complete. Generally, it is easier for an attacker to “escape” a container (into the real host OS) than to escape from a full virtual machine.

Software Vulnerabilities

Having studied a little about how computers protect themselves and isolate processes from resources, I will now turn your attention to some of the tricks attackers play to get out of isolation. One of the biggest thorns in the side of computers for decades is *control flow hijacking* attacks, such as *buffer overflow* attacks. A buffer overflow attack is not the only example of a vulnerability, but it is one of the most significant. And it does a good job of illustrating the challenges to securing a computer system.

To understand how these attacks work, you first have to go back to how a process runs on a processor. Remember from earlier in this chapter that a program is a sequence of instructions. Branching instructions allow the processor to jump around in the set of instructions based on inputs. This allows the program to be interactive and dynamic as otherwise it would just execute the same instructions every time, one after the other.

There is a special kind of branching instruction used extensively in computer programs called *call*. Computer programs have to be broken down into what are, essentially, mini programs. Programs are too complicated to not subdivide. The *call* instruction tells the processor to jump to another set of instructions to perform some subtask and then return back to where the call happened when the subtask is complete.

For example, imagine a weather app on your phone or other mobile device. The app reports temperatures in both Fahrenheit and Celsius. Internally, all of the data is in Celsius, and it computes the Fahrenheit from this using the commonly known formula: $F = \frac{5}{9}C + 32$. Although fairly simple to express in mathematical notation, this operation requires a number of instructions to the processor. There is at least one division operation, one multiplication operation, and one addition. Moreover, because of the limited number of registers, there may need to be copying of intermediate values to memory and then back into registers for the next operation. It would be inefficient to write these instructions over and over again throughout the program. Instead, it can be written *once* as a subroutine.

A *subroutine* is a chunk of functionality or set of instructions within a larger computer program. In the example I am using here, the subroutine provides the instructions that can take a number representing a Celsius temperature as an input and produce an output number that represents the corresponding Fahrenheit temperature. Another common name for a subroutine is a *function*.

Conceptually, a subroutine or function is almost like its own mini program within a program. It becomes a building block that can be used over and over, reducing duplication and making it easier to fix errors. For example, imagine that the author of a program had been given the wrong formula for this calculation: $F = \frac{5}{19}C + 32$. If a program was written to repeat these instructions everywhere they were needed, the programmer would have to fix the formula all throughout the program. On the other hand, if the formula were written *once* within a subroutine that was used repeatedly, the programmer would only have to fix it once!

Using subroutines, every time a calculation is needed, the processor will jump to this one set of instructions to execute them. But once the subroutine is finished, the processor needs to jump back to wherever it was before. But how does the processor know where to jump back to? The call to the subroutine may have happened in hundreds or even thousands of places in the process's instructions. The only solution is to record this information in memory. Once the subroutine is finished, the processor can look up the "return address" and jump back to where it had been. This process is illustrated in Figure 7-3.

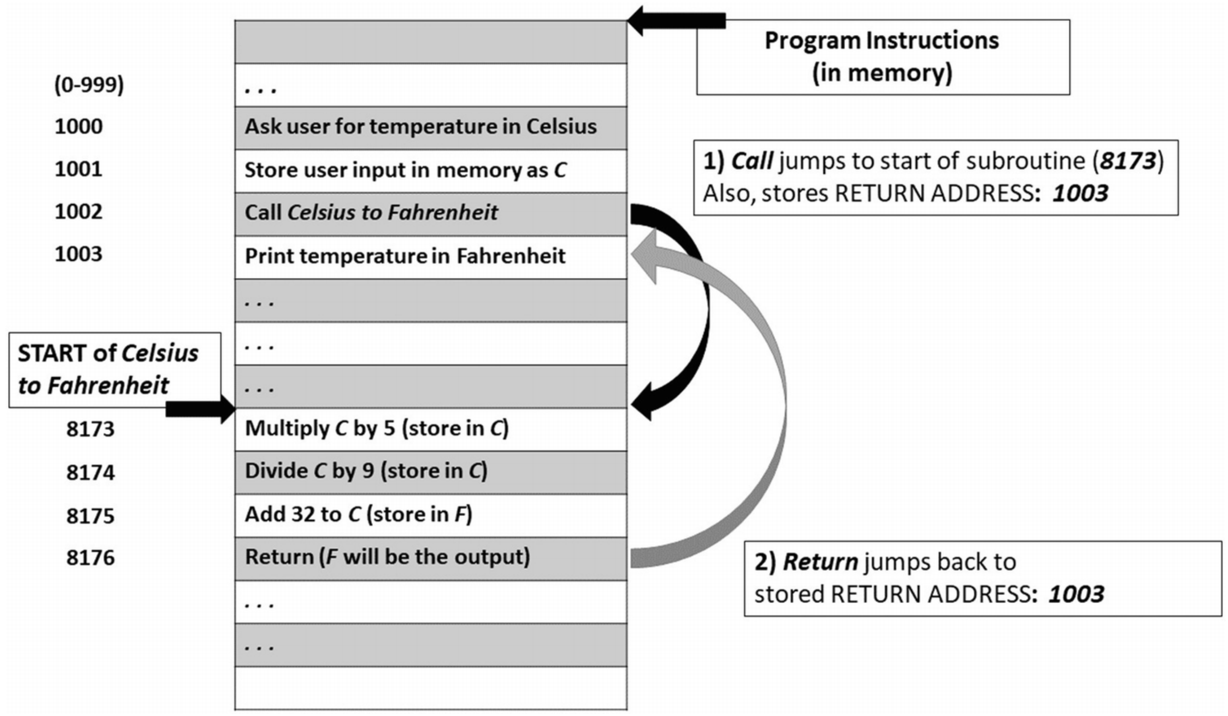

Program Instructions (in memory)
(0-999)
1000
1001
1002
1003
. . .
Ask user for temperature in Celsius
Store user input in memory as C
Call Celsius to Fahrenheit
Print temperature in Fahrenheit
1) Call jumps to start of subroutine (8173)
Also, stores RETURN ADDRESS: 1003
START of Celsius to Fahrenheit
8173
8174
8175
8176
Multiply C by 5 (store in C)
Divide C by 9 (store in C)
Add 32 to C (store in F)
Return (F will be the output)
2) Return jumps back to stored RETURN ADDRESS: 1003

Figure 7-3 This figure depicts a simplified view of a subroutine call and the subsequent return. Instructions are stored in sequential memory. A *call* triggers a jump to some other location in memory and stores where it came from. The *return* triggers a jump back to where it left off

NOTE:
The representation of memory and subroutine calls and returns in Figure 7-3, and others in this chapter, is abstracted and simplified. The key details illustrated in these figures are that each instruction has an address and that addresses are sequential in memory.

In a real system, instructions typically take more than one byte, and the addresses increase by the size of the instruction (i.e., the address increases by 4 after a 4-byte instruction). Also, the instructions listed here are not true machine-level instructions but are more understandable operations that help explain the example.

Another important simplification is how data is stored. In Figure 7-3, for example, I describe the data as being stored with names like **C** and **F**. In a real computer system, there are no names like this, and data is stored in registers or on the stack.

A full and detailed explanation of how low-level computer architecture works cannot be covered in this book. For the very technically minded, additional reading is suggested at the end of the chapter that can fill in these gaps.

Calling subroutines is complicated by the fact that a subroutine can call another subroutine. This is sometimes called a *call stack*, and it necessitates storing the return addresses of each call in memory. That is, if there is a first call to subroutine A from address 1, that address must be stored in memory. Then, if in executing the instructions of subroutine A there is a call to subroutine B from address 2, that second address must also be stored in memory. This process can repeat indefinitely (until the computer runs out of resources), but to finish the example, suppose that in executing subroutine B there is a call to subroutine C from address 3. This address must also be stored. Figure 7-4 illustrates a simple call stack.

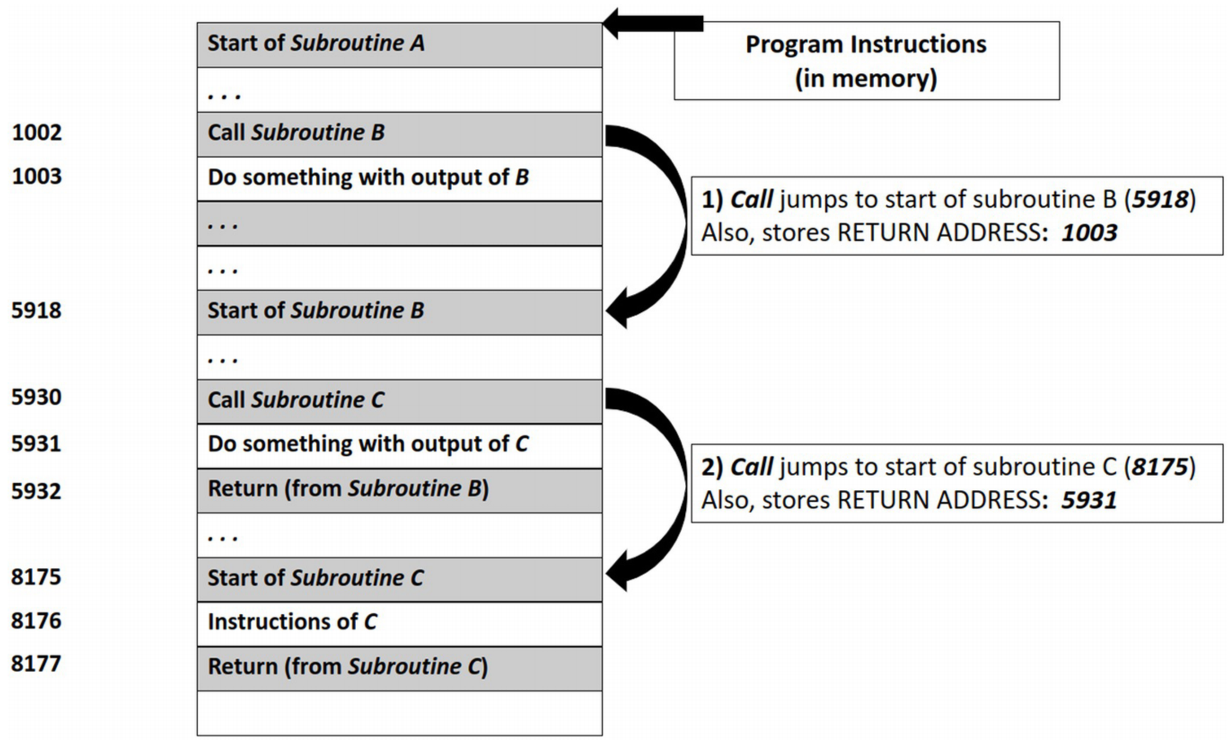

Start of *Subroutine A*
. . .
1002
Call *Subroutine B*
1003
Do something with output of *B*
. . .
. . .
5918
Start of *Subroutine B*
. . .
5930
Call *Subroutine C*
5931
Do something with output of *C*
5932
Return (from *Subroutine B*)
. . .
8175
Start of *Subroutine C*
8176
Instructions of *C*
8177
Return (from *Subroutine C*)
Program Instructions (in memory)
1) *Call* jumps to start of subroutine B (*5918*) Also, stores RETURN ADDRESS: *1003*
2) *Call* jumps to start of subroutine C (*8175*) Also, stores RETURN ADDRESS: *5931*

Figure 7-4 This figure depicts a simplified view of a chain of subroutine calls. Each call jumps to some other location in memory and stores the return location

When subroutine C finishes, it needs to return to where it was called. The processor looks up in memory and sees that C was called from address 3 and returns to that location. This instruction was part of subroutine B which now is able to complete its own operations. Once finished, subroutine B needs to jump back to wherever it was called. The processor looks up in memory and sees that this location is address 2. The processor jumps back to address 2, which was part of subroutine A. The instructions of subroutine A finish executing and then need to return. The processor looks up in memory and sees that the stored address is address 1 and it jumps back there, which is where this entire call stack started. Figure 7-5 illustrates these return operations.

The details of these operations can be a little bit complicated. The return addresses are not the only pieces of information stored in memory. There is, in fact, a not insignificant amount of contextual data that must be stored for each call. It is also the case that when subroutines are doing their operations they typically have to use memory to store intermediate and other values necessary for their subtask. All of that data has to be preserved even if the processor has to jump to some other subroutine in the middle of it.

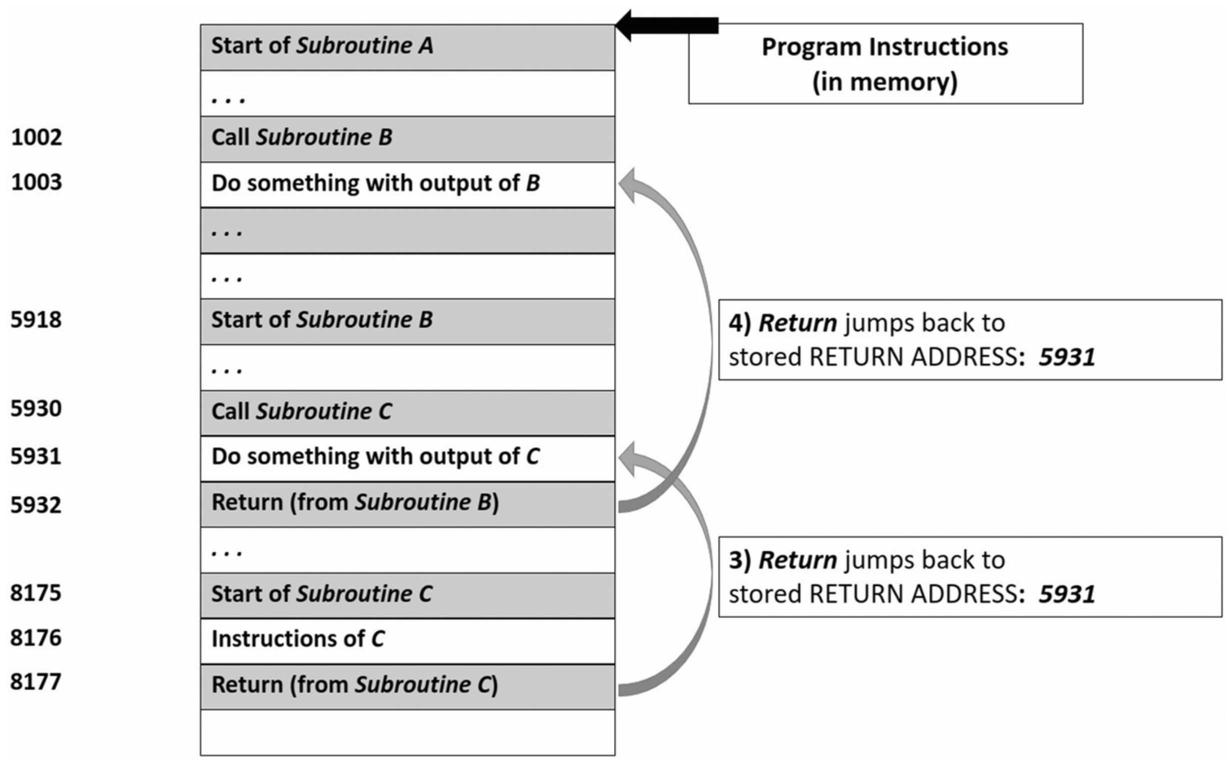

Start of Subroutine A
. . .
1002 Call Subroutine B
1003 Do something with output of B
. . .
. . .
5918 Start of Subroutine B
. . .
5930 Call Subroutine C
5931 Do something with output of C
5932 Return (from Subroutine B)
. . .
8175 Start of Subroutine C
8176 Instructions of C
8177 Return (from Subroutine C)
Program Instructions (in memory)
4) Return jumps back to stored RETURN ADDRESS: 5931
3) Return jumps back to stored RETURN ADDRESS: 5931

Figure 7-5 Returns in a call chain reverse each call. Each subroutine's return goes back to the location of the subroutine that made the call

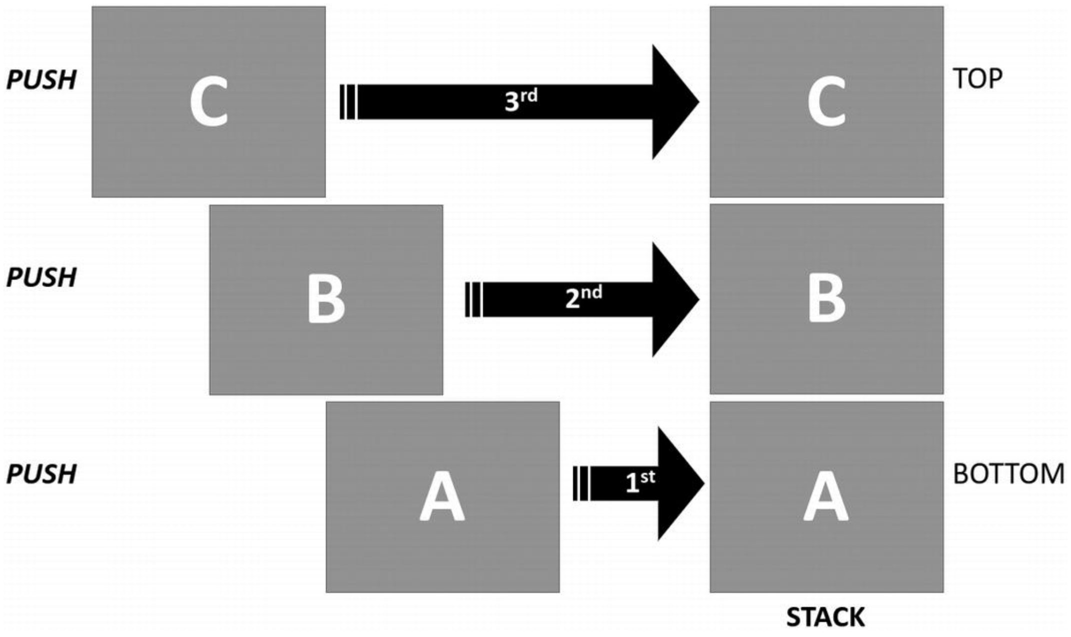


Figure 7-6 When data is pushed/added onto a stack, it is pushed at the top

To manage all of this, the data is organized into what is called a *stack*. A stack is a concept for storage based on a “last-in-first-out” (LIFO) principle. The name comes from a visualization where one piece of data is placed on top of another and then removed in reverse order as shown in Figures 7-6 and 7-7.

When a process executes, one part of the memory is set up to serve as a stack. Data is added onto the stack and then removed in reverse order. This permits the call stack example to work as outlined. Address 1 is added first, then address 2, then address 3. But when data is removed, or “popped,” from the stack, address 3 comes off first, then address 2, and then address 1.

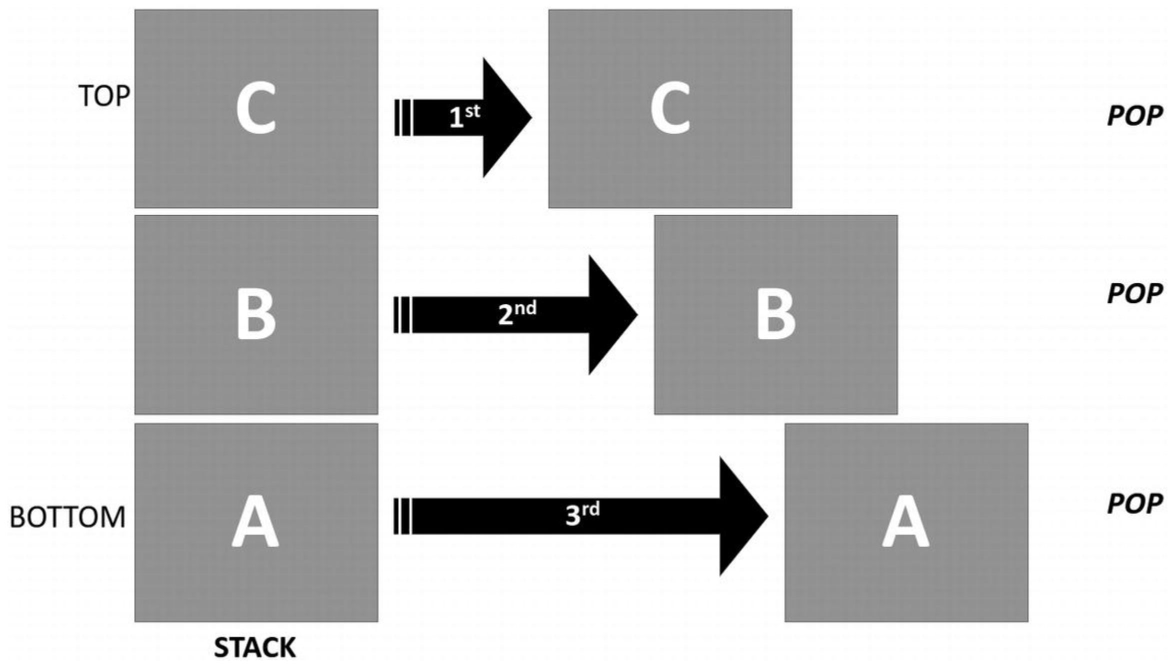


Figure 7-7 When data is popped/removed from a stack, it is popped from the top

To be clear, all of this manipulation of memory is conceptual. Pushing data onto the stack is not, somehow, creating new memory spaces, nor is popping data off of the stack destroying data. Rather, there is a memory address that is considered the "top" of the stack. Memory addresses after that are simply unused or ignored. When data is added, the top of the stack moves, and new addresses are put into use. When data is popped off, the top of the stack is reduced, and the addresses of the popped data go back to being unused (or available). In computer systems, the stack is depicted as growing from higher memory addresses to lower ones, giving it the appearance of growing down instead of up as shown in Figure 7-8.

As hinted out earlier, not only is the return address stored on the stack, *but so is all of the scratch pad memory for holding intermediate values*. This data is also stored on the stack along with the return address values. So, when subroutine A is executing, it uses space on the stack for its intermediate values. When it calls subroutine B, the stack is extended creating space for B to have its own intermediate values. In Figure 7-9, a *main* function[2] calls the *Celsius to Fahrenheit* function that I used as an example earlier. As shown in the figure, the *Celsius to Fahrenheit* function needs to use the stack to store the correct return location in *main*, but it also needs to store local data for calculating the Fahrenheit number.[3]

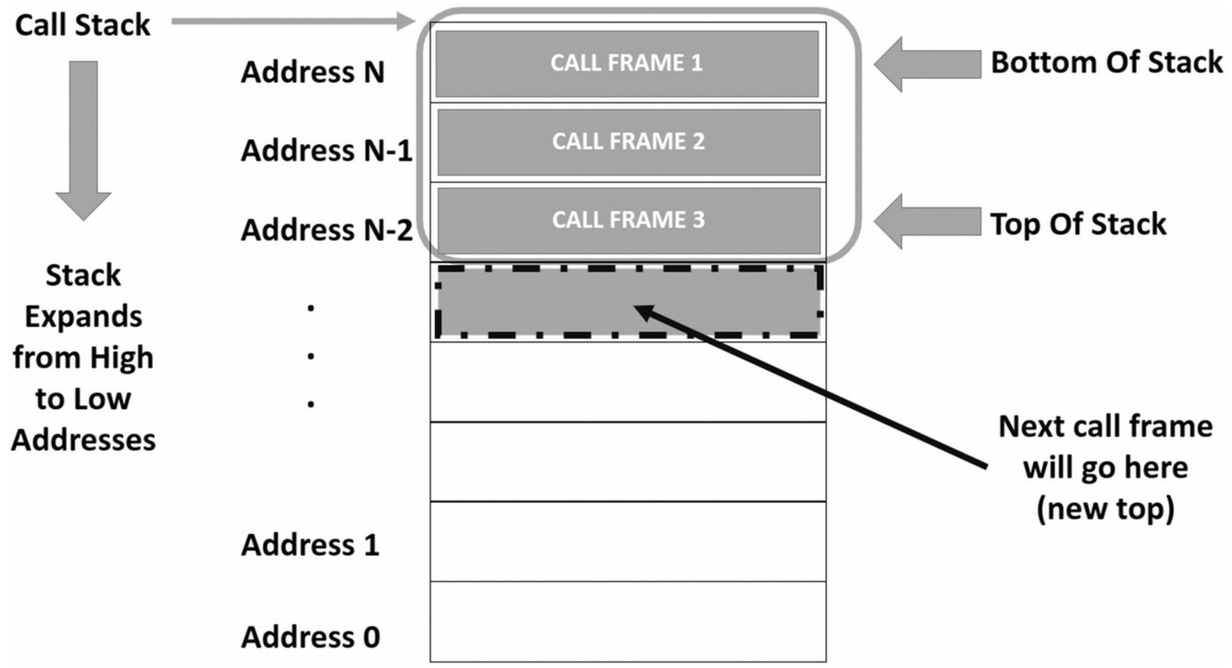


Figure 7-8 An abstract depiction of a stack in a process's memory. In memory, the stack grows downward, meaning that new data items are added at a lower address value. The stack holds a call frame for each call made while executing instructions

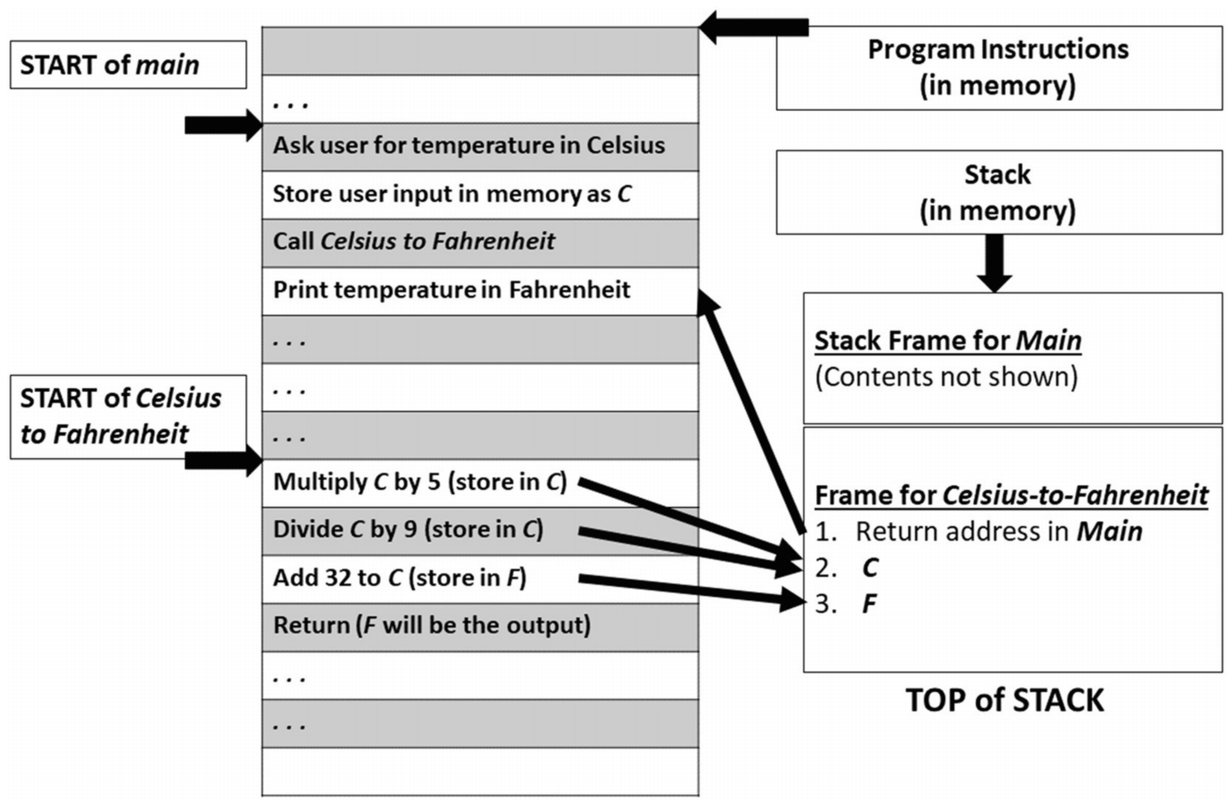

START of *main*
Program Instructions (in memory)
...
Ask user for temperature in Celsius
Store user input in memory as *C*
Call *Celsius to Fahrenheit*
Print temperature in Fahrenheit
Stack (in memory)
Stack Frame for *Main*
(Contents not shown)
START of *Celsius to Fahrenheit*
Multiply *C* by 5 (store in *C*)
Divide *C* by 9 (store in *C*)
Add 32 to *C* (store in *F*)
Return (*F* will be the output)
Frame for *Celsius-to-Fahrenheit*
1. Return address in *Main*
2. *C*
3. *F*
TOP of STACK

Figure 7-9 An abstract depiction of a stack in a process's memory. In memory, the stack grows downwards, meaning that new data items are added at a lower address value. The stack holds a call frame for each call made while executing instructions

NOTE:
Memory is not *physically* divided between processor instructions and stack, as could be inferred from Figure 7-9. Memory is just one big long sequence of addresses. To manage a computer program, designers and programmers *conceptually* divide up the memory into different functions and different *mental* models. Pay attention, for example, to how I have organized these two memory models differently. In figures showing the memory addresses for instructions, like Figures 7-3, 7-4, and 7-5, I have chosen to start the instructions with low addresses at the top. I did this for my own mental model because I wanted the program instructions to flow from top to bottom in the figure (Figures 7-6 and 7-7).

However, as explained in this chapter and illustrated in Figure 7-8, the stack addresses start with high addresses at the top. In the actual memory, of course, there is no real top or bottom. But in order to reason about memory and use it effectively, we have to create mental models that are used to manage all the different things that memory is used for: instructions, data, intermediate calculations, and so forth.

Unfortunately, this organization of data is vulnerable to the buffer overflow attacks mentioned at the beginning of the section. The intermediate values of a subroutine include “buffers” allocated for holding user input. A buffer is just a contiguous space in memory used for holding data. Suppose, for example, a subroutine was used to get a user’s login name and password. It would need a buffer to hold the login name and another buffer to hold the password. These buffers have to be set aside in memory, and one of the places they can be put is inside the scratch data on the stack. Buffers on the stack are of a fixed size. They cannot be resized.

In Figure 7-10, a buffer has been set aside to hold up to 80 bytes, which can store 80 western characters.[4] This buffer is used by a subroutine for reading in a password from a user and checking to see if it is correct. Figure 7-11 shows an example where the user’s password is “password.” This takes up 8 of the 80-character storage space. Note that the data is inserted at the bottom of the buffer and flows up to the top.

If the program is well written, it should check that the data going into the buffer will not be bigger than the allocated size of the buffer. Sadly, there are many programs that are not well written and do no bounds checking. In these cases, if an attacker puts in more data than is allocated, the data will still be inserted into memory and will overflow the buffer. In Figure 7-12, the user put in 100 As as their password, which is bigger than the allocated buffer.

The problem is that when the buffer overflows, data outside the buffer is overwritten. And what else is stored on the stack? *The return address for the call operation*. Many times when there is a buffer overflow, the attacker can figure out how to overwrite the return address with a specific value. Then, once the call completes, and the processor goes to jump back to where it came from, it jumps, instead, to an address that the attacker put there. The attacker will then attempt to use this to take control of the process. This whole technique is sometimes called *smashing the stack*.

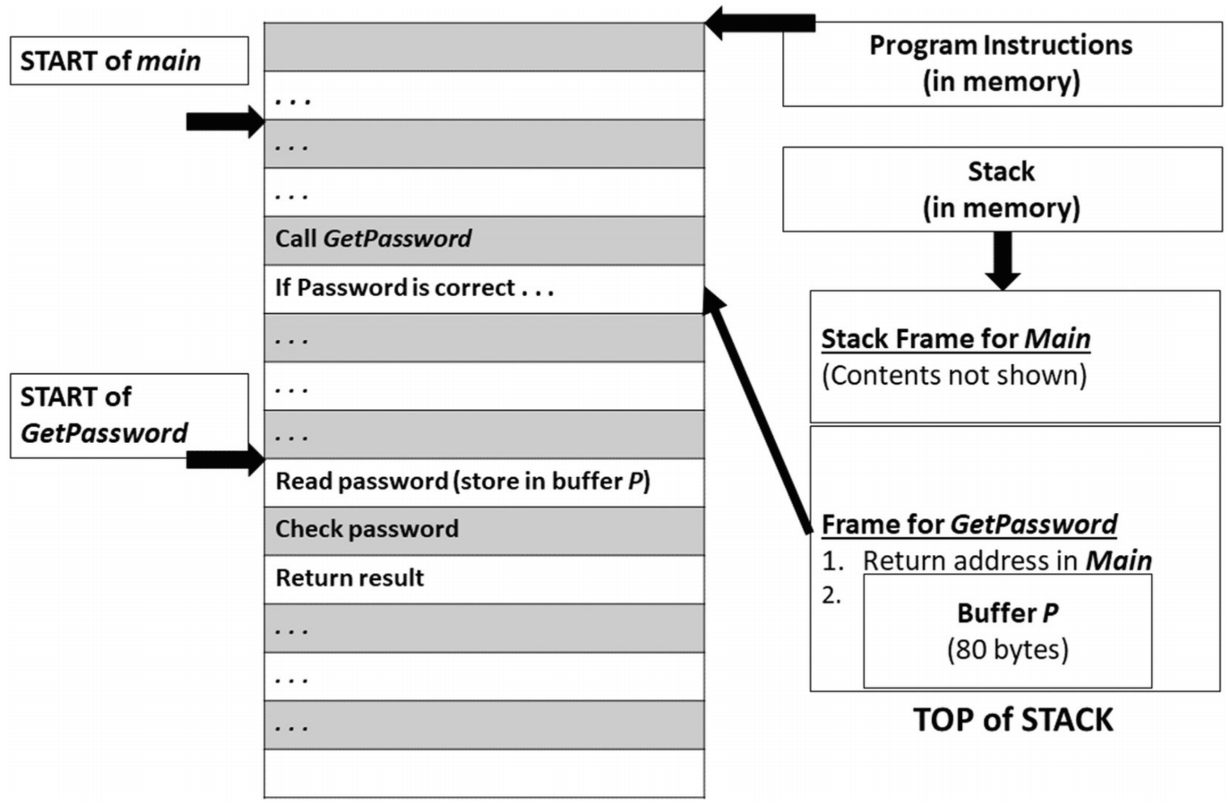


Figure 7-10 A stack frame for a password checking program. It has a buffer of 80 bytes for storing the password

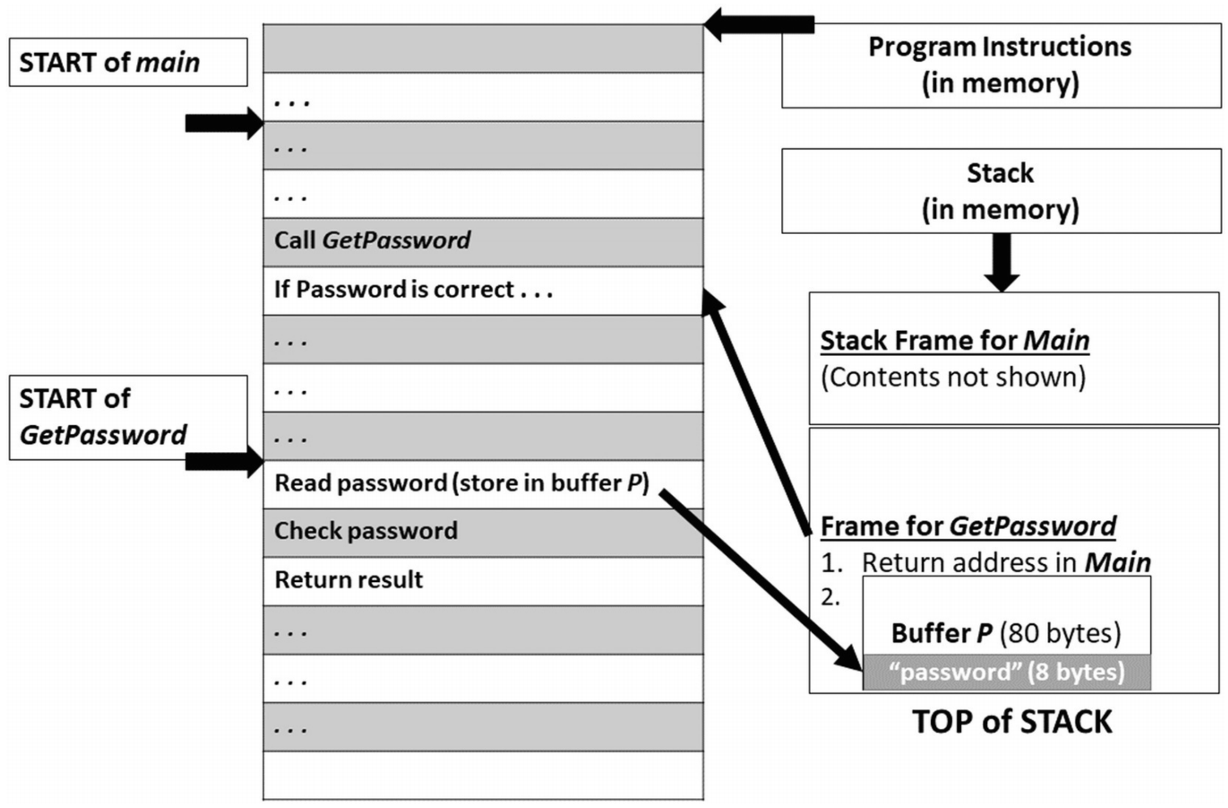


Figure 7-11 The same stack frame as in Figure 7-10 but after storing the password "password"

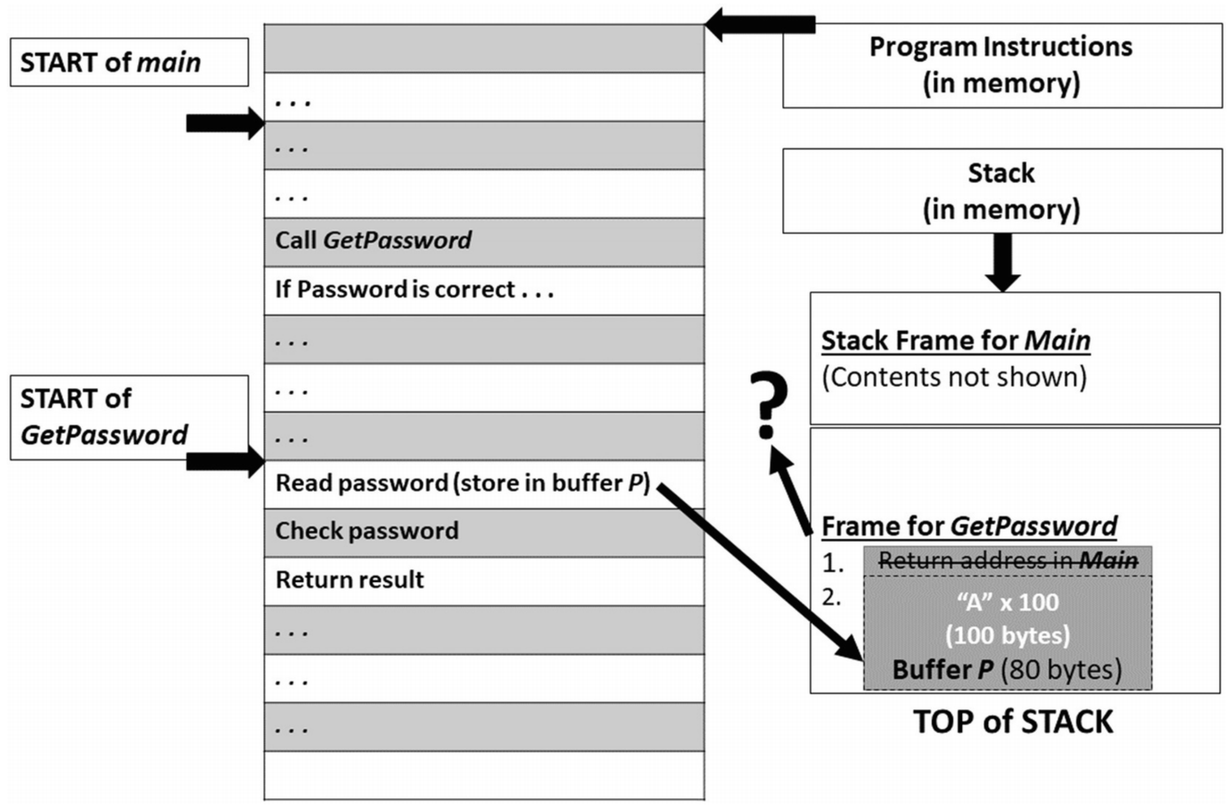


Figure 7-12 The same stack frame as in Figure 7-10 but after the user entered 100 As as the password. The buffer can only hold 80 characters and the data spills over, overwriting the return address that was stored. When the return is called, the old address has been wiped out and replaced

One of the early ways that attackers could take over the process is they inserted their own instructions into the overflow data itself. Even if the input is supposed to be a login, remember that all data is just numbers. The attacker can put in arbitrary data that does not make much sense as a login name but becomes instructions when handled by the processor. So, the attacker inserted instructions and enough padding (useless data) to overflow the buffer and rewrite the return address. The rewritten address pointed back to the beginning of the buffer itself. The processor would jump to the very place where the attacker had just inserted data and start processing it as instructions. An illustration of this classic style of buffer overflow is depicted in Figure 7-13.

This problem is possible because both instructions and data are in the same memory. The instructions and data are both just numbers, so it is impossible to tell if something should have been an instruction or is just a piece of data.

Modern systems are not so easily crackable though. Memory is internally broken down into subunits called *pages*. Each memory page can be marked with permissions including read, write, and execute. Modern systems do not permit a page to be marked both writable and executable. The input from the attacker must go to a writable page, but once on a writable page, it cannot be executed. So, even if the attacker convinces the processor to jump to the buffer, execution will stop. The program will crash, but the attacker will not get control.

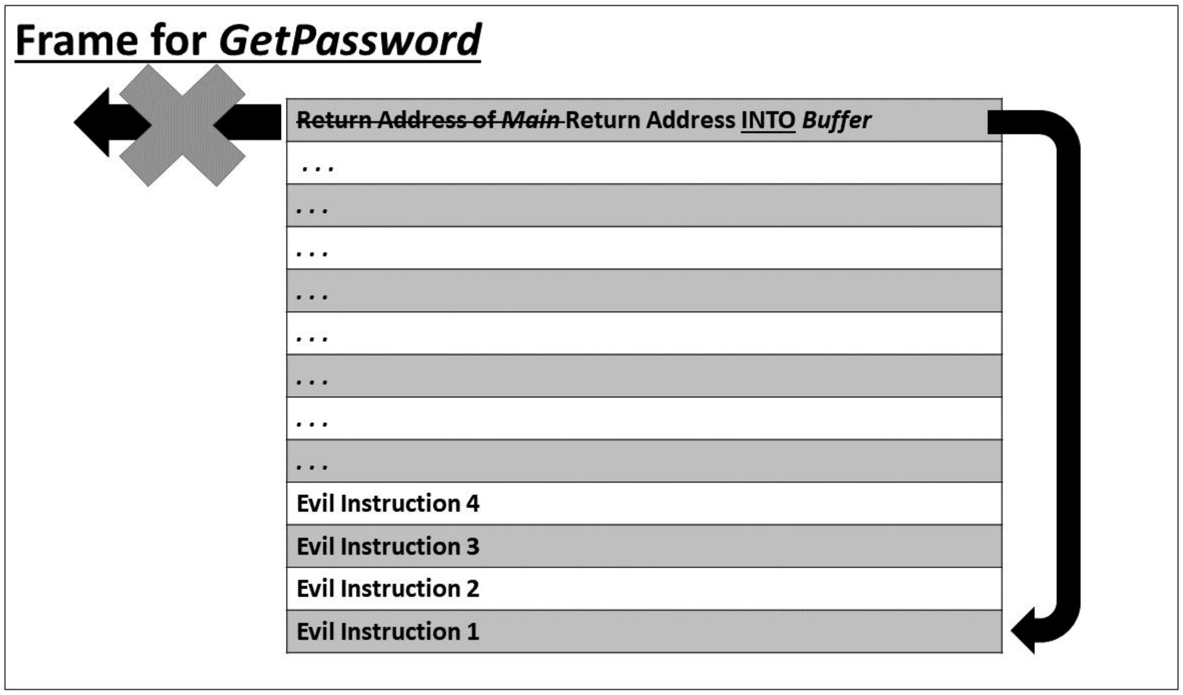


Figure 7-13 The same stack frame as in Figure 7-10 but with only the stack frame itself shown. This time, a malicious attacker inserted an overflow on purpose where the overwritten return address jumps *into* the buffer rather than back to the normal instructions. The stack is written with low addresses at the bottom, so the evil instructions are illustrated moving up the page

To get around these defenses, attackers use other approaches that include jumping to existing functions that are already executable. Almost every computer program, no matter how small or simple, will incorporate other standard code. For systems like MacOS, Linux, and other similar systems, there is a standard library of functions called *libc* (pronounced “Lib C” and refers to the “C standard library”). On Windows systems, there are equivalent libraries. The reason for having these standard libraries is because there are basic operations that almost every program needs as well as interfaces for interacting with the operating system.

One example of such an interface function is the ability to launch other programs. It is not unusual for running programs to need to launch and interact with other running programs. Because only the operating system can perform the actual launch operations, programs need to have an interface to signal the OS to do so. Because this is part of the standard library, almost every program will have this code available to it in memory. Attackers that overflow a buffer can overwrite the return address of their current subroutine to jump to the start of one of these program launching functions. The attacker then uses the function to launch certain types of programs called *shells* that provide the attacker with general control of the machine. From a shell process, the attacker can examine files, launch additional processes, transfer data off the machine, and so forth. This kind of attack is called the *return-to-libc* attack because *libc* is so commonly exploited [162].

More modern systems use a defensive technique called Address Space Layout Randomization (ASLR). ASLR randomizes the location of all the libraries in a running process. By randomizing the location of a library of functions, attackers cannot figure out where to jump to when they attack a machine. They may still be able to overflow the buffer, but they do not know the address of libc or its functions, so they cannot get the processor to jump to them. Again, the overflow may crash the machine, but at least the attacker did not gain control of the system.

There are weaknesses to some forms of ASLR, and there are a number of ways attackers can still get around them. One advanced attack method is called *Return-Oriented Programming* or ROP. ROP has been compared to when kidnappers send messages with letters cut out from different magazines. In ROP, attackers overwrite the return address to jump to the last instruction or two of some other function. Because it is at the end of the function, the subroutine will return and the processor will try to jump somewhere else. The ROP attacker has overwritten the stack such that when it returns from one function, it will jump to the next place in memory that the attacker wants it to go.

To help make this understandable, remember that each time a subroutine finishes, marked by a *return* instruction, the processor will look to the stack to see where it should go next. Because of the way buffers overflow, the ROP attacker can put a whole series of return addresses on the stack. The first address jumps to an instruction or two followed by a return. When that return happens, the processor goes back to the stack where it encounters the next address the attacker put there. Again, the processor jumps out and runs an instruction or two before returning.

This enables the attacker to basically create an *entirely new program* made up of bits and pieces of *other* programs. Each specific chunk of a function (again, usually one or two instructions followed by a return) is called a *gadget*. By stringing gadgets together, the attacker can program attack code to perform various operations. Usually, the goal of the attack code is to find a way to launch a shell just like in a standard return-to-libc attack.

Figure 7-14 illustrates chaining together gadgets in order to construct their own program.

The ASLR defense discussed previously is supposed to prevent ROP. ASLR, if complete and thorough, can make ROP much harder. However, gadgets within libc are all *relative* to the start of libc, even if libc itself is randomized. If the starting address of libc can be found, ROP can be used effectively. The address of libc can sometimes be found through other vulnerabilities defeating the value of the ASLR.

Another problem with ROP is that in many large programs the code is built up from components. It is sometimes the case that some components are randomized and some are not. ROP attacks can be launched at the part of the program that is not randomized, and those gadgets can be used to find the randomized addresses of the other components within the system.

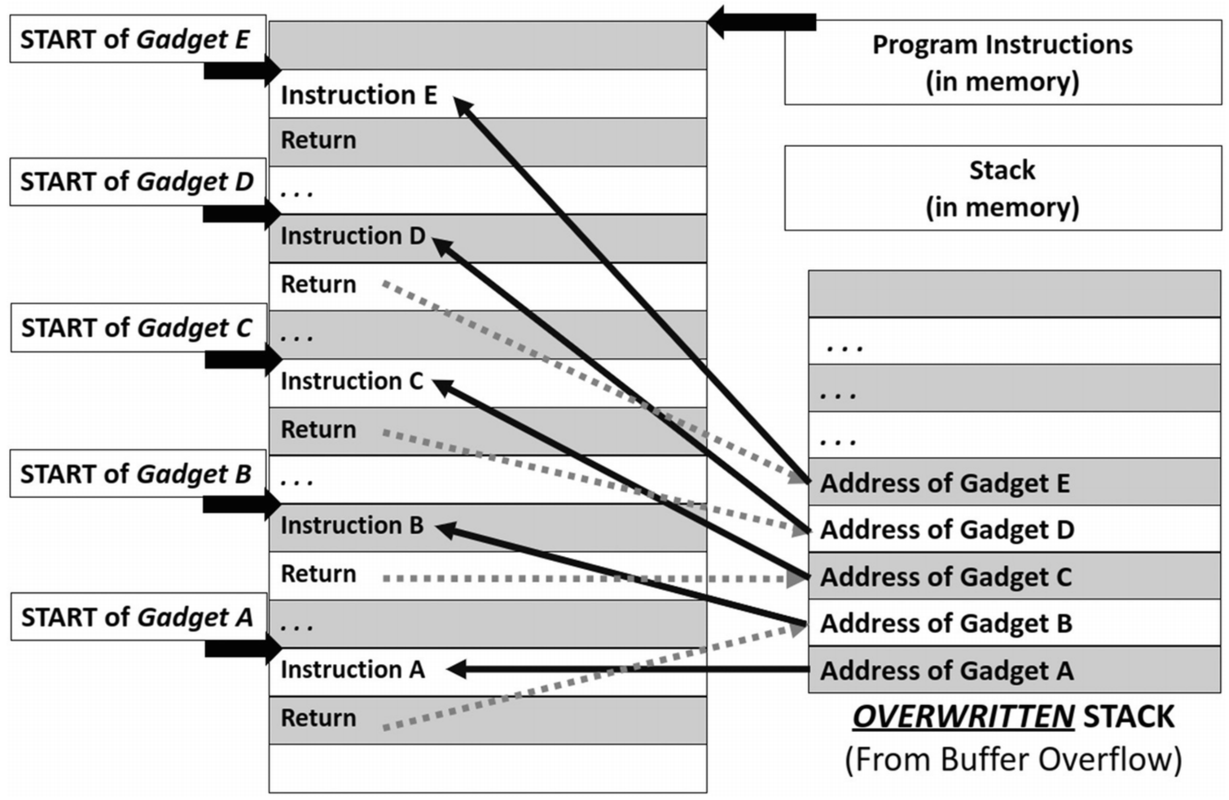


Figure 7-14 The stack contains various addresses to gadgets. When the program goes to these addresses, it executes a few instructions and then returns to the next gadget on the stack

Another place where ROP is still very successful is on IoT systems. Many IoT systems have limited ASLR [142]. And others may have ASLR, but the ASLR is not effective. This can happen when, for example, the address space is too small and there are too few possible address options for a library to be randomized to. For example, if a library can only be randomized to a few thousand locations, it is fairly simple for a ROP attack to brute-force search for where it is. The ROP attack can easily be launched a thousand times in an automated fashion.

Although ASLR and other modern defenses are helpful, it has not stopped control-flow hijacking attacks.

This should help illustrate, however, why isolation of software is so important. If an attacker takes control of an *unprivileged* process, they can still only execute unprivileged operations. Or, with even stronger isolation systems like those found in iOS or Android, they should be confined largely to their own sandbox and unable to interact with or interfere with other processes.

Malware Classifications, Impact, and Scope

Of course, just like users interact with computers using software, so do the attackers. When attack software is running a user's own machine, it is called *malware*. Malware is defined by NIST to be

> Software or firmware intended to perform an unauthorized process that will have adverse impacts on the confidentiality, integrity, or availability of a system [121].

There are a number of ways of classifying malware. Before about 2010 or so, malware was primarily described by how it spread. More recently, however, many of the malware infesting cyberspace is characterized by what it does to the user or for the attacker. Nevertheless, the early forms of malware still exist, and those categories still are in use. It is also true that many of these malware types do not have “bright line” distinctions. Many of malware can be classified with multiple labels.

Viruses

Probably the most “classic” of the malware classifications is a *virus*. For the better part of a decade or so, viruses practically defined malware. Viruses made such an impact on security that even today the term “virus” is often used to refer to malware in general, even if it does not meet the classic definition. And many security products that defend against malware still call themselves “antivirus” (AV) products.

The term “virus” and the basic description of a computer virus were described by the security researcher Fred Cohen in the 1980s. Cohen discussed the idea of a piece of software that inserts itself into other software, much like a biological virus inserts itself into normal cells [76]. Cohen’s security research demonstrates a relatively rare success in accurately predicting future computer security problems as computer viruses hardly existed at the time of his analysis. Sadly, many of the biggest computer security challenges are not anticipated and come as a nasty surprise. It is remarkable that Cohen was able to do so with viruses.

In more detail, the classic computer virus was not a program that could be run by itself. Rather, it was a set of instructions meant to be inserted into the instructions of other programs. In an overly simplistic description, imagine a program as a set of instructions as described in this chapter. Evil instructions (i.e., the virus code) could be inserted *at the end* of the program. Then, the first instruction (or an early instruction) could be replaced with a *jump* command. Remember that branching instructions can be conditional or unconditional. A jump is an unconditional branch that immediately jumps to a new address of a different instruction. So, by replacing the first line of the program with a jump, the program skips the normal code and can jump to the end where the virus code is instead. Once the virus code has finished executing, it can call a jump instruction back to the beginning of the program. If necessary, the first line (which was replaced with the jump instruction) can be appended to the end of the virus code so that it executes before jumping back to the rest of the real program's instructions.

The reason for this jump to the end and then jump back approach is because insertion into existing data is actually really difficult for a computer. If the virus wanted to just insert itself at the beginning, it would have to *copy* all of the existing program further into memory to open up space. It is much easier to replace an early instruction with a jump, copy the virus at the end, and copy the replaced instruction after the virus code followed by a jump back to the beginning as shown in Figure 7-15.

Because the virus code executed first and then returned to normal operations, a user might not even be aware something went wrong. After all, the normal program would behave as it should once the virus code was finished. A user would only notice the virus when it made itself known in some way or form.

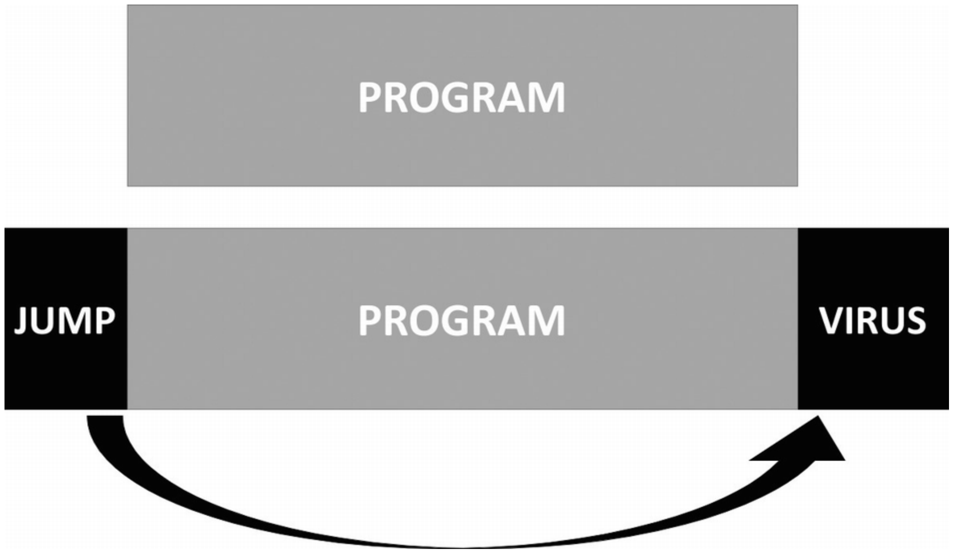


Figure 7-15 When a virus infects a program, it may modify it to have malicious instructions near the beginning and the end

The execution of the virus code itself would typically do two things. First, it would spread itself to any other programs it could find on the computer. Second, it would perform whatever mischief it was designed to do.

Early viruses primarily targeted the DOS operating system that was commonly used in the late 1980s and early 1990s. DOS was a very primitive operating system and did not even have privileged vs. unprivileged operations. Malware could easily take control of the entire system. At the same time, in this early, pre-Internet age, it was very rare to have software running that was not explicitly started by the user. Part of the way viruses worked was to copy itself into as many programs as possible so that any program the user ran would execute the virus. Although the Internet was not available, viruses could spread by shared disks and, for those relatively few people that had them, through modems, devices that connected to networks over the phone line.

Another interesting fact about early viruses is that they did little to no damage. Most of them were not all that destructive. For example, there was a virus known as the *8*

Tunes virus. This virus would play one of seven random German folk tunes every seven minutes. Annoying to be sure, but relatively benign. There were destructive viruses as well, but it is amazing how many of the viruses of the early 1990s appeared to be practical jokes and attention stunts for the authors [75].

As computer systems evolved, so did the viruses. In 1995, the first *macro* virus appeared in the field, called the “Concept” virus [191]. The term “macro” refers to a kind of script that can be inserted into office documents, especially Excel spreadsheets. This was a big deal because, up until this time, antivirus software only had to examine program files (also called “executable files” or just “executables”). But now, with the introduction of the Concept virus, antivirus software had to begin examining documents as well.

Concept was another example of a virus that did no additional harm. Its only operation was to spread itself around. A common way it spread was via email, which was just becoming common in offices and other professional environments. When a user received a document with this kind of virus in it, they would commonly just launch the document directly from their email. The macro would execute automatically when the document was loaded and would corrupt other office files. When the user emailed a document to someone else, the virus would infect a new host.

Throughout the late 1990s and early 2000s, viruses have proliferated in both number and form. With that said, the classic virus, and by classic I mean malware that inserts itself into other programs or data and attempts to replicate when run, has become less and less common. But because “virus” was the term that defined malware (and “antivirus” the term that defined anti-malware) for more than a decade, the term is sometimes used to refer to any malware generally. For example, the security company Fortinet defines a virus to be a “malicious software, or malware, that spreads between computers and causes damage to data and software” [27].

In my own security investigations and research, I have seen little of this early form of malware. The reasons are probably because it simply is not needed anymore. As I suggested earlier, viruses made more sense when there was a need to spread it as far and as wide as possible within a user’s own files. This was necessary to ensure it would be both spread and launched. Now, however, attackers have so many more effective delivery mechanisms that the passive execution and infection is simply archaic.

I stress, however, that the term “virus” is still widely used. It is just used to describe malware that classically was not virus-like behavior. Many of the other malware types discussed in the following sections are often described as “viruses” in various sources. For the purposes of this chapter, however, I will use the term virus to refer to the classic meaning.

Worms

A *worm* is another classification of malware that has some similarities with a virus. For example, a worm seeks to propagate and replicate across as many computers as possible. The difference is that, instead of attaching to files and waiting for passive activation, a worm is typically self-contained and actively attempts to spread.

Worms often rely on *vulnerabilities*, such as the buffer overflow attacks described in Section “Software Vulnerabilities”. The name “worm” was chosen because these bits of bad software seemed to tunnel through the Internet and in between systems. Once a vulnerability is found, it is not difficult for a piece of malware to spread itself far and wide using the vulnerability as a way to get past defenses and isolation techniques. Apparently, the concept of a computer worm is first discussed in the 1975 science fiction book, *The Shockwave Rider,* by John Brunner [247].

One of the earliest worms happened in 1988 before most American families had personal computers and before there was a World Wide Web. Nevertheless, the impact was sufficiently significant that it garnered media attention. The publicity was also increased by the family relationships of the worm's author: Robert Morris.

Figure 7-16 Sun Microsystems 3/80 workstation

The Morris worm, as it came to be called, was embodied in a computer program that was specifically designed to replicate on machines common for the time period. In 1989, only 15% of American households had a personal computer [154], and the vast majority of those machines were not hooked up to any kind of network at all, let alone the Internet. Although the World Wide Web did not yet exist, the Internet did. It connected the networks of researchers at universities as well as certain industrial and military organizations. The types of machines common to these systems, and targeted by the Morris worm, were Sun Microsystems Sun-3 systems and VAX computers (Figures 7-16 and 7-17).

Figure 7-17 Digital Equipment Corporation VAX 11/780 minicomputer and terminals (Source: Bernd Gross, CC-BY-SA-4.0, https://commons.wikimedia.org/wiki/File:VAX_11-780_und_Nachentwicklung_K_1840_TSD.JPG)

The types of machines in use at the time ran an operating system somewhat like Unix called BSD.[5] This 1980s vintage OS had no graphical interface. Users interacted with the system using text-based commands. Nevertheless, it was their connectivity to the Internet that made them targets for a worm designed to spread.

An infection of the Morris worm worked in stages and consisted of two different programs. The first program was a bootstrapping program, also called *the vector program*. The second program was the main program of the Morris worm that attempted to replicate itself. The vector program was designed to prepare the victim machine for infection and to contact an already infected machine to download and install the main program. In other words, the vector program's job was to get the main program installed and operational on the infected host.

The Morris worm inserted the vector program into victim machines using three techniques. Using the first method, the Morris worm exploited a buffer overflow attack in a system called *finger*. The finger program was an early form of *online presence* management. A finger server ran on many of the computers connected to the network. A user on one machine could connect to the finger server on the second machine to ask who was logged in on the second machine. But the Morris worm could overflow a buffer in the finger server and subvert the program. Once compromised, the finger program would give the Morris worm access to a *remote shell*. This is basically a remote text entry system for entering commands into the compromised system. The Morris worm would use this remote shell to send over the code for the vector program and get it operational.

A second means for getting the vector program installed was a bug in the *sendmail* program, used with email. In an era before webmail, transmitting and routing email was of critical importance. The sendmail program, however, had a little known feature that would permit a remote party to put the sendmail program into a debugging (or testing) mode and transmit commands. The commands could be exploited to install the vector program.

The third and final approach for compromising a remote host did not rely on a bug at all. Instead, it relied on attempting to crack passwords. Like with modern password lists, passwords on the BSD machines were stored in what was effectively a hashed format. The Morris worm could look up a username and a password from a compromised machine and try to crack the password with smart guesses (such as the username as the password), guesses from an internal dictionary of 432 words, and guesses from a dictionary file that already existed on the compromised machine (i.e., a dictionary intended for legitimate users of the machine). Cracking passwords on a machine that is already infected may seem odd, but then, as now, users reused passwords across systems. If the Morris worm could crack a password on the infected machine, it would then try to connect remotely to other machines using the same username and password combination. Machines of this era had files identifying which remote machines were commonly connected to, and the Morris worm could try the cracked passwords with a username on the remote machines to see if the user had an account there with the same credentials. If the username and password worked, the Morris worm could transfer the vector program directly.

Once the vector program was installed using any of these three methods, it connected back to the machine that installed it. Using a network connection, it downloaded and installed the main program. The victim machine was now fully infected. The Morris worm would now use this new machine to find new victim machines and replicate further.

Like many malware of the era, the Morris worm was not designed to actually do damage on purpose. Unfortunately for the Internet of 1988, the Morris worm had a problem of its own. As you can probably imagine, the design of the Morris worm lended itself to a lot of *reinfection*. If machine

A infects machine B, B can infect A. If A infects B and B infects C, C may have its own connection to A and infect it. This problem is illustrated in Figure 7-18.

The problem with reinfection is that it results in *two* copies of the Morris worm running on the same machine. And, if then infected again, *three* copies of the Morris worm would be running at the same time. The author of the Morris worm understood the problem because the worm was programmed to check if the system was already infected. If so, the duplicate worm was supposed to quit. However, Morris wrote the code so that one out of every seven times it would ignore the result. It has been theorized that this was a defensive measure to prevent system administrators from easily killing the worm by simply faking an infection.

Whatever the reason, the number of Morris processes multiplied on each machine causing a *denial-of-service attack*. Effectively, this brought much of the Internet of 1988 to a halt. According to the FBI:

> The worm did not damage or destroy files, but it still packed a punch. Vital military and university functions slowed to a crawl. Emails were delayed for days. The network community labored to figure out how the worm worked and how to remove it. Some institutions wiped their systems; others disconnected their computers from the network for as long as a week. The exact damages were difficult to quantify, but estimates started at $100,000 and soared into the millions [103].

The effect of the Morris worm was sufficient to warrant media attention. Front page news and TV news anchors discussed the event and the impact. The media buzz about the worm was only heightened when the author, Robert Morris, was identified. Morris, it turns out, was the son of the chief scientist for a computer security arm of the National Security Agency of the United States. *The New York Times* had a front page story on November 5, 1988, three days after the Morris worm was released, with the headline: “Author of Computer ‘Virus’ Is Son Of N.S.A. Expert on Data Security” [174].

The Morris worm was effective for its time. But worms became able to spread even faster once more computers were connected to the Internet and Internet communication became more commonplace. Unlike the Morris worm which depended on some relatively technical attack systems, the new worms could spread using some fairly simple social engineering to spread by email. The “I Love You” worm, which appeared in the year 2000, was one of these. The virus was contained within a Visual Basic script attachment called “LOVE-LETTER-FOR-YOU.txt.vbs”. The “vbs” extension is an indicator that this is a Visual Basic script. Then, as now however, Windows often hides the extension, so all users would see in their email was “LOVE-LETTER-FOR-YOU.txt”. Thinking it was only a text file, many users assumed it was benign (text files generally cannot be malicious). Once the user double-clicked on the attachment, it would start executing the VBS script. Once running, the script would scan Outlook for contacts in the address book and automatically mail copies of itself to everyone in the list.

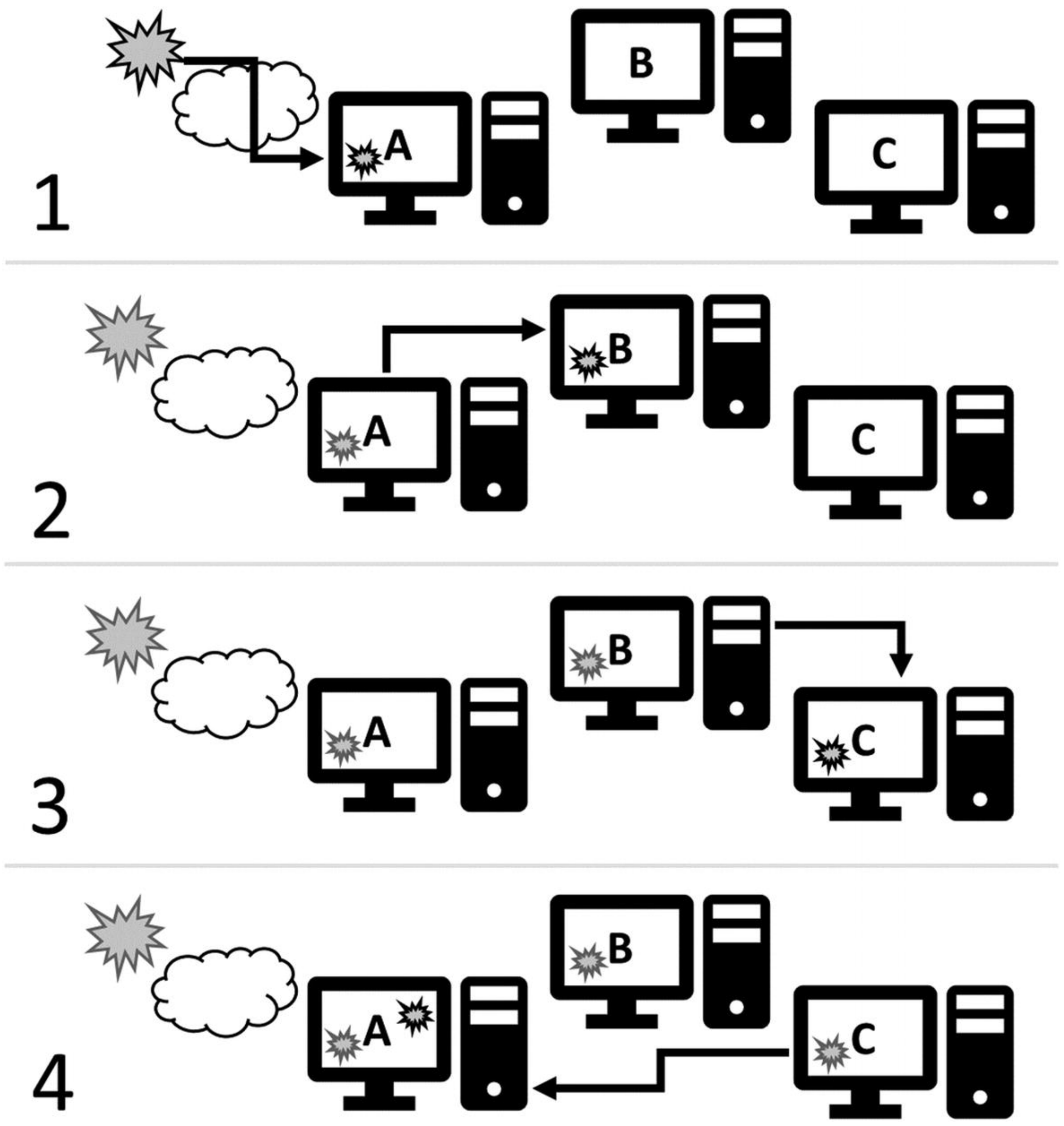


Figure 7-18 Sample reinfection: (1) A gets infected through the network. (2) A infects B. (3) B infects C. (4) C reinfects A, which now has two copies of the worm

This type of worm is called a *mass mailer*. It spread across the world very rapidly. It was released into the wild on May 4, 2000. In five hours, it spread worldwide. Like the Morris worm, there was no way to prevent reinfection, so inboxes were getting slammed with thousands of copies of the message. Government bodies, like the United

Kingdom’s House of Commons, had to turn off their email servers because they had become useless. So did technical companies, like Microsoft [118].

This denial-of-service attack on inboxes affected almost everyone, as anyone with a Windows machine and Outlook (and in an infected connection’s contact list) was a target. But for those that did not click on the attachment, the limit of the damage was relatively small. They would only lose email and perhaps Internet access for a short time.

But for those individuals that did click on the attachment, the result was devastating. The worm deleted files including image files, program files, and other common file types. Many people had no backups of any kind. These individuals suddenly found their personal data destroyed forever with no hope of recovery.

Sadly, the I Love You worm was not the end either. In 2004, a new mass mailer emerged called the MyDoom worm. MyDoom was released on January 26, 2004, and like the I Love You virus, it appeared as an email message with an attachment. The initial version has subject lines such as “Error,” “Mail Delivery System,” “Test,” or “Mail Transaction Failed.”

In this time period, getting emails like these was common, and it was not unusual to investigate them. In fact, the message itself is relatively uninteresting, but it was often enough to get users to look and see an attached message. The icon for the attachment was the same as a text document even though it was a program. Once launched, the worm would attempt to spread through a lot of different methods. It would attempt to harvest email addresses from more than just the contact list too. MyDoom checked web-related documents in the browser’s cache as well as the hard drive generally. It was also willing to try some random email names with the same domain.

Interestingly, it also avoided certain domains and email address names that suggested more technical users. In other words, it was specifically targeting less savvy users that would be more likely to click on the attachment. Clearly, a lot of thought was put into the infection mechanism.

The results of this planning and thought clearly worked. In the first 24 hours, 1.2 million MyDoom emails were transmitted. At the time, 1.2 million emails represented almost 10% of email traffic. The result of this deluge was so extreme that Internet traffic slowed, resulting in a noticeable reduction in loading speed for websites [259].

Amazingly, MyDoom emails *are still being sent*, at least as of 2019. In that year, researchers from Unit 42 (associated with Palo Alto Networks) estimated that 1% of all malicious traffic was still MyDoom [201].

In the interest of space, I have left out a lot of additional details about MyDoom including its ability to create a bot network. I will discuss bot networks separately later. But the point is, MyDoom was a terror in 2004, and the echos of it can still be felt today.

Worm tactics continue to be used by malware today. Some ransomware, for example, have worm-like capabilities. I'll cover those in a separate section as well. But in conclusion, attackers still figure out ways to slice and dice through security, often employing social engineering and vulnerabilities to spread their malware through networks.

Trojan Horses

One of the earliest classifications of malware was the *Trojan horse*. A Trojan horse is malware that masquerades as something benign or desirable. In other words, it looks like a normal computer program, but when it runs, it does some undesirable operations. The Trojan horse classification is possibly the least specific label that can be applied and one that can be applied to many other malware, as will be seen throughout this section.

However, it is useful to understand the various principles of psychology discussed in Chapter 1. Very often, a Trojan horse malware is made to look enticing to the user or otherwise motivating them to action. Trojan malware has been hidden inside files purporting to be pornography, video game components, music and video files (usually downloaded illegally), and other files that tend to get a lot of downloads [62]. More modern examples of a Trojan horse are apps sold through iOS or Android stores that appear to be legitimate apps but perform undesirable behavior once installed [202].

The concept of a Trojan horse has been a concern for computer security professionals from a very early date. As discussed in Chapter 3, the Bell-LaPadula model (BLP) *assumes* that Trojan horses might be running in the system. The reason for the “no-write-down” policy was, at least in part, to prevent unknown but evil software from exfiltrating data to a lower level of sensitivity [40, Chapter 9].

In a consumer operating system, Trojan horse software should be limited in the damage it can do by the OS. For example, a Trojan horse that is executed normally should be running unprivileged. However, clever Trojan horse software can try to run as administrator. Even if it has to ask for permission, the typical user may not understand the danger of providing administrative permissions. If a Trojan horse can run as an administrator, it can do significantly more damage. Of course, even an “unprivileged” trojan can do massive damage to the user’s data by deleting or modifying it. But if it can run as administrator, it can work to circumvent all security controls and hide itself from security software and other defenses using the privileged powers of the OS.

Because of the potential damage that Trojan horse malware can do to a system, some systems attempt to employ additional isolation techniques. Versions of Windows since Vista attempt to protect system integrity using a model similar to Biba (see Chapter 3). Programs downloaded over the Internet are considered lowest security level and, in accordance with the no-write-up policy, may not modify any data of a higher security level, such as the user's data and, of course, the operating system's files.

More recent versions of Windows with adequate hardware support enable some partial virtualization by default. Instead of creating a completely separate virtual machine, the late versions of Windows 10 and all versions of Windows 11 have options to partially virtualize the operating system. This approach is called "Virtualization-Based Security" or VBS, and when VBS is enabled, various additional security options can be enabled. But the basic idea is that certain core features of the OS run at a lower level, and the other parts of the OS, including third-party code called device drivers, run at a higher level. This permits additional isolation and security checks to prevent Trojan horses that get administrative privileges from being able to alter these core parts of the system [183–185].

Rootkits

A *rootkit* is a slightly different classifier for malware than virus or trojan because, unlike those two terms, it does not describe how it spreads. Instead, the term "rootkit" refers more about where or how it is installed in a user's system. The word root, as used here, refers to "root access." Root is a concept on Unix and Linux machines. It is comparable to administrator on Windows machines. Rootkits are so called because they aim to get root or root-like privileges. This makes them exceptionally dangerous.

Recall that the whole point of an operating system is to make everything else run unprivileged. If an unprivileged program has an error or, worse, is malicious, the damage is confined to at most the data of the compromised user. Other users and the system should remain more or less intact. Additionally, malicious software should be detectable and removable by security tools.

A rootkit undoes all of this security by being effectively inserted into the operating system itself. Actually, some rootkits aim to go even lower than the operating system by infecting the startup code (called boot code) or even low-level firmware that is installed in hardware components. But for the brief overview of this chapter, I will simply talk about OS-level malware. The other types of rootkit are based on similar ideas and have similar impacts.

To illustrate the problem of rootkits, I am going to use two different examples that are strikingly different than the other examples I have used so far. Both of the rootkit examples I walk through were created by *legitimate* companies for presumably nondestructive and nonmalicious reasons.

The first example happened in the 2005 era. At the time, music services like iTunes were in their infancy, and compact discs (CDs) were one of the primary means by which consumers purchased and consumed music. Concerned about the prospect of rampant consumer copying and distributing of music tracks, Sony BMG decided to equip their audio CDs with one of two copy protection systems. One was called XCP (Extended Copy Protection), and the other was called MediaMax CD-3. Both were extremely problematic.

In terms of rootkits, however, XCP ("Extended Copy Protection") fit the bill. When a user inserted an audio disk with XCP into a Windows machine, it would pop up an End-User License Agreement (EULA) that users had to accept or the system would eject their disk. Thus, to play music legally purchased, users had to accept a license with Sony BMG. But the EULA did not describe all of the things XCP would do to a user's system or apparently any of the XCP software at all.

First and foremost, the XCP software was installed in such a way that it actually modified the operating system. The modifications permitted it to hide (cloak) itself. This meant normal ways of seeing that the XCP software were disabled. This included hiding files, registry keys, and the running processes themselves.

Second, the XCP software included some kind of "phone home" component that would transmit information to XCP-related servers.

The XCP software initially had no uninstaller, and the only way to get rid of it was to do some very deep and manual tweaking to Windows. Moreover, if parts of the software were uninstalled, the computer could no longer play CDs at all.

In addition to all the other problems of rootkits, they by nature create their own security holes and vulnerabilities. It was demonstrated that the XCP software could, itself, be attacked by malware that would piggyback on it. And generally, the XCP software's bugs just made the system wasteful and prone to errors [226].

In summary, rootkits are bad whether or not they are installed by a legitimate company. After a public outcry (and lawsuits), Sony recalled all of the problematic CDs and paid out settlement fees. They also released an uninstaller.[6]

You would think that companies would learn from these kinds of bad decisions. But the second example of a rootkit is somewhat similar. In 2016, it was revealed that video game company Capcom had bundled a rootkit with their game *Street Fighter V*. As with Sony, the justification for the rootkit was to prevent cheating. The rootkit basically *disables privileged protections of the OS so the unprivileged game can snoop for cheating*. Although not as intrusive as the Sony rootkit, the Capcom rootkit opened the door for the same kind of problems where other malware might attack the OS through Capcom's own system.

Fortunately, the Capcom rootkit was retired fairly quickly after discovery. While that is a positive development, it was interesting to see a news article state, "A lesson quickly learned" [282]. It would have been learned more quickly if they had learned from Sony's mistakes.

Ransomware

Over the last 15 years or so, the classifications of malware have shifted more from how they spread (e.g., virus, worm) to what they do. One of the strongest examples of this is *ransomware*. Ransomware is malware that locks up data or other system resources and demands money in exchange for the data's release. Ransomware is what the malware does, but it could spread like a virus, a worm, a Trojan horse, or other means.

The first recorded case of ransomware goes back a long time. Released in 1989 by one Joseph Popp, the ransomware was spread by disk. In fact, Popp literally handed out the disks at participants at the World Health Organization's AIDS conference. And, not coincidentally, the name of his ransomware was called "AIDS" (alternatively, Cyborg). The disks claimed to have information about AIDS. But when inserted into a computer, a message would pop up stating that the computer had AIDS. Files were locked and users were instructed how to mail money to an address in Panama to obtain their release.

Fortunately, Popp’s malware was poorly written. It used symmetric cryptography to lock the files, and, as you learned in Chapter 5, the same symmetric key for encryption is used for decryption. Because the symmetric key was still stored on the computer, it was easy to decrypt [196].

Although ransomware has popped up in one form or another since Popp, it really began to accelerate after 2010. At least one reason why is *Bitcoin*. Ransomware largely needs to combine three elements:

1. An effective delivery mechanism
2. Relatively fast public-key and symmetric key cryptography
3. A form of payment that is as untraceable and as unstoppable as possible

As demonstrated in this chapter, there has never been a shortage of mechanisms for spreading malware. That component has always been with cyberspace.

The cryptography necessary for good ransomware attacks has also existed for a while. However, the increases in speed as well as the increased understanding of how to use cryptography in the 2000s moved this requirement along. I will walk through a common cryptographic approach shortly.

But the third requirement was the hardest. There are so many ways to trace credit card transactions, money wires, and so forth. There are also numerous ways to block or cancel these kinds of transactions. But as Bitcoin became more mainstream, cyber criminals quickly discovered its value as a currency for ransom payments [156]. If Bitcoin did not lead directly to the explosion in ransomware, it is a very (un)fortunate coincidence.

As alluded to in Chapter 6, ransomware attackers have evolved their own cryptographic system [84, 173], and it is fascinating. Here is how a popular version called *hybrid cryptography ransomware* works. Refer to Chapter 5 if you need to review the details of any of the cryptography terms here.

To get started, presume that the ransomware authors, or at least those receiving payment, are operating some server R. At server R, they generate an RSA public and private key pair. For the purposes of this example, I will call these two keys *server_RSA_public* and *server_RSA_private*. The *server_RSA_public* key will be embedded in the ransomware software that attempts to infect victim machines.

When a victim machine is compromised by the ransomware, the ransomware will generate a new RSA pair. I will call this pair *victim_RSA_public* and *victim_RSA_private*. Remember that RSA public keys can encrypt (relatively small amounts of) data. The ransomware will use RSA to encrypt the *victim_RSA_private* key with the *server_RSA_public* key. Importantly, only the *server_RSA_public* key is on the victim's system ever. Anything encrypted by this key can only be decrypted by the criminals back at server R.

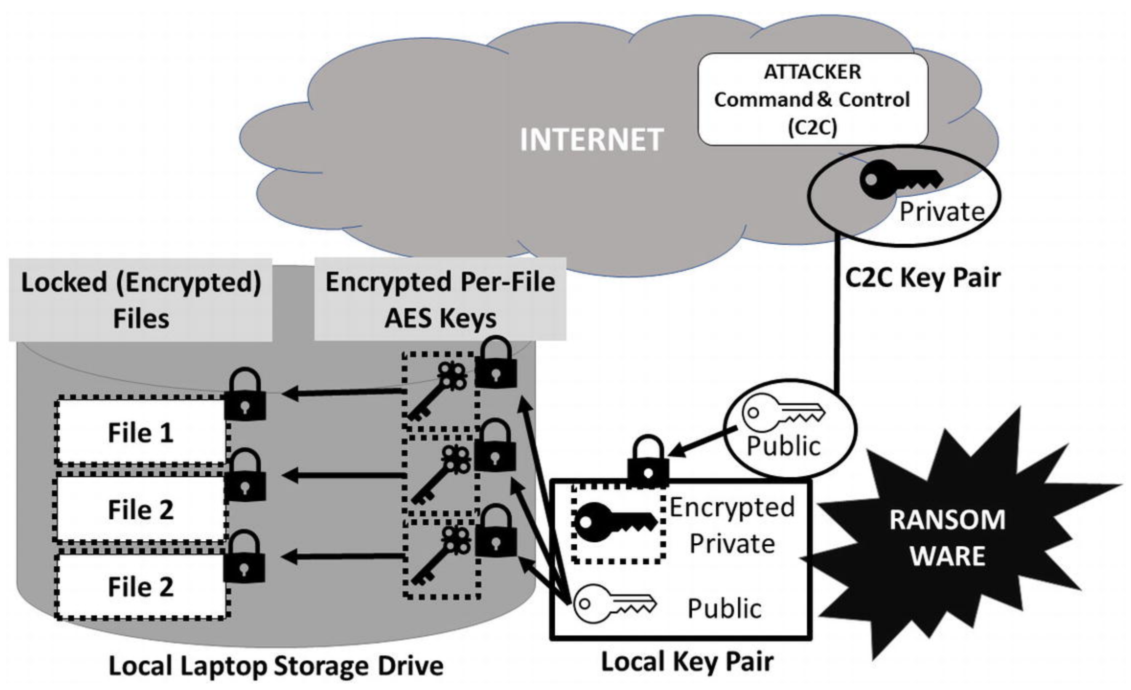


Figure 7-19 Ransomware encryption using hybrid cryptography. This variant locks each file with an AES key. Each AES key is locked by a local public key. The corresponding private key, which could unlock it, is encrypted with a server public key. The server private key is never on the victim computer

The ransomware next generates an AES key *for each file to be encrypted*. The key is used to encrypt its associated file using an AES algorithm such as AES-CBC. Once encrypted, the ransomware will encrypt the AES key using *victim_RSA_public*. So, in order to recover the encrypted file, a user would have to get the AES key. But to do that, they would have to be able to decrypt the AES key with the *victim_RSA_private* key. But that key requires *server_RSA_private* key, which is not on the victim's system. This configuration is depicted in Figure 7-19.

Once all of the files are encrypted, the victim is notified. If they choose to pay the ransom (and the criminals actually release the encrypted data), the ransomware on the victim's machine transmits the encrypted AES keys to the server along with the encrypted *victim_RSA_private* key to the server R. Here, the private key is decrypted, the AES keys are decrypted, and AES keys are sent back to the victim machine. The AES keys are then used to decrypt all of the locked data.

Notice that this cryptography approach solves a lot of problems:

1. Neither the victim machine nor the server R has to be connected to the Internet at the same time.
2. Files are encrypted quickly with symmetric key encryption.
3. Keys are per file permitting a few free decryptions to prove it can be done.
4. Keys are also per system so that revealing one does not reveal all data on all machines.

Ransomware has proven to be a massive threat. Some ransomware has developed worm-like capabilities [46]. Ransomware in 2020 had over 50% of its infections come from unsecured remote access to Windows (lock down your remote access!), but about 25% still came through email phishing [238]. Ransomware has attacked hospitals, businesses, public school districts, universities, and governments [128].

Bot Networks

The last type of malware I will talk about in detail is a *botnet*, which is short for a “bot network.” The concept of a bot is a computer that has been turned into a robot for the malicious attacker. These computers have sometimes been called “zombies” as well.

As with ransomware, a botnet is more of an end result. A botnet could be created through any means. The example I will use here is a worm called *Mirai*. The Mirai worm took advantage of the vast number of IoT (Internet of Things) devices becoming connected to the Internet. In the 2010s, there was an explosion of connecting devices like cameras and other IoT devices to the Internet. Unfortunately, most of these devices were *practically thrown* onto the Internet with very little thought about the computer security considerations. For example, many devices had default passwords left in place that could be used to easily subvert the device. And that is exactly what Mirai used to acquire its bots.

The main idea behind a botnet is that there is a Command and Control (C2) server that operates somewhere on the Internet. When a device becomes compromised, it connects to the C2 server and reports in. Once this happens, the compromised machine no longer needs to work on automatic. Rather, it can receive instructions from the C2 server, and its operations can be coordinated with the other computers in the botnet. The C2 also provides logistical support such as patches and upgrades to the botnet software itself. A victim bot machine will try to find other computers to compromise and turn into bots like itself. When these machines are subverted, they will also contact the C2 and join the network.

Mirai would scan the Internet for vulnerable IoT devices, get in using the default credentials, and then would block *other* malware from gaining access or eject malware that was already there! These IoT devices were so vulnerable that malware was fighting over which one would be in control of it. But for what purpose?

One of the primary uses of a botnet is a *distributed denial-of-service* (DDOS) attack. A DDOS attack usually involves sending a large amount of data, usually from a large number of computers, to a single source such as a web or game server. When the victim system receives all of the traffic, it overwhelms it and the server becomes unable to serve legitimate traffic. This can take a website offline, or it can cause various kinds of issues for the game running on the server. In the case of Mirai, the first targets were *Minecraft* game servers. *Minecraft* servers are very popular, and website owners can make tens, and even hundreds, of thousands of dollars from subscribers. Some *Minecraft* server owners paid the controllers of the Mirai botnet to DDOS competitors to drive their players out and over to (presumably) their own servers [88].

This is an important point. The people that *wanted* to do the DDOS were not the people in control of the Mirai network. Rather, the people in control of the Mirai network *offered their services for money to others*. Sometimes called DDOS-as-a-service, it means that malicious parties do not need to have their own botnets or even be particularly savvy. They simply need a target and money.

Scanning for IoT devices, Mirai was able to put together one of the largest botnets ever seen. According to sources, “the end result was a mammoth botnet of 200,000–300,000 enslaved devices capable of generating up to 1.1 terabits per second in junk traffic” [88]. If you do not have a lot of tech experience, you might be having trouble grasping just how big 1.1 terabits per second is. It is big. *Really big*. And sending that amount of data can take down even the most well-provisioned servers.

In fact, one such website attacked by the Mirai botnet was the blog of cybersecurity researcher and investigator Brian Krebs. His site, “Krebs on Security,” was even protected by a company that offered DDOS protection services. The total traffic sent at Krebs’s blog was too much even for them, and the system did go down for a time. It should be noted that Krebs got his revenge. Through his extensive research, he was basically able to identify two of *the authors of Mirai’s source code*. This led to the arrest and conviction of Josiah White and Paras Jha, who pleaded guilty to the charges[7] [88].

Malware-Specific Defenses

This chapter has focused so far on concepts of isolation used by host systems for defense and a survey of the malware that tries to bypass them. But there are specific anti-malware strategies that are also widely used and are sometimes better known than isolation because they are more visible.

I prefer to group these strategies into three categories based on *when* in the malware's life cycle they begin:

- **Identify and neutralize** strategies attempt to defeat malware *before* the malware is either active or actively attacking the system.
- **Mitigation** strategies are designed to reduce the effects or duration of attacks that are in progress.
- **Recover and respond** strategies engage after the attack is over with goals to undo damage caused by the attack and/or prevent future attacks.

Please note that these terms are not industry terms. They are my own formulations.

Identify and Neutralize Strategies

In order to stop malware from being able to attack and damage the system, the malware must first be identified. That may sound like a simple task. In fact, it is overwhelmingly challenging. There are even theoretical proofs that show it is impossible to identify *all* possible malware.

Another way of saying identification is *classification*. The malware classification problem is, *given a program, or set of instructions, can the classifier determine if it is malicious or benign*? Over the three plus decades of commercial anti-malware research and industrial development, there have been a wide range of classifiers created. They tend to fall into one of two categories: classifiers based on static analysis and classifiers based on dynamic analysis.

Static Analysis

Static analysis classifiers are designed to analyze malware that is *not running*. That is, a static classifier analyzes the instructions being classified without running them. This category of classifier includes classic antivirus software based on *signatures* of known malware. It also includes systems that attempt to predict what the instructions will do when running based on *heuristics*. A heuristic is any kind of rule, measurement, estimate, or other shortcuts to a longer or more complicated operation. Actually running the instructions on a processor with real user input is the only way to know exactly what a given program will do. Heuristic rules attempt to make “good guesses” about what a program will do based on *telltale* signs or indicators in the instructions.

Since the beginning of anti-malware technology, signatures have been the predominant method for protecting systems. The first approaches were quite unsophisticated. This included publishing actual binary sequences of known malware in printed publications. For example, *Virus Bulletin* began publication in 1989 and is still an active anti-malware organization. When they first started publishing, the World Wide Web did not exist. Their primary publication was printed. The PDFs of these publications are still available at their website, and you can still see where they used to publish the early virus signatures.

For example, the “8 Tunes” virus I mentioned earlier in this chapter can be seen identified in the January 1991 *Virus Bulletin*.

```
33F6 B9DA 03F3 A550 BB23 0353 CB8E D0BC
```

This is hexadecimal, of course, and represents the binary bits that appear in a file infected with the 8 Tunes virus. As I explained about these viruses, they cannot execute on their own and are not separate stand-alone programs. Rather, they are instructions inserted into existing programs. So armed with these kinds of signatures, computer administrators could scan their files for these sequences. If they found them, they knew they had an infection. This was a very manual process, and often it was a manual process to remove the virus and repair damaged files. Repairing basically means removing the virus instructions from a file to restore it to its original form. Or, sometimes, it was best just to reinstall the program.

Antivirus software evolved to automate this process. Early antivirus products, such as McAfee VirusScan, could automatically look for these signatures and, in many cases, repair the damage. By the mid-1990s, there were many antivirus (AV) software programs attempting to combat the thousands of viruses floating around. Despite a general lack of Internet access, viruses spread via disks and over modem connections. The AV programs collected large libraries of samples to identify and remove viruses.

The virus authors got smarter too. To eliminate easy signature matching by identifying a well-known sequence of bytes, virus authors started to create *polymorphic* viruses. A polymorphic virus involved a couple of components. First, the main virus was “encrypted,” although encrypted does not mean the strong cryptographic encryption described in Chapter 5. It was just strong enough to make the data unrecognizable. And it was encrypted *differently with every infection of every file*. This means that the virus could not be found with simple scans because the bytes would be different in every file.

The polymorphic virus also needed a decryptor that would decrypt the virus and run it. The decryptor also needed to be unrecognizable and different in each infection, or virus scanners would just scan for the decryptor. So included in the encrypted virus was a *mutation engine*. The mutation engine, once decrypted, would mutate the decryptor. Mutate is not encryption. It simply reorganizes the instructions, or adds unnecessary instructions, in a way that does not change the functionality. This is similar to shopping at the store. It does not matter if you go down aisle 1, 2, or 3 first, you will end up with the same items in your cart. Similarly, the instructions for the decryptor could be rewritten in many ways, and the mutation engine could produce a more or less custom decryptor. Armed with a custom decryptor and a new key, the new file could be corrupted with a different decryptor and a different ciphertext virus (Figure 7-20).

In order to capture polymorphic viruses, antivirus tools had to resort to simulating the execution of part of the code. Viruses had to insert their jump near the beginning of the program to ensure they took control of the software before any branching started. This meant that if the AV started to execute instructions, or simulate the execution of instructions, it was likely to encounter virus instructions very quickly. By executing instructions, it could (relatively) quickly execute the decryption instructions. The very presence of a decryptor usually meant an infected file, but the AV would scan the decrypted data to determine which virus was installed. That was necessary for proper cleaning.

The simulated execution of instructions required the use of an *emulator* for the processor. In other words, the antivirus created a small sandbox by virtualizing the processor. Within the sandbox, the AV could execute the instructions without risking any harm to the system. Even though this is partially executing the file, which I described earlier as dynamic analysis, this is still primarily a static technology. The simulated execution was only meant to get past decryption so that normal signature scanners could make a definitive determination of the infection.

In fact, at least some signature scanners created signature definitions that were more like small computer programs. Rather than a sequence of bytes, the signature would have instructions like “check byte 10, if 0 jump forward 30.” The signature could also have decryptor instructions built in. But even though the signature is dynamic, this is still static analysis on software that is not yet running.

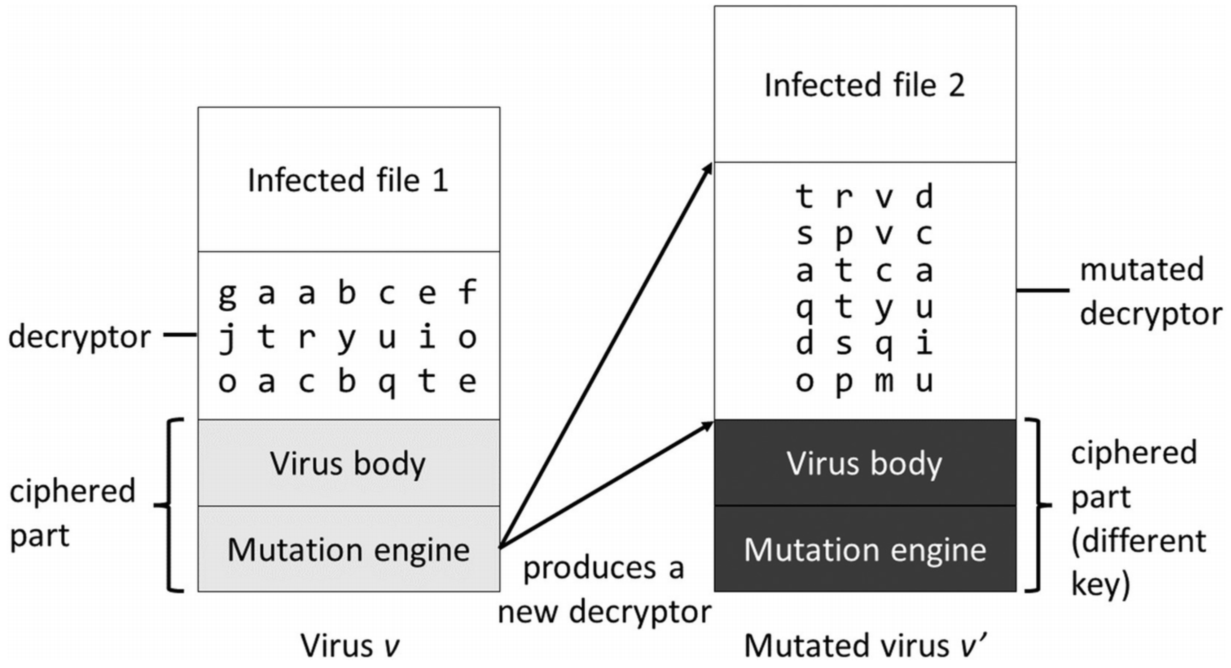


Figure 7-20 Polymorphic virus infection process. Source: Based on [72, Figure 1]

Signature scanners had two major problems. First, these scanners could not detect any *new* viruses. Until a virus had been analyzed and a signature extracted, the defense could do nothing. The second major problem was just keeping up with the large library of signatures that must be maintained. Although in contemporary systems memory is less of an issue, in the mid-1990s there was quite a bit of concern about antivirus software with large libraries. Not only did they chew up memory but it also made scanning slower because every file scanned had to be compared against every signature.

This led the more widespread adoption of a different scanning technique called *heuristic rules*. As described earlier, a heuristic rule is a “good guess” based on telltale signs. A paper written by Dmitry Gryaznov in 1995 explained a bit about the problems with signature scanners and how heuristic scanners were supposed to solve the problem:

> At the beginning of 1994 the number of known MS-DOS viruses was estimated at around 3,000. One year later, in January 1995, the number of viruses was estimated at about 6,000. By the time this paper is being written (July 1995), the number of the known viruses has exceeded 7,000. Several anti-virus experts expect the number of viruses to reach 10,000 by the end of the year 1995. This big number of viruses, which keeps growing fast, is known as glut problem and it does cause problems to anti-virus software, especially to scanners. Today scanners are the most often used kind of anti-virus software. Fast growing number of viruses means that scanners should be updated frequently enough to cover new viruses. Also, as the number of viruses grows, so does the size of scanner or its database. And in some implementations the scanning speed suffers [122].

Gryaznov explains further, “Today more and more anti-virus software developers are looking towards heuristic analysis as at least a partial solution to the glut problem.” And he provides an explanation of what heuristics are: “In the anti-virus area, heuristics are a set of rules which should be applied to a program to decide whether the program is likely to contain a virus or not” [122].

In this paper, Gryaznov walked through a couple of simple heuristics that help to explain the idea:

1. The program immediately passes control close to the end of itself;
2. it modifies some bytes at the beginning of its copy in memory;
3. then it starts looking for executable files on a disk;
4. when found, a file is opened;
5. some data are read from the file;
6. some data are written to the end of the file.

After explaining the basic concept of viruses, these rules should make sense. The first rule is about the instruction to jump to the end. The second rule might not make sense at first because these viruses do things a little different from what I described. Instead of including the overwritten statement at the end for execution, it is actually written back into place at the beginning of the file *in memory only* so that when the virus jumps back to the beginning, the program appears to be completely unaltered.

The other rules are what the virus does to spread. It looks for executable files, opens them, and writes data to the end.

Gryaznov explained that a new virus could be analyzed according to these steps. If it was found to have instructions that perform these kinds of steps, it has a very good chance of being a virus, even if it has never been seen before [122].

Heuristics were touted as an effective solution to unknown viruses and the overwhelming number of virus definitions. Unfortunately, they did not live up to their promise. There are some fundamental reasons for this.

First, heuristics are, in fact, *guesses*. And sometimes they guess wrong. Similar to the problems associated with biometrics discussed in Chapter 2, a false positive is when the heuristic falsely guesses something benign is a virus. A false negative is when the heuristic falsely guesses that a virus is benign. Both are problematic but in many ways the false positives are worse. False positives force the user, most of whom are not virus experts, to decide if the alarm is real or fake. The user is unlikely to know the answer especially for difficult situations. Because the user has to clear the alarm, the false positives actually train the user to not trust the heuristics. It soon becomes automated for the user to clear the alarm without paying much attention.

The second problem with heuristics is that virus authors become aware of the heuristics and simply change the flow of the virus. A virus is not *required* to follow Gryaznov's six steps. There are other ways of achieving the same results, so as soon as a virus author knows that the AV products are using those rules as heuristics, they start using something else instead.

It turns out that malware classification is simply a hard problem. There are even theoretical proofs about this. The *Halting Problem*, for example, is a famous computer science proof. This proof shows that it is impossible to create a classifier that can tell if any given program with a particular input will get stuck and run forever or if it will halt (finish). A classifier can be written that can tell if *some programs* will halt, but the classifier cannot tell if every possible program will halt. This result was later extended to show that classifiers cannot categorize every possible program for any nontrivial characteristic. And whether or not a program is a virus is a nontrivial characteristic. What this means is it is impossible to write a classifier that can tell if, for every possible program, the tested program is a virus or not. There will never be a perfect antivirus [76].

This, however, is a theoretical result. So even though Gryaznov knew that heuristics could not possibly catch 100% of viruses, he expressed his belief that they could deal with 99% of the viruses, and then signatures could be used for the remaining 1% [122]. It turned out Gryaznov was wrong.

The problem with Gryaznov's reasoning is that he was treating computer viruses as if they were a *random* rather than an *engineered* process. If viruses were random, perhaps heuristics could catch 99%. But they are engineered by humans. And the virus authors appear to be *very motivated*. The virus author does not throw up their hands and say, "well too bad for me, someone came up with heuristics. I guess I am finished."

Truly, the very fact that the heuristics catch 99% of a current generation of malware simply gives the virus author a road map into what the form of the next virus should be. When the virus author sits down to write a new virus, will they write one like the 99% getting caught by heuristics or will they write it like the 1% that is not? Clearly, they will change to the form the heuristics miss, and the next generation of viruses will largely bypass the heuristic scanning. I have illustrated this process in Figure 7-21.

These examples so far have focused on older malware no longer in wide use. What about modern malware? Today, our defenses still use signatures and heuristics to defeat the trojans, worms, and other malware that still infests the Internet. These days, however, most malware is self-contained and runs on its own. For this reason, many signatures are now just hashes of the malicious data. And these signatures can be collected into cloud databases. For example, VirusTotal is an online service with a massive catalog of collected malware and hashes on them. Users can even submit files to VirusTotal for analysis. VirusTotal reports back to the user if their submitted file matches the hash of any malware in its database.

An example of modern heuristics is found in dealing with ransomware. Many of the modern anti-malware systems monitor for unusual encryption activity that is often associated with ransomware. This fairly narrow heuristic checking appears to work fairly well in practice.

Dynamic Analysis

Dynamic analysis is different from static analysis in that it actually runs the software, or part of the software, to determine if it is malicious. However, the dynamic analysis discussed here is still part of the identify and neutralize strategies rather than the mitigation strategies. How can you run malware and not have it do bad things?

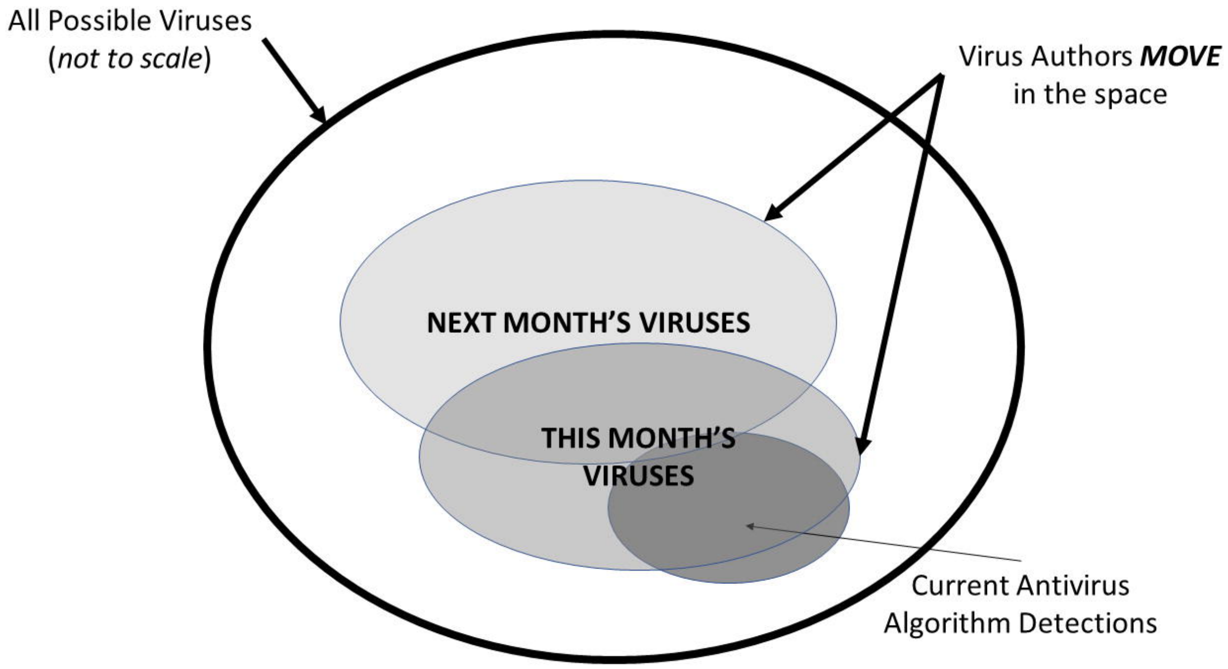


Figure 7-21 No matter how well defenses detect the malware in a given time frame (e.g., a month), it will not detect malware as well in the next time frame.

If the detection mechanism detects 99%, in the next time period, the malware authors will be creating their malware to be like the 1% that was not detected

The answer is to use some form of virtualization. Using a safe virtual environment, the potential malware can be executed or simulated and the results analyzed. The goal is to classify the software as benign or malicious based on the behavior.

The level of virtualization depends on the nature of the simulation. As described in the antivirus signature scanning, even antivirus in the 1990s had to simulate the execution of a virus in order to decrypt the virus payload. The antivirus program actually simulated a processor completely with memory and other processor components. It would read the instructions just like a real processor would and simulate the effects in its emulated processor environment. But the goal was not to classify. The goal was to decrypt for scanning.

Emulation can be done to actually detect behavior. This kind of emulation, for example, monitors any types of activities that can be suspicious. What counts as suspicious depends on the vendor and the security policy they adopt. But it often includes things like unusual data storage (or deletions!), changes to system settings, and network connections to certain undesirable locations. The emulator does not necessarily simulate the actual hard drives, settings, or network interfaces. Instead, it simply monitors for those types of operations being attempted. This kind of simulation is not meant to be completely real. It is usually an attempt to see some of the kinds of operations that program will attempt.

You may wonder about user input. After all, what makes a program a program is the branching in the instructions based on inputs and conditions. In some of the emulation programs I have examined personally, they attempt to execute *all branches*, but only for a relatively short period of time. Branches that look “interesting” can be followed for a longer period of time.

Some systems, however, want more realism than this. These systems opt for the actual use of virtual machines with full operating systems. These systems can insert a suspected piece of malware into the virtual machine and “detonate” it (run it). Usually, it is limited to run for a period of time (like a minute), and it will not receive any user inputs. But in many cases, this is enough to see if the software does something “harmful,” like delete files or do other nasty things. And because everything is in a fully isolated virtual machine, the malware is not able to escape into any real systems. The virtual machine is reset to its state before the detonation, and it is ready to test something else.

Obviously, most users do not have virtual machines launching on their system to check for malware. Usually, full virtual machines are used on some kind of centralized system, and the results are stored for distribution. Many anti-malware vendors will use these kinds of systems in their labs. It can be automated so that as samples are received from customers, they are automatically deployed to a VM for detonation. The results are processed and can be distributed to customers worldwide.

There are, however, some firewalls that can do this. I will discuss products like these more in the next chapter.

Story Time: A Third Form of Defense—Human Vigilance

Sometimes, an alert and security-conscious human spots suspicious attacks when no other defenses (dynamic, static, or otherwise) would have done so.

For example, in 2016, a human rights activist by the name of Ahmed Mansoor was targeted with a social engineering attack. He received a text message with a link that would supposedly provide secret information about torture happening in the United Arab Emirates. Fortunately for Mansoor, he was very wise and did not click the link. Instead, he sent it for analysis to the Citizen Lab group at the University of Toronto. They recognized it as coming from the NSO Group, a company that sells “lawful intercept” software (i.e., government-sanctioned spyware) to governments.

What Citizen Lab found is that the link was configured to deploy zero-day exploits that would have sliced through any security and installed the malicious software, now known as Pegasus. It is unlikely that any static or dynamic analysis would have caught it. Static analysis could not have caught it because it was unknown at the time. Dynamic analysis probably would not have caught it because it is exploiting a vulnerability in the iOS operating system directly. In any event, none of the defenses on the iPhone at the time would have stopped this attack from taking over Mansoor’s phone. Only his human thinking, correctly applied, prevented the infection [20, 50, 171].

Mitigation Strategies

Unlike the systems described in the previous section, mitigation strategies typically are designed to kick in after the system is already under attack. At the very least, the malware is already running on the potential victim system. These technologies try to inhibit the malware from doing damage.

In the early days of malware, *behavior blockers* were examples of this kind of strategy. A behavior blocker was installed on a system, and it would prevent all kinds of supposedly unwanted behavior. The behavior blocker would run in the background and detect when file access or operating system changes were requested. Any of these unwanted behaviors would bring up a warning to the user and, optionally, ask the user if it should be allowed. An example behavior blocker was the *Flushot* program.

Behavior blockers had a number of limitations and problems. In the first place, the malware was already running. Behavior blockers could not identify the malware nor remove it. All they could do was block some of its unwanted effects.

Another problem with early behavior blockers is that their alerts were false positives too often. Like heuristics, behavior blockers trained users to ignore them. Exceptions could be made permanent, but this also required more expertise than the average user has in the area of malware.

Still, the concept has persisted. McAfee (recently acquired and merged into Trellix), for example, offered a Host Intrusion Prevention System (HIPS) [45]. HIPS evolved into what is now known as McAfee Endpoint Security. This system offers access control rules for hosts including blocking writing to certain critical directories. It can also block certain types of web requests and operating system interactions. Endpoint security is a defensive system to prevent damage if something bad does get through other defenses [28].

Recover and Respond Strategies

Unfortunately, sometimes malware gets in and does damage. Some technologies are focused on recovery and response. One of the most immediate needs for this type of operation is simply *detection that an attack occurred at all*. Not every attack is like the I Love You worm that deleted most of a user’s files. Some attacks are data exfiltration, with no visible destruction at all. Other attackers are stealthy on purpose because they are stepping stones to later attacks.

Early technologies for this included “integrity checkers.” Integrity checking tools would take a checksum (which is like a hash) of every file on the system. At regular intervals, the files would be checked to see if they had been altered. The problem, of course, is that many files *should* be regularly altered. Documents are not static. And even operating system files change during normal system operation. Patching was nonexistent in the 1990s when integrity checkers appeared, but nowadays sensitive system files are updated regularly. How is an integrity checker to tell the difference between authorized and unauthorized changes?

Modern integrity checkers are far more nuanced and can be configured in accordance with policy. Kaspersky Endpoint Security includes, for example, a file integrity monitoring in various forms. This tool makes use of operating system features to monitor changes to the files in real time. Changes are not automatically blocked or flagged necessarily. Rather, each change is an event sent to the security center. Policy at the security center determines whether or not a single event, or a series of events, warrants a response. Nevertheless, Kaspersky’s product can compute file checksums for storage and later comparison.

Assuming that an attack is known, some systems have a built-in recovery system. These systems work in parallel with an automated backup service. If an attack is detected, the data that is damaged or destroyed can be recovered automatically from backup. Obviously, this approach is easy for a user to do themselves manually from their own backup, but the integration with the security system enables seamless cooperation. This type of technology can be used to mitigate ransomware attacks. Acronis is a security company with products that mitigate ransomware. Their ransomware protection can be configured to restore a compromised file from a wide range of locations including internal disk, external disk, remote server, or cloud storage [29].

More advanced systems for modern host security incorporates the ability to detect attacks as they are happening, or after they happen, and respond to prevent the attack from happening again in the future. Security of this form is sometimes called Endpoint Detection and Response, or EDR. EDR is defined by Gartner as

> The Endpoint Detection and Response Solutions (EDR) market is defined as solutions that record and store endpoint-system-level behaviors, use various data analytics techniques to detect suspicious system behavior, provide contextual information, block malicious activity, and provide remediation suggestions to restore affected systems [112].

EDR is stated by the same source to have four required functionalities:

1. Detect security incidents
2. Contain the incident at the endpoint
3. Investigate security incidents
4. Provide remediation guidance [112]

If you are wondering why the terminology has switched so strongly to “endpoint” rather than host, it is a matter of perspective. “Host” is a term that makes more sense to the user interacting with it. “Endpoint” is more appropriate for organizational security professionals that are trying to protect a network of resources. From their more centralized vantage point, hosts are endpoints in the network of data. EDR is still largely for commercial, rather than personal, computing.

Summary

Host systems are designed to protect themselves against accidents and malice largely by using isolation. By isolating running programs from each other, it is possible to enforce access controls and contain bad things from spreading through the whole system. Initially, operating systems (OSs) were the fundamental enforcer of isolation and access control. In recent years, many devices (especially mobile phones) have been increasing enforcement by specially secured areas, such as TrustZone’s Trusted Execution Environment. On the flip side, there are security controls above the operating system, such as middleware security controls and enforcement mechanisms within applications. A good example of application security isolation is the per-process protections in a modern web browser that isolate each tab from another.

Attacks on host systems are often based on either exploiting errors in the legitimate programs or getting malicious software to run and execute on the system through various means.

One of the most well-known and notorious errors that can make a program exploitable to an attacker is a buffer overflow. A buffer overflow happens when input data is larger than space allocated on the stack to hold it. As the data overflows, it overwrites return addresses that control the code that will be executed by the processor. In this way, an attacker can redirect the processor to code of their choosing in order to subvert the program and the system. If defensive measures prevent an attacker from just uploading their own instructions directly, they can use techniques like *return-to-libc* or *Return-Oriented*

Programming (ROP) to achieve their results. Techniques like Address Space Layout Randomization (ASLR) make these kinds of attacks much harder.

On the other hand, attackers can also take over machines by just getting their evil code inserted directly onto the host. In the past, viruses attached evil code to legitimate code so that running one would run the other. While viruses of that type are less common now, systems are always at risk from worms, rootkits, ransomware, and other malicious software. There are three basic approaches to protecting systems: identify and neutralized, mitigate, and recover and respond. Identifying and neutralizing malware (i.e., before the malware does any damage) is preferred. Two approaches for this type of protection include static analysis, such as signature checking, and dynamic analysis, like testing the malicious software in a virtual machine.

Further Reading

As always, I recommend Anderson's *Security Engineering* book. For the topics discussed in this chapter, I recommend Anderson's chapter on Access Controls [40, Chapter 6], Tamper Resistance [40, Chapter 18], and Side Channels [40, Chapter 19]. In the Tamper Resistance chapter, Anderson goes through a number of examples of how to "hack a cryptoprocessor." In these sections, Anderson is talking about hacking *hardware security modules*, which are often used as external devices for secure storage of cryptographic keys and similar operations. However, many of the concepts here could be applied to cracking a trusted execution environment. Anderson's chapter on Side Channels includes information about Spectre and Meltdown as well as other examples.

Bishop's *Computer Security* includes an entire chapter called "Confinement Problem" [60, Chapter 18]. This chapter includes a section on the *Isolation* problem as well as several sections on covert channels, which include side channels. The book also includes a chapter on Malware that goes into much more technical detail and formalism [60, Chapter 23].

One of the earliest papers on security in operating systems is "Multics Security Evaluation: Vulnerability Analysis" by Paul A. Karger and Roger R. Schell. Originally written in 1974, it was reprinted in a conference in 2002 [149]. This analysis from the 1970s introduced key and fundamental concepts such as a *reference monitor*, security levels ("rings"), hardware isolation, controls on memory segments, and many more. This is a great starting point for these concepts.

For those interested in modern trusted execution environments, Arm publishes a number of technical specifications related to the operation of TrustZone [42, 43, 288]. Other documentation can be found on Arm's website. Moreover, there are many research papers that discuss TrustZone's security [206]. There are also various attacks against TrustZone [70].

Another interesting concept is that of *provable correctness*. Formal proofs can be constructed on software to prove that it has no buffer overflows or other types of vulnerabilities. These proofs are very difficult to construct on large software, but seL4 is a type of operating system that has certain provable properties [151]. Interestingly, even though the *software* can be proven correct under certain circumstances, that does not protect it from hardware errors like *Spectre* and *Meltdown* [13].

Buffer overflow vulnerabilities have been around for a long time. A commonly cited paper is "Smashing the Stack for Fun and Profit" from 1996 [35]. It is still used as a starting-point paper when teaching computer science students about these kinds of attacks. Many papers have been written since promoting various protections or showing methods for bypassing defenses [82, 83, 90, 160, 220, 239]. Thomas Dullien wrote a paper showing that attackers' interactions with software are effectively a new program (their interactions create a series of state transitions, and those transitions are more or less a new computer program). Their goal is to get the software into a "weird" state, meaning a state that was not anticipated or tested by the authors [98].

Many of these papers are somewhat academic. On the more practical side, various books have been written for learning how to prevent or exploit vulnerabilities. A good survey of programming "sins" that lead to exploitable vulnerabilities is in *24 Deadly Sins of Software Security* [137]. On the flip side, *Exploiting Software* shows the reader how to find and exploit the vulnerabilities in someone else's software [135]. NIST has a special publication on the Secure Software Development Framework, a framework for developing code with minimal vulnerabilities and vulnerability robustness. This document describes various practices and approaches that support these goals [246]. NIST also has a special publication on patching strategies. Once a vulnerability is found, it is essential that it be removed (i.e., via patch) as soon as is feasible [245].

It is also worthwhile to read about how NIST incorporates host security controls into the overall security picture. NIST provides a special publication entitled "Security and Privacy Controls for Information Systems and Organizations," also known as SP 800-53. This document "provides a catalog of security and privacy controls for information systems and organizations to protect organizational operations and assets, individuals, other organizations, and the Nation from a diverse set of threats and risks, including hostile attacks, human errors, natural disasters, structural failures, foreign intelligence entities, and privacy risks." This catalog is broken down into 20 families of controls that are intended to be comprehensive, including everything from *Access Control* to *Physical and Environmental Protection*. Although there is no specific family of controls for host security, both the *System and Services Acquisition* controls and the*System and Communication Protection* controls include requirements applicable to hosts based on the isolation and mediation principles discussed in this chapter. Other families of controls, such as *Assessment, Authorization, and Monitoring*, *Incident Response*, *Maintenance*, and *System and Information Integrity*, include requirements that address malicious code identification, mitigation, and recovery [121]. It should be noted that NIST also produces another special publication called "Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations," or SP 800-171. This document contains a subset of the requirements of SP 800-53 and is often used for compliance purposes in American commercial entities that do not have access to classified government information [224]. However, because the controls are taken from SP 800-53, it is useful to refer back to that document for context and explanations even when SP 800-171 is used as the compliance target.

I did not discuss in this chapter the people that spend their time reverse engineering malware, which is often necessary to figure out what damage they have caused, how they work, and how to prevent them, but it can be a very difficult process unless you are deep in the art. A book like *Practical Malware Analysis* will show you how to analyze data down at the level of binary instructions to figure out what the attacker did [243].

From July 1989 to June 2014, *Virus Bulletin* published monthly in a magazine style. The publication was originally hard copy before switching to online-only in 2006. The *Virus Bulletin* archives at www.virusbulletin.com/virusbulletin/archive have all of these publications, including PDFs of the hardcopy magazines. These archives are a fascinating view into the early days of viruses and antivirus technologies.

Ransomware is probably one of the most destructive malware types at this time. NIST offers a comprehensive guide for enterprises to deal with the threat. The guide references other NIST documents but includes everything from risk assessment to containment in the event of a compromise [69].

References

11. TA13-032: Oracle Java multiple vulnerabilities. 2 2013. Last updated: 19 Oct 2016.

13. Crisis: Security vs performance. 04 2018.

20. Forensic methodology report: How to catch NSO group’s pegasus. 07 2021.

23. Application sandbox. 10 2022.

25. Intel SGX deprecation review. 01 2022.

27. What are computer viruses, 2022.

28. Mcafee endpoint security 10.6.0—threat prevention product guide—windows. Technical report, Trellix, 2023.

29. Acronis. 2018. A guide to ransomware and how acronis active protection can help.

31. Agarwal, A., S. O’Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom. 2022. Spook.js: Attacking chrome strict site isolation via speculative execution. In *43rd IEEE Symposium on Security and Privacy (S&P’22)*.

35. Aleph One (Unknown Author). 1996. Smashing the stack for fun and

profit. 7(49): 11. newsgroup article or web page (?).

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

42. Arm Limited, Cambridge, England. 2018. *Arm TrustZone Technology for the Armv8-M Architecture*, version 2.1 edition.

43. Arm Limited, Cambridge, England. 2022. *Introduction to the Armv8-M Architecture and Its Programmers Model*, version 1.0 edition

45. Arntz, P. 2013. What is host intrusion prevention system (HIPS) and how does it work?

46. Arntz, P. 2021. Ryuk ransomware develops worm-like capability.

50. Bazaliy, M., C. Neckar, G. Sinclair, and in7egral. 2016. Technical analysis of the pegasus exploits on IoS. Technical report, Lookout

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

62. Bossler, A., and T. Holt. 2009. On-line activities, guardianship, and malware infection: An examination of routine activities theory. *International Journal of Cyber Criminology (IJCC) ISSN* 3: 974–2891.

69. Cawthra, J., M. Ekstrom, L. Lusty, J. Sexton, and J. Sweetnam. 2020. Data integrity: Detecting and responding to ransomware and other destructive events. Special Publication (NIST SP) 1800-26, National Institute of Standards and Technology, Gaithersburg.

70. Cerdeira, D., N. Santos, P. Fonseca, and S. Pinto. 2020. SoK: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, 1416–1432.

72. Chaumette, S., O. Ly, and R. Tabary. 2011. Automated extraction of polymorphic virus signatures using abstract interpretation. In *2011 5th International Conference on Network and System Security*, 41–48. IEEE.

75. Cluley, G. 2013. The dying art of computer viruses. *Virus Bulletin* 2. www.virusbulletin.com/virusbulletin/2013/08/dying-art-computer-viruses.

76. Cohen, F. 1987. Computer viruses: Theory and experiments. *Computers and Security* 6(1): 22–35. [MathSciNet][Crossref]

82. Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P.

Wagle, Q. Zhang, and H. Hinton. 1998. StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks. In *7th USENIX Security Symposium (USENIX Security 98)*, San Antonio. USENIX Association.

83. Cowan, C., P. Wagle, C. Pu, S. Beattie, and J. Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX'00)*, vol. 2, 119–129. IEEE.

84. Craciun, V.C., A. Mogage, and E. Simion. 2019. Trends in design of ransomware viruses. In *Innovative Security Solutions for Information Technology and Communications*, ed. J.-L. Lanet and C. Toma, 259–272. Springer International Publishing.
[Crossref]

88. Cushing, T. 2017. How minecraft led to the Mirai botnet. *TechDirt*.

90. Dang, T.H., P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (ASIA CCS'15), New York, 555–566. Association for Computing Machinery.

98. Dullien, T. 2020. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* 8(2): 391–403.
[Crossref]

103. Federal Bureau of Investigation. 2018. The Morris worm: 30 years since first major attack on the internet.

112. Gartner. Endpoint detection and response (EDR) solutions reviews and ratings.

118. Griffiths, J. 2020. 'I love you': How a badly-coded computer virus caused billions in damage and exposed vulnerabilities which remain 20 years on. *CNN Business*.

121. Group, J.T.F.T.I.I.W. 2020. Security and privacy controls for federal information systems and organizations. Special Publication (NIST SP) 800-53r5, National Institute of Standards and Technology, Gaithersburg.

122. Gryaznov, D. 1999. Scanners of the Year 2000: Heuristics. In *Proceedings of the 5th International Virus Bulletin*.

126. Halderman, J.A., and E.W. Felten. 2006. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th Conference on USENIX Security*

Symposium – Volume 15 (USENIX-SS’06). USENIX Association.

128. Hassold, C. 2022. The victimology of ransomware: 4,200 ransomware victims and counting.

135. Hoglund, G., and G. McGraw. 2004. *Exploiting Software*. Addison-Wesley Professional.

137. Howard, M., D. LeBlanc, and J. Viega. 2009. *24 Deadly Sins of Software Security*. McGraw-Hill.

142. ITL, C. 2019. Binary hardening in IoT products.

149. Karger, P., and R. Schell. 2002. Multics security evaluation: Vulnerability analysis. In *18th Annual Computer Security Applications Conference, 2002. Proceedings*, 127–146.

151. Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. SeL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP’09)*, New York, 207–220. Association for Computing Machinery.

153. Kocher, P., J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P’19)*.

154. Kominski, I. 1989. Computer use in the United States: 1989. Technical Report 171, U.S. Department of Commerce, Bureau of the Census, Washington, DC.

156. Kshetri, N., and J. Voas. 2017. Do crypto-currencies fuel ransomware? *IT Professional* 19(5): 11–15.
[Crossref]

160. Larochelle, D., and D. Evans. 2001. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 2001)*, Washington, DC. USENIX Association.

162. Lhee, K.-S. and S.J. Chapin. 2003. Buffer overflow and format string overflow vulnerabilities. *Software Practice and Experience* 33(5): 423–460.
[Crossref]

163. Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018.

Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

171. Marczak, B., and J. Scott-Railton. 2016. The million dollar dissident. Technical Report 78, University of Toronto.

173. Marinho, T. 2018. Ransomware encryption techniques. *Medium*.

174. Markoff, J. 1988. Author of computer 'Virus' is son of N.S.A. expert on data security. *The New York Times*.

183. Microsoft Corporation. 2023. Memory integrity and VBS enablement.

185. Microsoft Corporation. 2023. Virtualization-based security (VBS).

191. Muttik, I. 1999. Macro viruses—Part 1. *Virus Bulletin* 13–14. www.virusbulletin.com/uploads/pdf/magazine/1999/199909.pdf.

196. O'Kane, P., S. Sezer, and D. Carlin. 2018a. Evolution of ransomware. *IET Networks* 7(5): 321–327.
[Crossref]

198. Oracle. 2013. Java applet and web start code signing.

201. Palmer, D. 2019. Mydoom: The 15-year-old malware that's still being used in phishing attacks in 2019.

202. Palmer, D. 2022. Android malware: A million people downloaded these malicious apps before they were finally removed from Google play.

206. Pinto, S., and N. Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys* 51(6): 1–36.
[Crossref]

220. Roemer, R., E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* 15(1): 1–34.
[Crossref]

221. Root, E. 2021. Spook.js, a scary bedtime story.

224. Ross, R., V. Pillitteri, K. Dempsey, M. Riddle, and G. Guissanie. 2020. Protecting controlled unclassified information in nonfederal systems and organizations. Special Publication (NIST SP) 800-171r2, National Institute of Standards and Technology, Gaithersburg.

226. Russinovich, M. 2005. Inside Sony's rootkit. *Virus Bulletin* 11–14. www.virusbulletin.com/uploads/pdf/magazine/2005/200512.pdf.

238. Seltzer, L. 2020. 2020 ransomware attacks still mostly through unsecured

RDP.

239. Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, New York, 298–307. Association for Computing Machinery.

243. Sikorski, M., and A. Honig. 2012. *Practical Malware Analysis*. San Francisco: No Starch Press.

245. Souppaya, M., and K. Scarfone. 2022. Guide to enterprise patch management planning. Special Publication (NIST SP) 800-40r4, National Institute of Standards and Technology, Gaithersburg.

246. Souppaya, M., K. Scarfone, and D. Dodson. 2022. Secure software development framework (SSDF) version 1.1. Special Publication (NIST SP) 800-218, National Institute of Standards and Technology, Gaithersburg.

247. Spafford, E.H. 1989. The internet worm program: An analysis. *SIGCOMM Computer Communications Review* 19(1): 17–57. [Crossref]

259. Szappanos, G. 2004. Virus analysis 2: We're all doomed. *Virus Bulletin* 9–13. www.virusbulletin.com/uploads/pdf/magazine/2004/200403.pdf.

263. Toulas, B. 2022. New Intel chips won't play blu-ray disks due to SGX deprecation.

282. Williams, C. 2016. Double ko! Capcom's street fighter v installs hidden rootkit on PCs. *The Register*.

288. Yiu, J. 2017. Software development in ARMv8-M architecture. Presented at Embedded World 2017.

Footnotes

1 There is another, related security design principle known as the principle of complete mediation. This principle states that a protected resource must have *all* accesses mediated.

2 Many programs start with a *main* function, or subroutine, that starts the program and calls all other subroutines.

3 As I explained earlier, in a real system there are no names like **C** or **F**. Instead, the data would be stored in the stack by address, as the stack is just a set of memory addresses. I also mention in passing that this kind of data might not be stored on the stack at all but could be stored in registers instead.

4 Actually, text like this almost always includes a special invisible character at the end called the *null terminating character*. This is used to mark the end of the string in the buffer. This means that this buffer can actually only hold 79 characters.

5 Coincidentally, BSD is also related to modern versions of the MacOS system.

6 However, the first uninstaller they released was also problematic and, according to some accounts, made things worse [126]!

7 The lesson is, do not mess with Brian Krebs.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_8

8. Classical Network Security Technology

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
Building on the previous chapter, I will now examine attacks and defenses for networks and how they have evolved over time as network architectures and attack vectors have evolved. Because this chapter is about *classical* network security, I will focus on *perimeter* defenses built around firewalls, proxies, and other similar devices. Intrusion detection, including defensive deception, provides a second layer of security by identifying intruders if they get past the outer walls. I also discuss how attackers bypass network security systems.

Key Concepts

1. In classical (perimeter) network security, traffic entering a network from outside has higher risk than traffic originating inside a network.
2. Network traffic can be analyzed, filtered, and shaped as part of security policy enforcement.
3. Firewalls and proxies provide perimeter

enforcement; IDS and IPS detect threats that get past them.
4. Defensive deception is designed to fool *the adversary*, waste time, track movements, and prevent misuse of real resources.

Common Pitfalls and Misunderstandings

1. Many of these concepts were created in the context of trusted networks within organizations, so they may not be implemented the same way in modern networks.
2. Throwing a firewall on a network will make the network “secure.”

Useful Vocabulary

- **Firewall**: Network device that controls, monitors, or restricts traffic between two network partitions
- **LAN**: Local Area Network. A defined network, perhaps within the premises of an organization, that allows interactions between more-or-less trusted machines performing similar functions
- **WAN**: Wide Area Network. A network outside a LAN that allows interconnections between other unrelated networks
- **DNS**: Domain name system. A system for translating human-readable domain names (e.g., google.com) to the IP addresses necessary for network communication
- **VPN**: Virtual Private Network. A bridge between two remote networks (e.g., “site-to-site VPN”) or between a machine and a remote network (e.g., “remote access VPN”)

- **IDS**: Intrusion Detection System. A system for detecting the presence of threats or anomalies on a network or host
- **IPS**: Intrusion Prevention System. A system for preventing threats from infiltrating a network or machine
- **DMZ**: Demilitarized zone. A network segment containing high-risk systems, such as public-facing servers, which helps isolate intrusions from other more sensitive network segments
- **APT**: Advanced persistent threat. A type of attacker known for stealthy, patient, and sophisticated operations to infiltrate even well-defended networks
- **DLP**: Data loss prevention. A system that blocks sensitive data from being exfiltrated

---

In this chapter, I will cover what I call “classical” network security. For at least two decades, and perhaps closer to three, network security has been defined largely by *perimeter security*. This security model presumed a relatively static computer network that exists in a primarily small geographic area. Hopefully, it will be obvious why this model for network security needs changes. The modern computing world is all about cloud computing, services spread across many networks, and increased remote access (e.g., from more people working from home).

Nevertheless, the classical network security models are still in use in many environments, and IT security professionals will spend a significant amount of time worrying about these systems in their organizations.[1] Moreover, the principles behind these models, as well as the lessons learned from developing them, are foundational. Understanding the past is key to understanding the present.

A good starting point is the security issues we have inherited from relying on technology designed in a presecurity era.

Legacy Networking Security Implications

The core concepts, protocols, and technologies of the modern Internet were developed in the 1970s and 1980s (for a review of these concepts, please refer to Appendix C). It is phenomenal that in an era when a mobile phone is obsoleted in a few years, the underlying network technologies have managed to last for so long.

Unfortunately, however, it also means that the majority of our Internet was envisioned and designed before there were so many of our modern security issues. It was designed to work on an open network between a relatively small number of nodes operated by largely trustworthy parties. This means that very little computer security considerations were incorporated into this architecture. Much of the security we have in computer networks now has been added on in an attempt to make an insecure system secure.

Of course, as discussed at length in previous chapters, security does not exist as an intrinsic virtue. Rather, security is about defining some kind of policy and enforcing it. Identifying the policy is essential for even discussing what network security should be.

In the early 1990s, some people felt that there was no need for network security at all. From their point of view, the network is not a protected resource. *Hosts* are the protected resource, and as long as hosts are adequately protected (e.g., using the principles and technologies discussed in Chapter 7), network security is unnecessary. In fact, to the extent that trying to secure a network led to a lax security policy on hosts, or a false sense of security for an organization that did not take host security seriously, “network security” could actually lead to a decrease in overall security posture [53].[2]

But with time and experience, it became clear that network security was important for many reasons.

First, computers on a network work together, and a compromise in one almost always leads to a compromise in another. This means that if you have ten machines and nine are “secure” (whatever that means) and one of them is insecure, it is probably true that all ten are insecure. By having some defenses in place before an attacker ever gets to a host (and maybe cannot even reach a host), the *attack surface is reduced*. Attack surface refers to how many different components of a system an attacker can try to compromise. Minimizing the attack surface is a principle of good security design. If an attacker can only attack one node on a network of ten nodes, then security can be emphasized and prioritized for the attackable node.

Second, networks enable distributed systems that are not confined to a single host. Without network security, it is difficult to enforce security policies on the cooperative operations.

Third, network security enables some specialization and centralization of security. It has been proven empirically that complexity leads to vulnerabilities, and vulnerabilities lead to security incidents. One of the real challenges of a general-purpose computer is that *it is general purpose*. All of the infinite possibilities offered by such a machine can potentially be turned against the system by an attacker. Network security enables the use of devices specifically tailored to security. These devices have reduced complexity[3] and can be designed to be very resilient to attacks.

Fourth, network security provides additional options for defense in depth, which has been discussed previously. Recall that ideally the failure of a single security mechanism should not result in a compromise. Network security is not meant to replace host security but add another layer of protection on top of it.

Fifth and finally, network security can provide early warning, mitigation, and forensic information. Even if an attacker tries to break into a system and is unsuccessful in doing so, system operations generally should be aware that such an attack was attempted. Attempting to bypass a network tripwire is an earlier alert than a host detection. Networks can track and log a significant amount of information that enables operators to figure out what happened if an attack was successful and even to mitigate damage in an attack happening in real time.

So what are common goals or policies for classical network security? Obviously, precise goals vary by organization, but there are a few general policy guidelines that tend to be common such as

1. Minimize the attack surface by protecting the servers
2. Identify and block inbound and outbound threats
3. Partition network space based on trust, policy, and/or risk
4. Detect and report on unsuccessful attacks

Servers and Port Scans

In terms of defending a system from a network intruder (e.g., an intruder that is connecting remotely over the Internet), servers are generally the most common way an attacker gains access to the system. Servers are designed to accept incoming network connections and respond to them. Although almost all servers will have access controls, an attacker can try to bypass them through any number of means. In fact, all of the issues discussed previously in Chapter 7 apply. An attacker, for example, may try to compromise a remote machine using a buffer overflow attack. In order to prevent an attacker from even trying such an attack, a lot of network security goes into preventing attackers from even accessing unauthorized systems. Effectively, network security attempts to introduce additional, network-based access controls.

Another important point about servers is that *they are processes (running programs), not machines*! Most consumers would not think of their personal computers as “a server,” but many programs that run on a consumer’s laptop can launch server processes. These can include certain video games, music apps, and even operating system services when in certain modes. Remote access systems for Windows and MacOS often require the system to operate a server. This means that *any computer on a network can be a potential server*.

What makes a process a server is if it is *listening on a port*. As discussed in Appendix C, a server has to *reserve a TCP or UDP port*. Ports are used to divide up network traffic to the different running programs, similar to a mailbox in an office. The IP address gets network traffic to the host, but the TCP port gets the correct network data to the right program. Servers reserve a port and then “listen” on that port, meaning that if traffic arrives with that port number, they will respond to it. When attacking a computer, attackers want to find these server ports.

In order to find these listening ports, attackers often perform what is called a *port scan*. There are only 65,536 port numbers on a machine total. It is a fairly quick process to send TCP packets to a machine *at each port number*. If there is no server listening for that port number, nothing will happen. But if there is a server, it will send back a response. The attacker does not need, at this stage, to even process the response. The goal was to perform reconnaissance and determine if there was a server running on that port at all. Network security should, ideally, prevent this from happening. But if somehow an attacker does learn of a nonpublic server's existence, network security should also prevent access.

In the remainder of this chapter, I will walk through some of the classical network security technologies used to protect network servers and systems. As you will see, much of this security is about controlling and protecting access to servers and their ports.

Firewalls

One of the primary components for classical network security is the *firewall*. A firewall is any network device that sits between two network partitions and applies security policies to the traffic that travels between the two. The security policies, at a minimum, must decide if traffic is allowed to pass through the firewall from one network to the other or if it should be blocked. Firewalls developed out of gateway routers, and even today gateways are the most common location of a firewall.

It was somewhat natural for gateway nodes to take on defensive capabilities. Although somewhat overly simplistic, networks are typically thought of as divided between local LAN and remote WAN networks, or "inside" and "outside." Typically, a LAN can only connect to the Internet through a gateway. A gateway may be as simple as a modem provided to the user by the local telecommunications company or a powerful and expensive system engineered to provide vast amounts of data with speed and resiliency. But no matter whatever other features it has, the gateway is the “choke point” for data going out and coming in (Figure 8-1).

Recall that the gateway router serves a primary purpose of knowing how to route packets out of the network from local, inside nodes, or how to route packets into the network from outside, remote nodes. It is called a “gateway” router because all the devices on the local network must send their data through the gateway to reach other networks.

For this reason, gateways emerged as a natural defensive position in the world of network security. Not only are they choke points, but they also were seen as a natural boundary in trust and security policy. The gateway separates “us” and “them,” and traffic that might be entirely acceptable for “them” may not be the least bit appropriate for “us.”

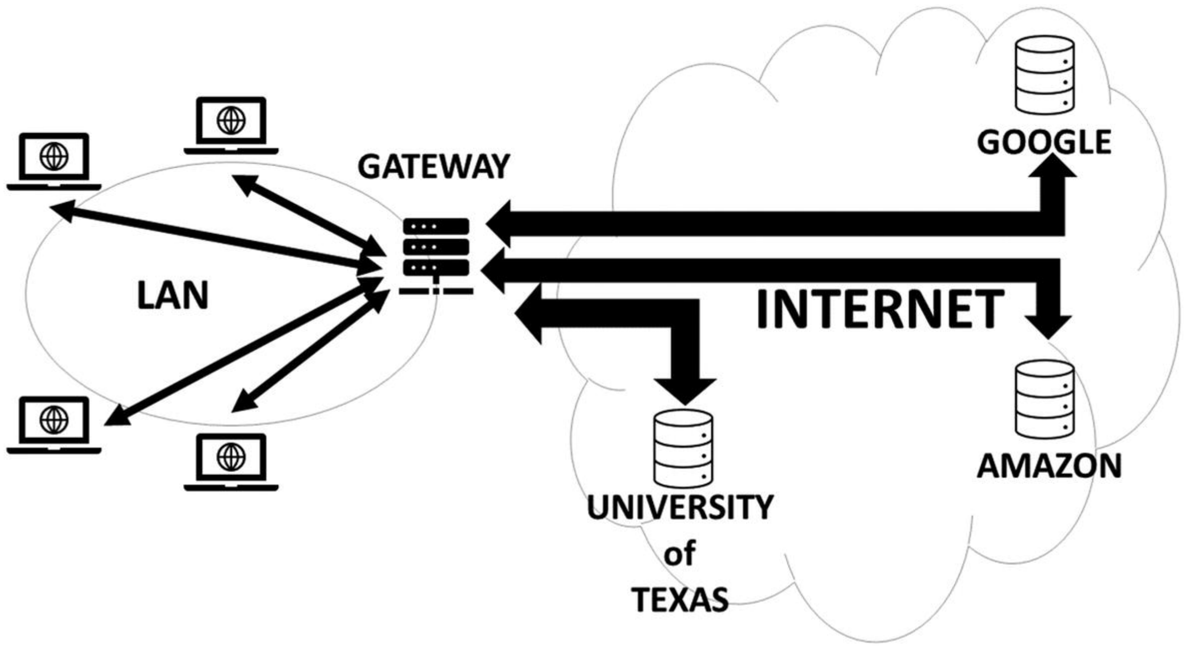


Figure 8-1 A gateway device connects the LAN to the Internet. No data gets into or out of the network without going through it

Story Time: The Great Firewall of China

Unfortunately, firewalls are not just used for defensive purposes. The Chinese government put strict digital controls on information going in and out of the country over the Internet. The government operates digital border controls that have, collectively, been referred to as the Great Firewall of China. This filtering system is used to block Chinese citizens from accessing certain websites and from posting certain content.

Although the systems used to control information in and out were put in place in 2003, they have accelerated under the leadership of Xi Jinping, who came to power in 2012. Under Xi, China has been willing to "cut off their nose to spite their own face." For example, the government was so concerned about blocking VPNs, which can bypass firewall inspections, that the government simply blocked VPNs. This move was made despite the fact that banks and businesses relied on VPNs for security connections.

Of course, a firewall can only control and limit connections. It has only a limited ability to detect "unacceptable" content using, for example, keywords (e.g., keywords related to Tiananmen Square). So, the filtering performed by the firewall is strengthened by an army of online censors and monitoring agents estimated in 2013 to number *approximately two million people* [100, 111, 286].

In the days before cloud computing, the vast majority of an organization's computing resources would be run by them. In modern terminology, all, or most, of their equipment was "on premise" (or "on prem" for short). An organization's servers, databases, and computing resources were organized into one or more LANs, each administered by the organization's IT department or functional equivalent. Web servers, email servers, and other resources were often controlled and configured internally, especially for large organizations.

Because LANs were created, controlled, and administered by an organization, a LAN represented a "cyber space" on the Internet for the organization itself—a kind of virtual territory that mirrored the organization's geographic territory. Although this description is somewhat of an oversimplification, and some organizations did outsource resources or rely on third parties, there were many organizations that operated in this manner, and much of computer security was built up around this conceptual architecture.

Before gateways evolved into firewalls, they included some very primitive security configuration. Technical documents in 1987 discuss having gateways do filtration on IP addresses that were "misbehaving" [64]. After the effects of the Morris worm, there was an acceleration to build in more explicitly security technology.

What emerged in the next few years were firewalls, and amazingly advanced firewalls all things considered. In fact, the core firewall technology still used today is not all that different conceptually from firewalls from 1994. One core concept from the early 1990s was packet filtration, which filters packets based on characteristics of each packet. Another core concept was that of a *stateful packet filter* which evaluates packets based on previously seen packets as well. Yet another important development was the *application layer gateway* (ALG).

Generally, modern firewalls use all of these concepts in one form or another, although the ALG has more or less merged into an advanced form of stateful firewall inspection. Modern firewalls can perform application-layer processing as part of a stateful packet tracking and analysis.

In the next sections, I will walk through some basics of both concepts historically and in modern systems.

Per-Packet Filtration

The most basic concept of a packet filter is to decide to allow or block the packet based on the metadata in the headers at layers 3 (IP) and 4 (TCP or UDP). Recall that layer 3 includes the source and destination IP addresses and layer 4 includes ports.

As mentioned in the preceding section, before the advent of firewalls, gateways could be configured to block incoming IP packets based on their source IP address. The idea was that if a machine outside the network was "misbehaving," it could be identified by its IP address and blocked from sending data into the local network.

IP addresses can be filtered for both inbound and outbound traffic. For traffic coming into the network, IP addresses associated with bad hosts and bad networks can be blocked. Although IP addresses are not perfectly associated with a geographic area, there is a reasonably good connection between an IP address and where it originates in the world. Some firewalls, from the 1990s until today, can attempt to block a geographic area of the world (e.g., eastern Europe or China) by this kind of geographic matching. It is very easy for an attacker to bypass this kind of check, however, using a VPN as I will discuss later in the chapter.

Filtering IP addresses can be used for one other very important purpose. Typically, an organization will have a very small number of machines that should be accessible from outside the LAN. For example, before the migration of most resources to the cloud, many organizations had their own on-prem web server machines. These machines *should* be accessible from outside the organization. But most machines *should not*. As I explained in the "Legacy

Networking Security Implications” section, just about any machine can have a server process running on it. But the vast majority of those machines were not meant to be accessible to the outside world. Using IP filtering, all inbound traffic from outside the firewall can be blocked to any machine that is not meant to provide an outside-facing server.

Outbound traffic can also be blocked. IP addresses can be associated with entertainment (which, at some organizations, is not an approved use of company resources) or illegal activities, such as copying music or movies, or even dangerous websites that a user might have been tricked into visiting. So long as the IP address can be associated with a malicious actor, it can be blocked. This type of filtering also has limitations. Generally, filtering on IP addresses alone can be too broad and not sufficiently granular.

As firewall technologies emerged, they were designed to expand on this basic filtration of IP addresses and also filter on ports. Packet filtering firewalls served as enhanced security gateways. As a gateway, all traffic from “outside” had to pass through the firewall to reach the “inside,” just like any other gateway. But the firewall would examine the packet and extract the source and destination IP address from the IP header and then the source and destination port information from the TCL header (encapsulated within the IP header).

The firewall could then filter on both IP address and port information. This enabled much better management of, and finer-grained policies for, inbound and outbound traffic. Because servers are associated with an address and a port, it is much easier to configure firewall rules using both pieces of information. The packet filter can also do some limited packet-type processing but *not* from layer 7. In other words, by looking at layer-4 data, the firewall can tell whether or not the layer-4 protocol is UDP or TCP.[4] But a packet filter cannot use layer-4 information to determine the application (layer-7) data, such as whether or not the packet is web traffic or email traffic. I will return to layer-7 processing in a moment. For now, it is sufficient to say that packet filtering firewalls can examine *five pieces of data*: source IP, destination IP, source port, destination port, and the layer-4 type. The layer-4 type is referred to in firewall documentation as “protocol,” but that could be confusing in this book. For clarity, I will focus on the first four pieces of information (addresses and ports) with the understanding that a firewall can have different rules for TCP packets and UDP packets. Most of our discussion will revolve around TCP packets anyway.

Using this data extracted from a packet, firewalls can apply a set of rules to the packet. When a packet arrives, the five pieces of metadata are extracted and compared to each rule one at a time and in order. If a given rule applies, the packet is either permitted to pass or is dropped and no other rules are checked. If a rule does not apply, the extracted metadata is compared to the next rule in the list. Firewalls are almost always configured with what is known as a *default deny rule*. This means that if a packet is compared to all rules and none of them match, the packet is denied by default. This is an example of a *fail-safe* rule. If the system fails (i.e., if there should have been a rule for the packet but the firewall is misconfigured or not completely configured), it will fail safely by blocking incoming data.

Using a firewall’s more granular filtration, servers can be more carefully protected. Recall that ports 80 and 443 are used by web servers for unencrypted and encrypted traffic, respectively. By filtering on both IP address and port, firewalls could restrict inbound web traffic to not only a single machine but to the specific ports of the machine.

This means that if another server were accidentally (or maliciously!) launched on the web server, it would not be reachable from outside the LAN.

This is a crucial concept. Because a server is a process, there can be many servers running on a single server machine. Generally, this is not good security practice. If two servers are running on the same server machine, a compromise in one leads to a compromise of both. Said another way, if an attacker can break into the machine through a bug or vulnerability in one server, they will most likely be able to compromise the other. For machines that are expected to be public-facing, and bear the brunt of attacks, the goal is to have as few vulnerabilities as possible and limit damage as much as possible.

By using packet filtering rules to block data coming from the outside to the web server, it is possible to enforce the limitation that only one server is accessible to outsiders. If an administrator accidentally runs a program that is or launches a server, the rogue server will not be available to outsiders.

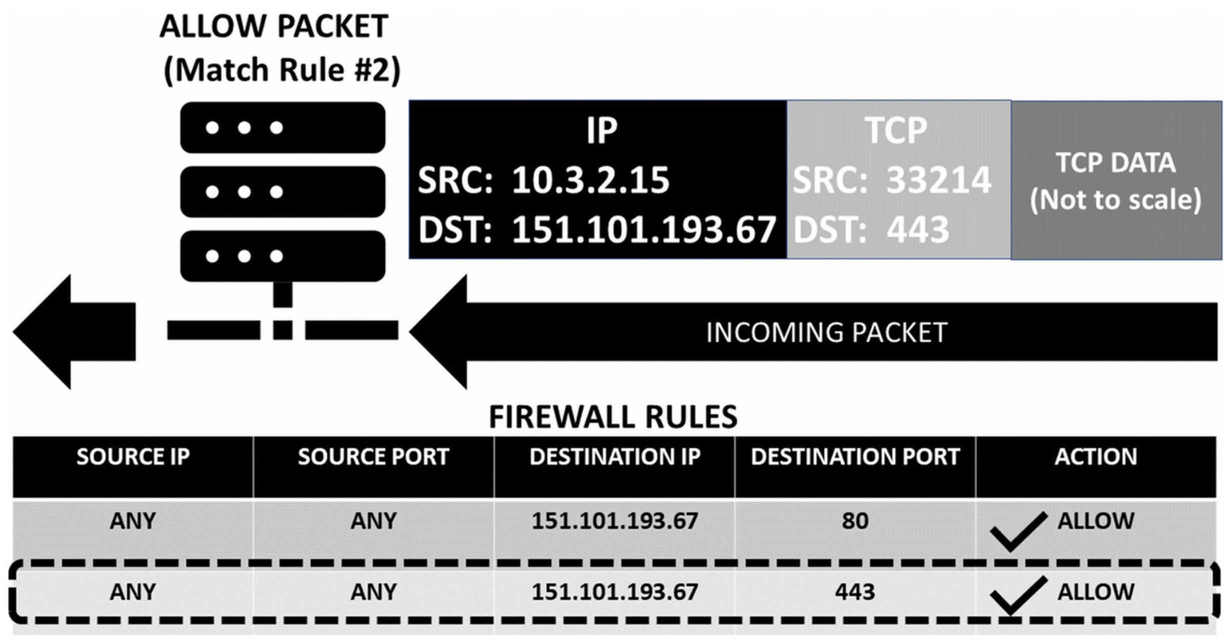
Example: Only Web Access for Web Servers

To visualize this, imagine a person at home wishes to check the news at cnn.com. The user opens their browser and types cnn.com into the URL bar and hits enter. This initiates the network protocol stack operations described earlier in this chapter. A packet is constructed with the web data as the application data. The most common protocol for web communications is called the *HyperText Transfer Protocol*, or HTTP. I will discuss HTTP and web traffic in greater detail in Chapter 9. For now, all you need to know is that HTTP data is layer-7 application data that is used by web browsers to communicate with Internet servers in order to, for example, view web pages.

When the HTTP data is sent by the browser, the HTTP data is encapsulated by TCP, which is encapsulated by IP, which is encapsulated by the MAC packet. This packet is transmitted through the Internet until it reaches the LAN on which the cnn.com web server resides. Before the packet reaches the server, it passes through a firewall that serves as the LAN’s gateway. Remember, there is no way into the LAN except through a gateway, and the gateway is the firewall.

Imagine that CNN’s firewall has a policy that explicitly permits web traffic to the IP address of the server machine hosting the web server so long as the port is 80 or 443. The incoming packet from the user’s browser has a destination IP address of the web server and a destination TCP port of either 80 or 443 depending on whether the traffic is encrypted or not. When the packet arrives, the source and destination IP addresses and the source and destination ports are examined. The firewall begins comparing this data to the firewall rules. When it reaches the rule that permits incoming traffic with a destination IP address of the web server and a destination port of 80 or 443, the packet is accepted. An illustration of the firewall accepting a packet destined to port 443 is shown in Figure 8-2.

The packet, once permitted, moves into the LAN normally and is transmitted to the web server. The CNN web server responds with the data for the CNN web page in various HTTP response packets. These packets start their return journey and must also go through the firewall where they will be examined themselves.



SOURCE IP	SOURCE PORT	DESTINATION IP	DESTINATION PORT	ACTION
ANY	ANY	151.101.193.67	80	✓ ALLOW
ANY	ANY	151.101.193.67	443	✓ ALLOW

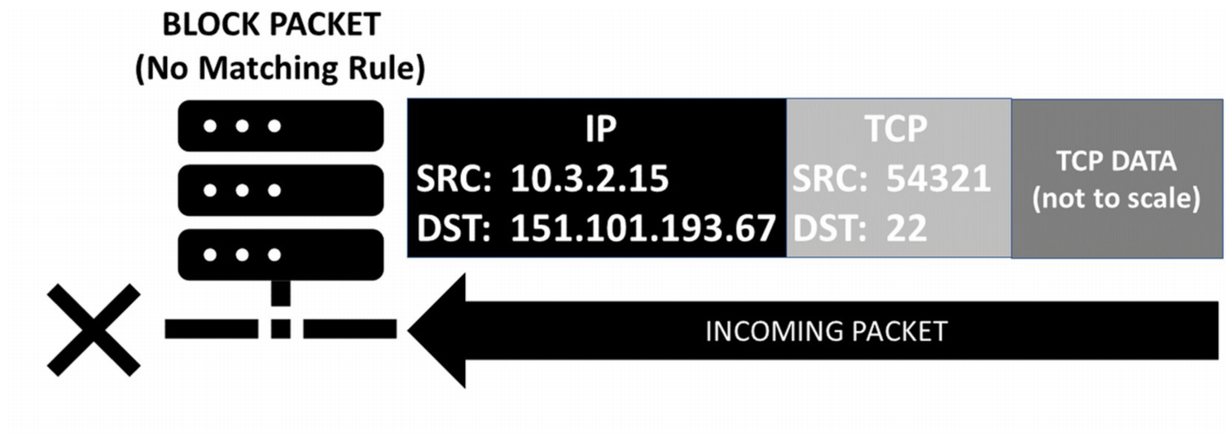
Figure 8-2 The firewall accepts the incoming packet because the destination IP and the destination port match a firewall rule with the action “allow.” The source IP and source port are not checked for this particular rule

On the other hand, imagine an attacker launching a port scan against CNN’s web server. Remember that there is no real data in a port scan. There is just a TCP packet that tries to establish a connection to see if there is a running server on a port. So the attacker creates a series of TCP packets starting at port 0[5] and continuing up to 65535. Each packet is transmitted to CNN.com.

As with the packet from the user’s browser, each of these packets of the port scan is intercepted at the gateway firewall and inspected. Extracting the source IP, destination IP, source port, and destination port, the firewall examined the packet against each of the rules. There is no matching rule for any of these TCP packets except the ones for ports 80 and 443. All other packets of the scan are dropped as there is no rule to permit their entry into the system as illustrated in Figure 8-3.

Stateful Packet Filtration

What I have just described is basic packet filtering. Basic packet filtering is really important and still widely used. However, firewalls got a little smarter and developed a new scanning that is considered to be *stateful*. The idea of a stateful firewall is that it could keep track of which packets had been sent before to know whether or not a given packet made sense in context. For example, it might be fine for a web server to send a response *but not if there was no preceding request*! If a web server was transmitting data outbound with no matching incoming data, that was generally a sign that an attacker had compromised the machine and was now exfiltrating data. Using stateful rules, firewalls can indicate that servers should only be responding to data and not initiating connections.



SOURCE IP	SOURCE PORT	DESTINATION IP	DESTINATION PORT	ACTION
ANY	ANY	151.101.193.67	80	✓ ALLOW
ANY	ANY	151.101.193.67	443	✓ ALLOW

Figure 8-3 The firewall blocks the incoming packet if the data in the IP and TCP headers do not match any rule. When no rule matches, the default "deny" (or block) rule is applied

Although stateful firewalls could be useful on many kinds of packets, they were exceptionally helpful on TCP packets. As explained earlier, TCP creates state in the form of a session. What I did not explain at the time was that

TCP communications actually have to start with a special set of control packets from TCP called a handshake. Both client and server have to exchange setup information in this handshake before any other application data (e.g., the HTTP data of this chapter's example) can be transmitted. Stateful firewalls can keep track of these kinds of exchanges to make sure that data is not being sent without a preceding handshake and that there is not a huge influx of handshake packets.

SYN Flood Attacks

Although the details of the TCP handshake are not described here, it is sufficient to know that the first packet sent by a client trying to open a TCP connection is called a SYN packet (pronounced "sin"). What is unfortunate is that attackers can use them to overload a system's resources and make it unusable by overloading the victim system with a *SYN flood*. This is an example of a DOS attack, or denial-of-service attack.

A SYN flood attack works by taking advantage of the fact that TCP creates sessions and that resources have to be allocated for those sessions. A server has to be able to differentiate between different incoming connections, even if they are from the same machine. Suppose, for example, that you had two browsers open to cnn.com. Perhaps one page has the sports news open and one page has world news. Both of the corresponding HTTP requests are going to come from the same IP address. How can cnn.com keep track of which data goes back to which request?

It turns out that network nodes keep track of source IP, destination IP, source port, and destination port for the TCP protocol in a manner that is somewhat similar to a firewall. But whereas a firewall uses it for inspection and policy, the TCP protocol just uses it to keep track of TCP *connections*. As explained earlier in the chapter, a TCP server has to reserve a specific port as a rendezvous point for clients. But each client program (e.g., browser, video game, etc.) when making the outbound connection to the server is assigned by the client device a random, *unused* TCP port for the connection's source port. This means that the server can differentiate between an incoming packet with a given source IP address and source port 54321 and an incoming packet with the same IP address but a different source port 43215. Using this information, the client and server associate packets for the same source and destination information together in a *logical* connection. By logical, I mean there is nothing physical about this so-called "connection." Rather, the two computers associate the related packets together and call it a connection. This logical connection only exists until a TCP operation terminates, or closes, the association. For the server, a new connection begins to be tracked when the SYN packet, which is the first packet of the session, arrives. Technically, the connection is "half-open" because TCP requires a response from the client to a separate SYN sent by the server in order to finalize the connection. The attacker will not respond and instead leave the connection half-open. Because of this, SYN flood attacks are sometimes called half-open attacks.

The problem is an attacker can transmit thousands of SYN packets, each with a different source port. The attacker has no interest in actually creating the true TCP connection or sending any data. Its sole purpose is simply to get the server to create those sessions, chewing up allocated resources. Note that the attack does not eliminate a port number that another legitimate client can use. The legitimate client has its own IP address, and if the attacker uses port 12345, it does not prevent the legitimate client, with a separate IP address, from also using port 12345. So a SYN flood does not "use up" port numbers.

But a server has a maximum number of connections it can have simultaneously open at one time. The SYN flood simply tries to exhaust this number. A SYN flood is also easy for attackers to use because it does not require the attacker to process any response from the server. This means that the attacker can use a *spoofed* IP address. Nothing forces an IP packet to use the real IP address except that if a computer used a fake IP address, the computer would never receive the responses. But with a SYN flood, the attacker does not want the responses and can use fake IP addresses without any negative consequences (to the attacker). An attacker can flood the server with SYN packets from potentially many (fake) IP addresses.

Stateful firewalls can deal with this problem at least in part. First, the firewall can refuse to have more than a relatively small number of half-open connections at a time from the same IP address. Even if the attacker is using a large number of fake IP addresses, the firewall can also *rate-limit* the number of SYN packets received at one time from any source.

There are other examples of unexpected or unusual packets that can be transmitted by an attacker. Stateful firewalls can usually recognize and block such packets.

DDOS and Amplification Attacks

Unfortunately, modern DOS attacks are usually far more powerful and harder to block. The Mirai attack discussed in Chapter 7, for example, is not restricted to SYN attacks. Because Mirai has taken control of thousands of devices, the attacker has enough power to launch attacks that do full TCP handshakes and send real data. The challenge of this kind of attack is that the firewall has no way of recognizing the bad data from good data. Note that when a DOS attack is distributed across a wide range of source computers, it is called a *distributed denial-of-service* attack, or DDOS (pronounced dee-doss).

Another example of a DDOS attack is the DNS amplification DDOS attacker. As discussed in Appendix C, DNS is (legitimately) used to convert a domain name like *google.com* into an IP address. Network programs need the IP address in order to be able to make the connection. The domain name provides no routing information.

However, DNS can also be abused by attackers. In a DNS amplification attack, an attacker requests DNS records from many DNS resolvers simultaneously but spoofs the sender's address. In other words, they send many DNS requests to many DNS servers. Each server (ignorant that other DNS servers have also been queried) dutifully responds with the requested records. But the attacker has *lied* about who the request came from. This means that when the DNS record is returned back as a result, this data is sent to the spoofed address.

By way of analogy, suppose that you wanted to flood a person's physical mailbox. You could write letters to 10,000 companies asking for a catalog of their products. But instead of putting *your* name and address on the envelope, you put the victim's name and address. Now 10,000 companies send their catalogs to the victim's mailbox at more or less the same time.

Similarly, if the attacker sent DNS requests with the victim's IP address, then all of the DNS results will be sent to the victim's system or network. If the victim's systems cannot handle this load, they will go offline and the attacker will have achieved their goal of denying service.

But what about the attacker? Is it not a problem that they have to send out all the DNS requests? In the analogy, you had to write 10,000 letters to flood your victim's mailbox. But note that the catalog sent back from the vendor is *bigger* than the letter. If each request is 1 printed page, and each catalog is, on average, 50 pages, you were able to write 10,000 pages to fill your victim's mailbox with 500,000 pages.

In security, we sometimes refer to this as an *asymmetry*. The attack relies, in part, on the amount of data in a DNS request to be significantly smaller than the data in the DNS response.

One way to make an even more dangerous DDOS attack is to combine the DNS amplification attack with a botnet. In this approach, the attacker sends commands to the bots to launch a DNS amplification attack against a target victim. When the bots receive this command, they send out spoofed DNS requests which are then going to send traffic to the victim's system or network. This type of attack is visualized in Figure 8-4. To extend our letter-writing analogy, if you turned a million people into zombies that would follow your every command, you could have each one of them simultaneously write the 10,000 letters to the 10,000 companies requesting catalogs (but with the return address of the victim).

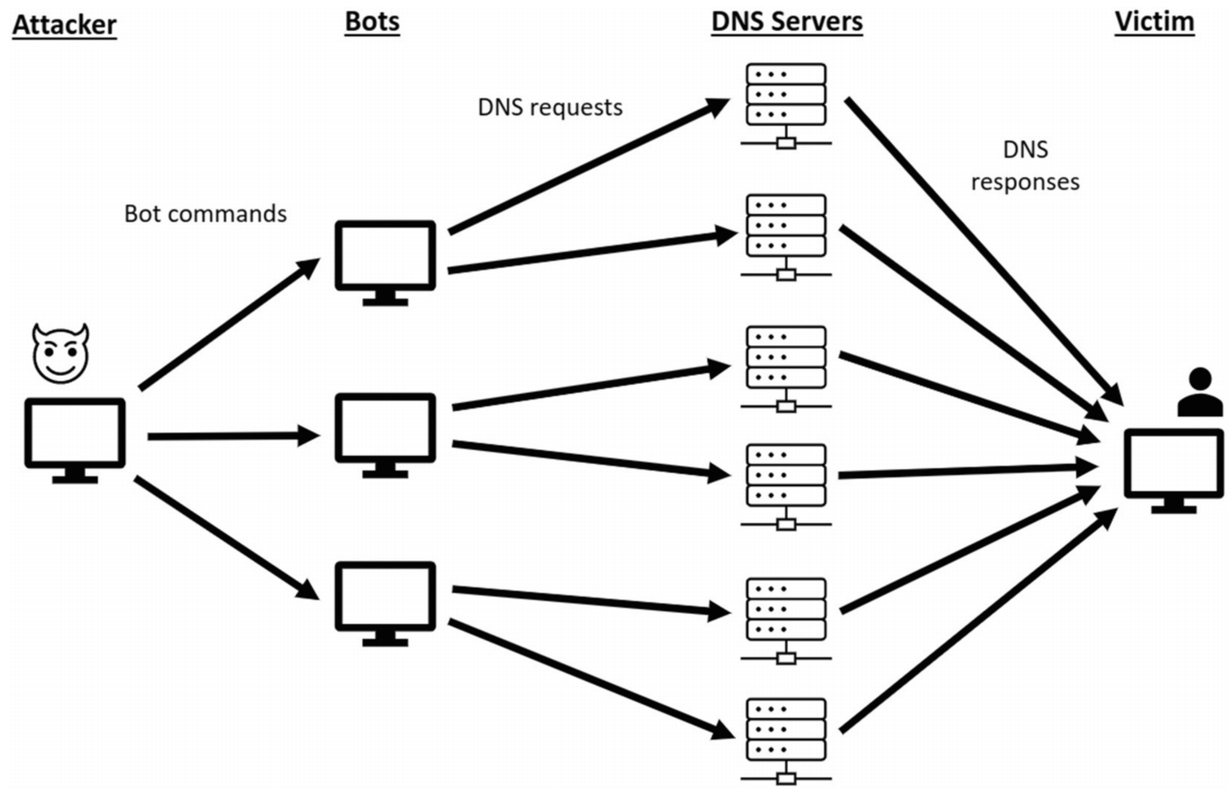


Figure 8-4 The attacker controls a botnet and sends commands to the bots to launch a DNS amplification DDOS attack against a target.

Application-Level Gateways

As I explained in the previous section, packet filtering firewalls in their original form do not process layer-7 information. That means any application data is completely ignored. Layers 3 and 4 are the primary sources of information for filtering.

The reason for this limitation is that layer-7 data is not guaranteed to be in a single packet! I mentioned briefly that network communications have maximum sizes on packets. If a packet is too big, it must be broken up into smaller packets. A small HTTP request will fit in a single request, but even a moderately sized response will not. How can a packet filter block a message on the application layer data when the application layer data is not all contained within a packet?

Application-level gateways (ALGs) were the original solution to this problem and emerged more or less at the same time as packet filtering. Some even considered application layer gateways to be a competing rather than complementary technology. At the very least, these gateways, also called “application layer gateways” or just “application gateways,” were described as being an “opposite” to packet filtering. Whereas packet filtering was a general problem for which rules could be written to solve more specific problems, an application gateway was a specialized solution to a specific problem.

An *application-level gateway* is a service running on a device that serves as an intermediary for a specific network application. That means for a network with a web server, there would be an ALG for just that server. If the network had its own email server, a separate ALG would be needed for the email server. The purpose of each ALG is to monitor, at an application level, the traffic going into and out of the system.

ALGs can be run on the firewall itself, but they can also be run on other machines in the LAN. The network is configured so that no data can reach the real server directly. Instead, all data is passed through the ALG in both directions. When data reaches the ALG, the application data is fully decapsulated from TCP and reconstructed according to application-specific rules. The reconstructed data can be analyzed and screened for security policy compliance before being retransmitted on the network to the real server.

Some ALGs were intended to be completely transparent to both client and server. But for some interactive applications, an interactive ALG was required [213]. Either way, the key feature was application-level analysis. Web traffic, for example, can be monitored for unusual URLs or unusual requests.

Conceptually, a network could be completely locked down with no permitted traffic in or out except through ALGs. Although there would have to be an ALG for each application that was meant to traverse the boundary, one approach is to not permit much network traffic to pass.

In practice, an ALG-only approach is not viable as there are just too many services that need to be configured and used, even in the 1990s. But ALGs and packet filtering networks could often be used together in complementary ways.

For example, FTP is a service that is almost completely unused today but was common in the 1990s. FTP servers could host data, often for retrieval, but also for uploading to storage. Prior to services like Dropbox and Google Drive, FTP servers were a common way of exchanging and distributing data.

FTP, however, was an unusual protocol in that it used two different TCP connections. One connection was used for control messages, and the other connection was used for the actual transmission of data. The data transfer port number, however, was not fixed. It was chosen during the communications and could be picked at random. In these cases, it was impossible to have a firewall preemptively have the data transfer port open because the firewall had no idea which port would be put into use.

Some vendors began to produce firewalls that could be paired with an FTP ALG. The FTP ALG would monitor the communications on the control channel. When a new port was opened for the data communications, the ALG would instruct the firewall to open that port in real time. The data transmission would begin without interruption. Once finished, the ALG would instruct the firewall to close the port. This is sometimes known as a *pinhole* port.

Even though FTP is hardly used anymore, the concept is an easy illustration of how application processing and packet filtering can be used together.

Layer-7 Firewalls

ALGs provided effective application screening. But as the 1990s and early 2000s went on, it became clear that consumers wanted integrated solutions. There was also a recognition that multiple applications might need similar scanning, and the separation into application-specific processing was not always ideal.

Layer-7 firewalls typically combine general-purpose application filtering as well as the usual packet filtering of a stateful firewall. In order to do this, L7 firewalls usually have built-in buffering capabilities in order to combine TCP packets and reassemble the application data.

Although the true L7 firewalls emerged in the 2000s, there was an interesting predecessor product released earlier. Check Point, which still produces firewalls and security products, released version 3 of their FireWall-1 product in October 1996. Version 3 introduced the "Content Vectoring Protocol" (or CVP), designed to allow the firewall to interact with any compliant third-party scanning technology. Check Point and other security companies collaborated to make it possible to integrate the third-party technology as a plug-in. In particular, Check Point FireWall-1 enabled an antivirus to be run on data coming through the firewall without waiting for the data to be scanned at the host.

The CVP protocol worked by first doing the normal stateful firewall inspection. If the packets were permitted, the firewall would do enough application scanning to figure out if it was one of an approved set of protocols, such as HTTP and email. If so, the data would be routed to another system that operated the virus scanning or other security scanning software. The scanner would report back to the firewall using the CVP protocol about the result.

Modern L7 firewalls are far more advanced. And with the growth in firewall hardware capabilities, many products do not need to use an external system at all. The firewall can handle all of the L7 processing.

One company with extensive work in L7 firewall processing is Palo Alto Networks. Palo Alto released their "Next-Generation Firewall" in 2007. Their firewall included what they call "App-ID." App-ID is designed to identify the application data being sent over a port by examining the data itself.

Stateful firewalls, as discussed already, use the destination port as a "good guess" as to what application is running. If it is port 80, it is *probably* a web server, for example. But there are two big problems with the mapping of port numbers to application data.

First, nobody forces a port to host a certain application. That is convention only. If an attacker has penetrated the network by compromising some server (e.g., using a buffer overflow vulnerability), they will often want to set up their own communications for transferring data off the network or receiving updated commands. In order to not have the data blocked, the attacker will typically look for some kind of setup that does not violate firewall rules. This can either be done by setting up a server on an allowed port or by enabling outbound connections to a Command and Control server operated by the attacker. Either way, the attacker often has to use ports that are already permitted.

Second, some widely used protocols are capable of *tunneling*. Tunneling is when one protocol is encapsulated in another. You already saw this in the network stack. IP is tunneled inside a MAC protocol. The TCP protocol is tunneled inside an IP protocol, and an application message like HTTP is tunneled inside of TCP. But application protocols can tunnel other application protocols as well. HTTP, for example, can be used as a generic tunnel, enabling a protocol that might otherwise be blocked to be transmitted over the “legal” port 80 channel.

App-ID, and technologies like it, does not rely on ports to determine the application. They use decoding systems to identify the application from the data itself. On whatever port the scanning is required, the data is analyzed until the firewall has figured out what it is. If the application is known to provide tunneling (e.g., HTTP), it can be decapsulated and rescanned. Using this kind of L7 processing, specific applications can be permitted or blocked regardless of what port number is being used.

In the subsequent year, Palo Alto Networks introduced Content-ID as well. Content-ID enables functionality like that of an ALG directly within the firewall processing. Malware signatures can be scanned within the content for multiple protocols without needing to defer to another machine. This is more fast and efficient, as well as less of a burden for IT administrators.

Attackers always find new ways to get around security though. One of the most ingenious techniques they have adopted is to use DNS traffic to exfiltrate data. In this scenario, an attacker has already compromised a machine on the network but needs to find a way to get the data out without raising alarms at the firewall or by being blocked by its rules. Fortunately for the attacker (and unfortunately for everyone else), the DNS protocol is almost never blocked because of how crucial it is. And even though DNS messages are small, it is not uncommon for there to be a lot of DNS traffic. An attacker can subvert a system to transmit the data over a DNS communications channel without even triggering App-ID. After all, the data *is* DNS data. It just happens to be carrying exfiltrated data in it and headed to a server that is designed to extract the exfiltrated data from DNS. This is an example of a *covert channel*. It can be caught, but it usually takes some extra effort to configure a system to detect and block this kind of problem.

There is one other interesting problem about DNS that is worth highlighting for the principles that it teaches. If a firewall detects a problem, it should be logged. And, depending on the configuration of the firewall, an alert should be generated to a system administrator. This is useful on outbound communications as well because if an attacker has infected a host, the administrator wants to know as soon as possible which host is compromised so it can be cleaned. So, if the firewall knows that some IP address is "evil" and it sees a host trying to make outbound connections to that IP address, it can alert the administrator to do some deeper digging and investigation.

But sometimes network cooperation means that the firewall cannot detect the *origin* of some malicious operation. Take, for example, if an attacker is trying to exfiltrate data to a malicious domain: "pwned.badguysrus.tech." If the attacker tries to send data to this domain, the first thing that happens is an attempt to resolve the domain name using DNS. For most networks, however, there is a DNS server within the LAN. If the server does not have the IP address, it makes the recursive call to a DNS server outside the LAN.

The firewall can detect the LAN DNS server making the recursive call. And if "pwned.badguysrus.tech" has been flagged as a malicious site, it can block the recursive call so that the LAN DNS cannot resolve it either. The attacker, unless they have another way of resolving the domain, will not be able to exfiltrate data.

It is very good that they will not exfiltrate data. The firewall did its primary job. But in terms of finding out which host sent the request, the firewall cannot help. The DNS request *did not come from the infected host*. *It came from the internal DNS server in response to a query from* *the infected host*. The firewall does not have the necessary information to help an administrator track down the problem.

One solution is to have the DNS server log every request and then synchronize log files from the firewall and the DNS server. But this is complicated and error prone. Some firewalls now offer an alternative solution: *a DNS sinkhole*. Using this technology, the firewall does not *block* the recursive call of the LAN DNS server, but *lies about the answer* instead. That is, the firewall answers the LAN DNS as if it (the firewall) were the queried DNS server. The firewall responds with an IP address for the malicious domain that will go through the firewall but then go nowhere. This is where the name "sinkhole" comes from. The data just disappears into the void, and any exfiltrated data is lost to the attacker.

But what this solution provides is a means of getting the malicious host to expose itself. When the LAN DNS responds to the DNS query for pwned.badguysrus.tech with the sinkhole address, the infected client will start sending data to the sinkhole address through the firewall, enabling the firewall to detect the compromised host directly in its own logs.

As advanced as firewalls are (and they continue to advance), the nature of the Internet, cloud, and mobile computing has radically shifted the playing field. LAN- and perimeter-based security are rapidly being obsoleted by transition to mobile users and cloud services.

Network Address Translation

One of the reasons for the need to shift from IPv4 to IPv6 is that IPv4 has a limited number of IP addresses. Recall that the addresses in IPv4 are 4 numbers, each of which is between 0 and 255. If you do some math, you will see that there are a maximum of 4,294,967,296 addresses possible.

That may sound like a lot, but there are already more devices hooked up to the Internet than that! It is impossible for each one to get a unique address!

This has actually been a problem for a while. One of the ways that networks have gotten around this limitation is by assigning private or nonroutable IP addresses on local networks. These addresses *are not required to be unique*. They can be reused over and over again (on different networks, not on the same network). Then, in order to make them able to connect to the Internet, their addresses are intercepted, usually at the firewall or gateway, and *modified*—in other words, when the IP packets are changed and modified en route! The local address that should only be used on the LAN is replaced with the firewall's public IP address. The firewall is tasked with keeping track of the incoming packet's address and port information and the rewritten packet's address and port information. When it receives a return packet on the same channel, it rewrites it back to the original sender's address and forwards it there.

To walk through this, let's reuse the browser connecting to CNN once again. One IP range that is nonroutable and commonly used is 192.168.xxx.yyy. Any IP address with this 192.168 prefix is not meant to be routed across the Internet and can be reused on different LANs around the world. Your own laptop has most likely used a 192.168 address on one network or another. So, for our example, I will use 192.168.1.100 for our browser's IP address. Every device on the LAN must have a 192.168.xxx.yyy address including the firewall. I will assign it the address of 192.168.1.1.

But the firewall has *two addresses* because it is connected to two networks. And its address on the other network is a public IP address that is unique worldwide. Let's pretend that its public address is 142.251.32.228.

When the user of the browser sends a message to cnn.com, the browser sends the HTTP request through the network stack where it picks up its TCP and IP headers. When it gets the IP header, it will have a source IP address of 192.168.1.100, its local IP address. It will also have a random source port, which for this example will be 54321. The destination IP will be 151.101.65.67, which is one of cnn.com's IP addresses at the time of this writing. The destination port will be 443, which is the port for encrypted web traffic. Once created, the packet travels on the LAN until it reaches the firewall at 192.168.1.1.

When the firewall receives the packet, it examines it like any gateway does to see if it is meant to be routed and forwarded. As the destination is 151.101.65.67, the firewall knows it needs to send it outbound. But it also knows that the IP address of 192.168.1.100 cannot leave the LAN. It is a nonroutable IP address. So, instead, the firewall performs *Network Address Translation* or NAT. It *rewrites* the IP source address and changes it from 192.168.1.100 to 142.251.32.228, the firewall's public IP address. The source port needs to not be in use, so the port may need to be modified as well. For this example, I will have the firewall change the port from 54321 to 43215.

Internally, the firewall keeps track of the incoming and outgoing address data. The incoming data is 192.168.1.100, 54321, 151.101.65.67, 443. The outgoing data is 142.251.32.228, 43215, 151.101.65.67, 443. These two sets of numbers are linked together. When CNN's web server receives the data, it will have the translated IP and port numbers. When it sends the response, it sends it to this public address and port of the firewall.

When the firewall gets the return packet, it will have a source IP and source port of 151.101.65.67, 443, which is the address and port of the web server. The destination IP and port will be 142.251.32.228, 43215. The firewall will look up these four numbers in its table and see that it is linked to the incoming data of 192.168.1.100, 54321, 151.101.65.67, 443. It rewrites the packet’s destination from 142.251.32.228, 43215 to 192.168.1.100, 54321 and puts the packet out on the LAN.

The packet now finds its way to the browser, which subsequently displays the website contained in the packet.

NAT was not originally designed as a security feature, and I considered putting this section of the chapter under the networking fundamentals. But NAT and nonroutable addresses are so valuable to computer security that I decided to include the concepts here. The reason is *most computers on a LAN cannot be reached from outside the LAN*. In our example, the browser with the address of 192.168.1.100 is unreachable for an attacker trying to send data from the outside of the network. This security feature is so important, even though IPv6 has enough addresses for every device that could ever be put on the planet, it also includes nonroutable IPv6 addresses in order to permit the same kind of security.

Putting It All Together

Having discussed firewall operations in pieces, I will now illustrate how all of these pieces are put together in a single firewall system. For this example, I will use information about the operation of a Palo Alto Networks (PAN) firewall. PAN publishes documentation about the operation of their systems that gives a very good insight into their firewalls’ operations. One of my favorite documents is “Day in the Life of a Packet” [21]. An image from PAN documentation is provided in Figure 8-5.

There are only a couple of key parts of this image that I will discuss and draw your attention to. The flow of this chart is from top to bottom. At the very top, a packet arrives for inspection at the PAN firewall. In the step

Packet Ingress Process, the system extracts L2, L3, and L4 data. That refers to layer 2 (MAC data), layer 3 (IP data), and layer 4 (TCP data). This firewall does layer-7 processing, but it must do that later in the process as I will explain in a moment.

For this example, let us assume that this firewall is processing a TCP session between a system outside the firewall and inside the firewall. A TCP communication session starts with a handshake. Once the handshake is complete, the real data is transferred. So let’s start with the very first packet.

When the firewall receives the very first packet of the TCP session, there is no existing session between the inside and the outside computer. Because this is the first packet of the session, the firewall has no preexisting data about it. The packet processing tries to start the fast path, as shown in the diagram, but there is no existing session and the lookup fails. So it will now switch over to the FW Session Setup/Slowpath.

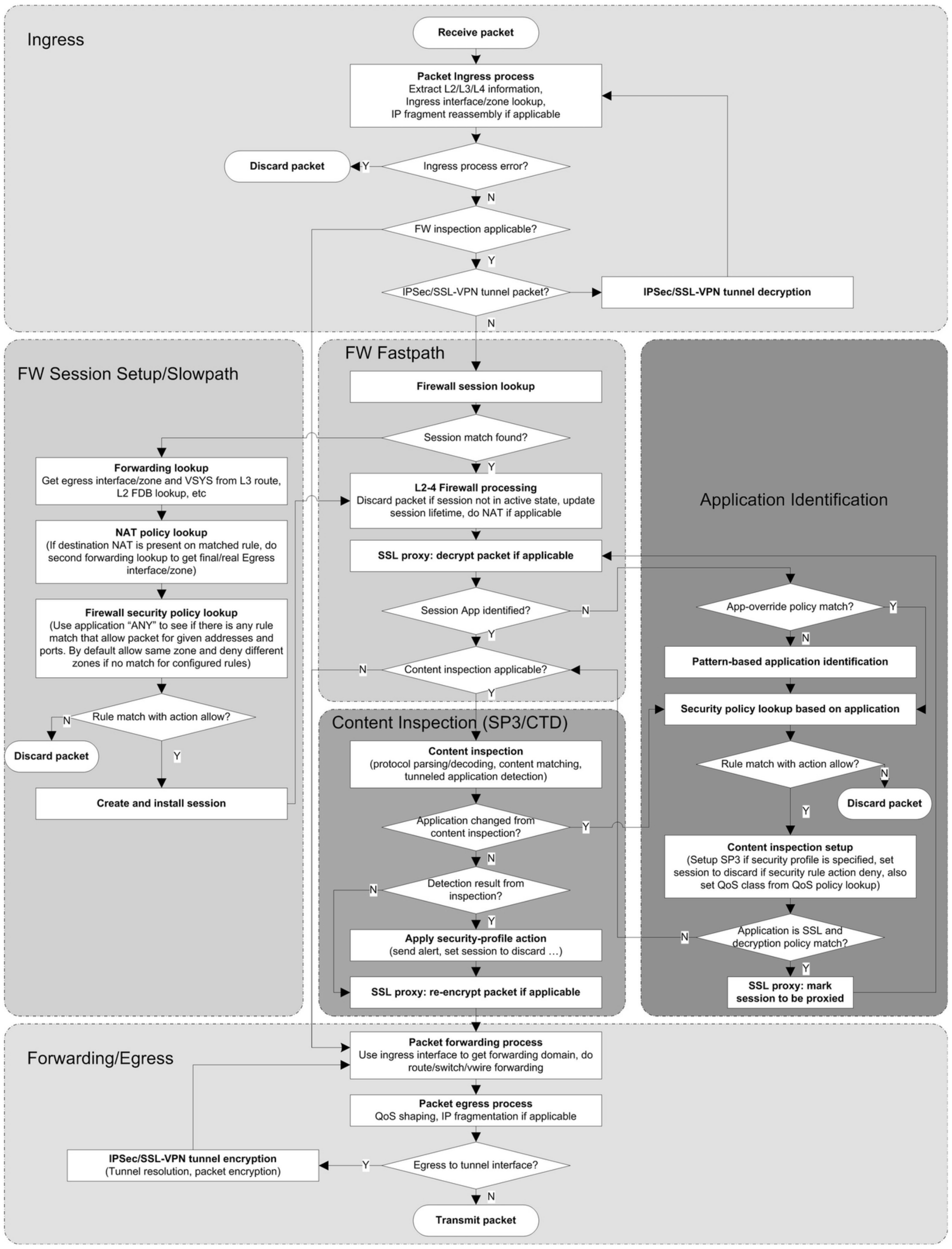

Ingress
Receive packet
Packet Ingress process
Extract L2/L3/L4 information,
Ingress interface/zone lookup,
IP fragment reassembly if applicable
Discard packet
Y
Ingress process error?
N
FW inspection applicable?
Y
IPSec/SSL-VPN tunnel packet?
IPSec/SSL-VPN tunnel decryption
N
FW Session Setup/Slowpath
FW Fastpath
Firewall session lookup
Session match found?
Y
Forwarding lookup
Get egress interface/zone and VSYS from L3 route, L2 FDB lookup, etc
L2-4 Firewall processing
Discard packet if session not in active state, update session lifetime, do NAT if applicable
Application Identification
NAT policy lookup
(If destination NAT is present on matched rule, do second forwarding lookup to get final/real Egress interface/zone)
SSL proxy: decrypt packet if applicable
Firewall security policy lookup
(Use application "ANY" to see if there is any rule match that allow packet for given addresses and ports. By default allow same zone and deny different zones if no match for configured rules)
Session App identified?
N
App-override policy match?
Y
N
Y
Pattern-based application identification
N
Content inspection applicable?
Y
Security policy lookup based on application
Rule match with action allow?
N
Discard packet
Y
Content Inspection (SP3/CTD)
Content inspection
(protocol parsing/decoding, content matching, tunneled application detection)
Rule match with action allow?
N
Discard packet
Create and install session
Application changed from content inspection?
Y
Y
Content inspection setup
(Setup SP3 if security profile is specified, set session to discard if security rule action deny, also set QoS class from QoS policy lookup)
N
Detection result from inspection?
N
Y
Apply security-profile action
(send alert, set session to discard …)
Application is SSL and decryption policy match?
N
Y
SSL proxy: re-encrypt packet if applicable
SSL proxy: mark session to be proxied
Forwarding/Egress
Packet forwarding process
Use ingress interface to get forwarding domain, do route/switch/vwire forwarding
Packet egress process
QoS shaping, IP fragmentation if applicable
IPSec/SSL-VPN tunnel encryption
(Tunnel resolution, packet encryption)
Y
Egress to tunnel interface?
N
Transmit packet

Figure 8-5 The flow of how an incoming packet is processed by a firewall

The key step to notice in the Slowpath is the **Firewall Security Policy Lookup**. Notice that it says it compares addresses and port information. Address information can come from layer 2 (MAC addresses) or layer 3 (IP addresses). Port information comes from layer 4 (TCP port information). The firewall *must* have at least one rule that could permit this combination of addresses and ports. Note that these rules could be stateless or stateful, and both types of operations are useful and important. For example, on the stateless side, the firewall may simply want to block an IP address or TCP port all the time. This operation requires no state. On the other hand, the firewall may want to block TCP packets that are not associated with a TCP connection; this operation requires stateful inspection.

In this example, the first TCP packet is a setup packet (e.g., a SYN packet) and has no application data at all. Because there is no application data to check, application data–based filtration cannot be performed yet. The only data available at this point is address, port information, and other data that is layer 4 and below. At this stage in the Slowpath, processing the first packet, the PAN firewall simply makes sure that there is an "allow" rule that matches this layer-4-and-below data. If so, not only does the firewall allow this first packet to go on its way, it also creates session information in the firewall for this communication stream. Generally speaking, the layer-4-and-below data will *be the same* for all other packets of this communication.

Now, suppose that, after the handshake is complete, the firewall receives the data packets for this same TCP session. These packets, when they hit the Fastpath check, will match the existing session created when the first packet hit the Slowpath. Look back at the Fastpath in Figure ??. Notice that if a session exists, *there is no additional check against the security policy*. The firewall knows that if there is a session it means that the policy is already permitted (so far). That is why this is called the “Fastpath.” By only checking the policy rules on the first packet of a session, packets do not need to be checked against the firewall rules again. This means all of the subsequent packets of the session can go through the firewall much faster.

Once the PAN firewall is receiving data packets, it can also perform the layer-7 (application data) processing. It is important to realize that unlike the processing performed on the first packet of the session in the Slowpath, layer-7 processing is *not packet processing or filtration*. Application data is not confined to a single packet. Thus, the PAN firewall extracts the layer-7 data and stores it for layer-7 processing. One key step is identifying the App-ID through the **Application Identification** process. If the App-ID is already known, as shown in the step **Session App identified** in the Fastpath, then the firewall checks if content inspection is enabled for this App-ID. If so, it will go on to the **Content Inspection** phase. Rules prohibiting certain kinds of applications or certain kinds of content will cause the session to be closed and subsequent communication halted.

I have not talked much about egress. There is no filtering function in this stage; it is for VPN processing, routing, and other transmission-related issues. I have also ignored a lot of steps in the ingress phase. Most of these are not particularly important for this discussion.

One issue that is worth pointing out is *decapsulation*. Sometimes, traffic can be wrapped up in other traffic. That is, one traffic stream can be hidden in another. This can disguise traffic from an application identification process, for example. The PAN firewall has a process for pulling the inner data out (decapsulation) so that it will be correctly processed.

Proxies

Another type of common network device is the proxy. A proxy is so named because it makes requests on behalf of other computers. For example, a web proxy would handle all of the web requests for its clients. One reason for using a proxy is caching. Each time a URL is requested through the proxy, the proxy can keep a cached copy of the data on its local storage. If at a later point there is another request for the same URL from within the network, the proxy can serve its own copy much faster, speeding up access to popular data. As explained in an early 1994 paper:

> Caching of documents has been introduced, giving noticeable speed-ups in retrieve times… The basic idea in caching is simple: store the retrieved document into a local file for further use so it won't be necessary to connect to the remote server the next time that document is requested… [167]

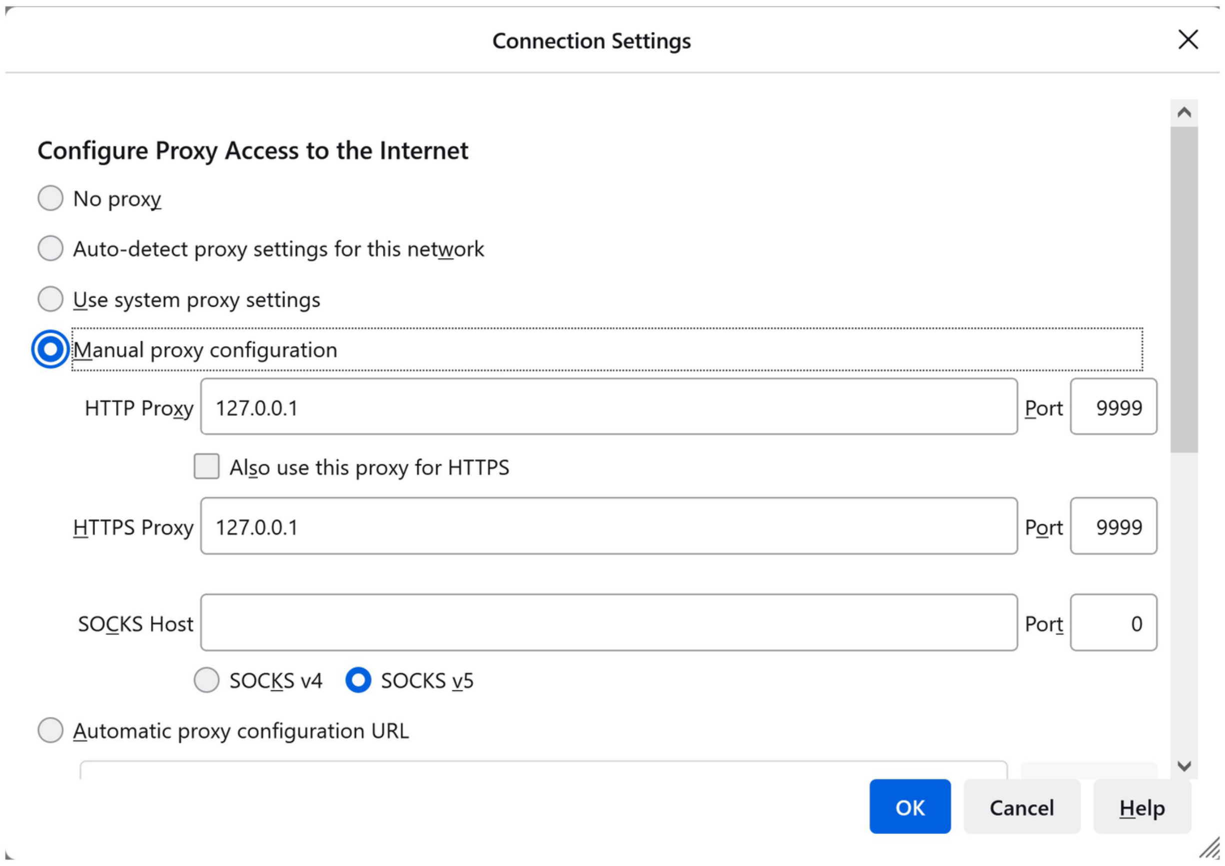
Proxies can also be used for security purposes, serving a very similar purpose to firewalls. If all of the outbound traffic is funneled through one or more proxies, the proxies can scan the outbound traffic as well as the responses and apply security policies. Because gateways also see outbound traffic, firewalls can perform similar functions, and many firewalls can be configured to also provide proxying. Exactly how to configure a system in terms of firewalls, proxies, and other devices depends on individual circumstances and preferences [71].

One of the more commonly known types of proxies is a *forward* proxy. A forward proxy is more or less a proxy for clients. When a client needs to access a certain server or resource, it contacts the proxy. The proxy makes the request to the server on the client's behalf. Sometimes, proxies have to be explicitly configured, and this is usually the case for optional proxies that are chosen at the user's discretion. For example, some browsers allow the user to specify the proxies to be used for surfing the Web as shown in Figure 8-6. Others rely on the operating system's proxy configuration, as illustrated in Figure 8-7.

But even a manually configured proxy can be made mandatory through firewall rules. If a firewall blocks all outbound web traffic except from a web proxy, the user will have no choice but to configure their browser to use the proxy.

More commonly, however, organizations use routing rules and other networking architecture tricks to perform *transparent* proxying. A transparent proxy is generally unnoticed by either the client or the server. By definition, a transparent proxy is not manually configured by the user because that would mean it was not transparent.

The purpose of proxies is most often to provide outbound controls on data. One of the biggest concerns in business environments today is data being accidentally or carelessly sent outside of authorized boundaries. DLP systems, or data loss prevention systems, are used to scan data as it exits the security boundary for data that should not be transmitted. A proxy that performs DLP scanning can proxy web traffic, email traffic, or other data transmission applications. The DLP scans the content for data recognizable as PII, company information, or other data that should not be released.


Connection Settings

Configure Proxy Access to the Internet

No proxy

Auto-detect proxy settings for this network

Use system proxy settings

[✓ Manual proxy configuration]

HTTP Proxy 127.0.0.1 Port 9999

Also use this proxy for HTTPS

HTTPS Proxy 127.0.0.1 Port 9999

SOCKS Host Port 0

SOCKS v4 [✓ SOCKS v5]

Automatic proxy configuration URL

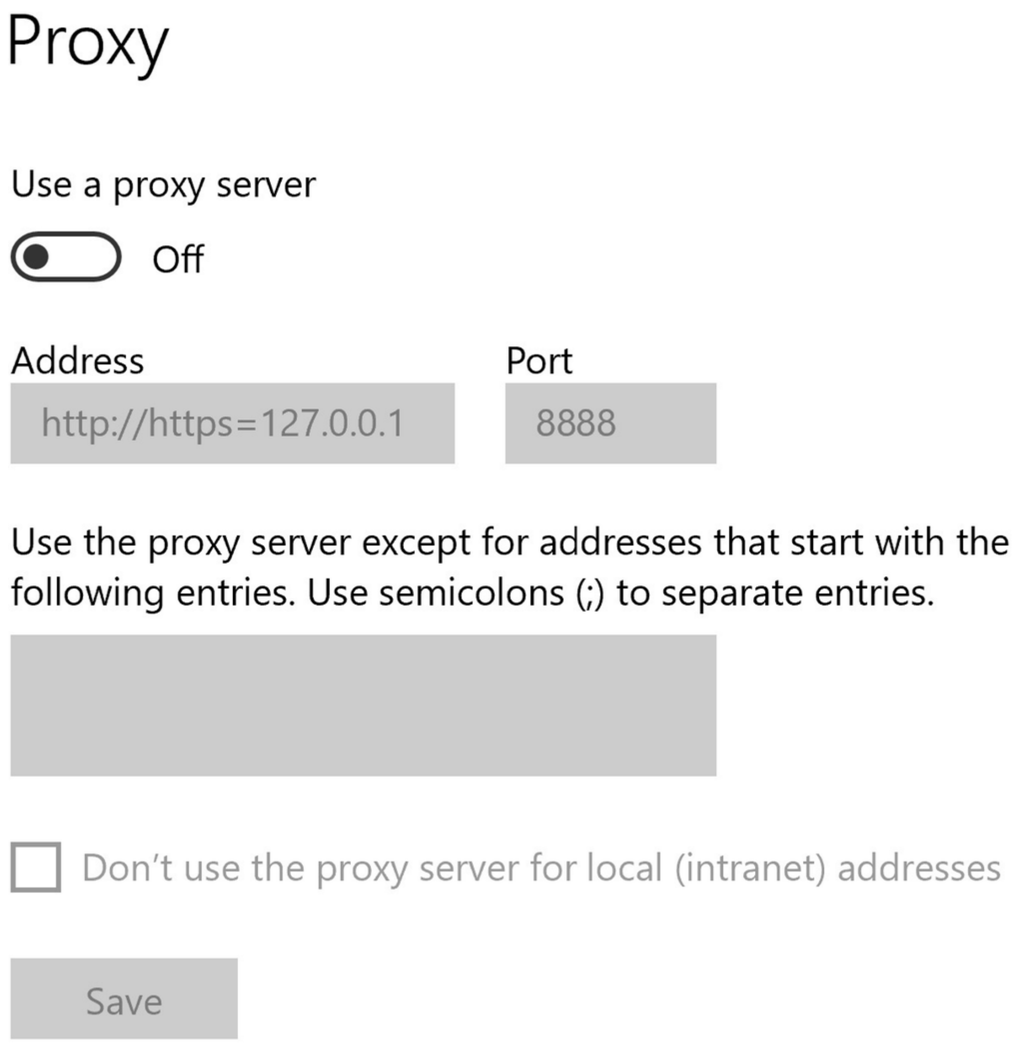
Figure 8-6 Proxy configuration for the Firefox browser. This browser permits a proxy configuration different from the operating system

Forward proxies can also be used to enforce malware scanning and other security screenings. Another purpose is authorization. Many universities have access to online journal and conference proceedings. People outside of the organization have to pay for access for these tools. But universities can provide their staff with a portal through the university to the online resources. This portal is essentially a proxy.

The inverse of a forward proxy is a reverse proxy. As you might have guessed, a reverse proxy is for servers. A reverse proxy works by listening for connections. When a new connection is received, the reverse proxy makes a connection to the real server and mediates the communications.

Reverse proxies provide servers with a number of valuable security features. For one thing, the reverse proxy can perform *load balancing*. This means that there can be more than one real server that can answer the requests. The reverse proxy can choose from any of these real servers based on whichever one has the least load. While not a purely security property, this enables systems to survive DDOS attacks more easily.

Another purpose for the reverse proxy is security screenings (just like for forward proxies). Requests can be screened for suspicious requests or malicious uploads. Additional access controls can also be enforced at the reverse proxy. These kinds of security capabilities could be built into the real server directly, but reverse proxies enable modularity, configurability, and focused security management.


Proxy

Use a proxy server

Off

Address: http://https=127.0.0.1

Port: 8888

Use the proxy server except for addresses that start with the following entries. Use semicolons (;) to separate entries.

☐ Don’t use the proxy server for local (intranet) addresses

Save

Figure 8-7 Browsers like Google’s Chrome browser for Windows do not have their own proxy configuration. They rely on the operating system proxy configuration

---

Virtual Private Networks

Firewalls, as discussed earlier, are a defensive filtration technique. There are other systems that enable protected access, like a VPN. The term VPN, or Virtual Private Network, has in recent years become a more widely used term. Unfortunately, much of that use is incorrect or, at the very least, misunderstood. A *Virtual Private Network* is where two remote networks are bridged using a secure connection.

One common configuration for a VPN is to join an organization's remote networks together into a single network. Branch offices can be linked to the home office using VPN connections to create a seamless LAN. This is sometimes called the "site-to-site VPN."

Another common configuration for a VPN is to have a single machine connect to a remote LAN and join it as if it were attached to it directly. This is sometimes called the "remote access VPN."

Both configurations do the same kind of bridging between networks. The only difference is the scale and how much configuration is required. A site-to-site VPN can require more configuration to ensure that both sites (e.g., the main office and the branch office) can work together in either direction. Additional configuration may be needed if there are services that should only work on one side of the network.

VPNs work by creating an encrypted tunnel between the two networks. All the data going over the public Internet is confidential and authenticated. When it is received and decapsulated, it is put onto the network as if it had been generated locally. There are also requirements for compatible addressing and ensuring that broadcasts are propagated on both sides of the VPN connection.

There are multiple technologies for creating VPNs at different layers of the network. One technology is called *ipsec*, which stands for IP security protocol. This protocol works down at layer 3 and creates special IP packets that encapsulate encrypted IP packets. Thus, packets coming off one network (e.g., the branch office's network) are encrypted from layer 3 and up. This kind of VPN requires either preshared keys for securing the connection or some kind of certificate-based system.

Alternatively, the tunnel can be created at the application layer. These systems typically use TLS to create a tunnel between systems. The advantage to this kind of connection is that TLS is an already commonly configured technology, and there is usually less configuration and setup. Moreover, because TLS operates at a higher layer, it can take advantage of things like NAT. Recall that NAT rewrites IP source addresses. This causes problems for ipsec without more complicated protocols. TLS VPNs, on the other hand, have no trouble at all with NAT. TLS VPNs, like standard TLS, typically just need a certificate for a server side and a shared password for authenticating the client. This kind of operation is very fascinating from an OSI model perspective. A layer-3 connection (IP protocol) is bridged using a layer-7 protocol!

There is another kind of VPN service that is used for large corporations with substantial service contracts from ISPs. For very large ISPs, their own set of backend systems is almost their own private Internet. Instead of the customers provisioning their own VPN with their own equipment, the ISP, providing service for both the main office and all of the branches, can configure “provider-provisioned VPNs” on what are called “provider-edge” equipment. These VPNs are used to keep one customer’s data separate from another and also advanced features like the customer’s different networks *discovering* each other automatically.

In recent years, most consumers associate VPNs with services promising some amount of network security and anonymity when surfing the Internet. These services work by hosting VPN servers on the Internet and providing connection software to their subscribers. When the subscriber activates the VPN connection software, their computer is bridged to the VPN’s network. Normally, when someone browses the Web, their web requests first go to the ISP before being routed toward their ultimate destination. When a VPN subscriber connects to their VPN, most or all of their network traffic is *encrypted* and sent to the VPN first and then sent from the VPN to the final destinations.

This has two supposed benefits. First, the ISP cannot monitor the websites the subscriber is visiting because all of the web traffic goes in an encrypted tunnel to the VPN. The ISP can see that the subscriber is using a VPN but cannot see what traffic is passing through to the VPN. The second benefit claimed by these services is that the website being visited does not know the subscriber’s true IP address. Thus, the subscriber maintains some anonymity from both their ISP and from the websites they visit.

However, these advantages are limited. Although the ISP does not know the subscriber’s traffic*, the VPN service does*. Whether or not this is better depends on the VPN. For example, if the VPN service is located within the same territorial jurisdiction as the ISP, it is most likely bound to the same legal requirements for productions of information by law enforcement. If the VPN service is in a remote jurisdiction, the subscriber would have little recourse if the VPN service went rogue and began selling their data or spying on their network usage. Similarly, it is not clear whether it is better for VPNs to know all of the subscriber’s traffic rather than the individual websites that they visit.

There is, however, a very strong reason to use VPNs: public WiFi or other public networks. When in an airport, hotel, or other location with public network access, using a VPN is an extremely good practice for security. When connected to a public network, the device is sharing that network with any other number of unknown individuals, some of whom might be malicious. Modern HTTPS will generally prevent these malicious actors from, for example, being able to snoop through someone’s online bank access. But there are different kinds of attacks they might try to break these protections. Moreover, they can try to attack your device’s open network ports as well. Using a VPN locks a device from being accessed by anyone else on the public network and ensures that all data is correctly encrypted and delivered to the right place.

Intrusion Detection and Prevention

In addition to systems that attempt to block threats from entering the network, additional security technologies can be used to detect threats that are currently active in the network. These same technologies are also useful for forensic analysis for attacks that are over, but require remediation and recovery. These systems are generically called Intrusion Detection Systems, or IDS. IDS is typically broken down into Host IDS (HIDS) and Network IDS (NIDS). Given the topic of this chapter, this section focuses on NIDS specifically.

Interestingly, IDS systems have not changed much at the theoretical level since their inception in the 1990s. Some interesting changes have occurred in the past few years through the use of cloud technology and a deeper integration with other tools. I will return to that topic at the end of the section.

The standard IDS system is largely an *alarm* system, designed to alert human operators the (possible) presence of a threat. These systems operate in a kind of data pipeline to reach conclusions about the presence of a threat.

In the first phase, data is simply gathered. IDSs are known for collecting, or at least analyzing, a significant amount of data. IDS data can be collected directly from raw packets collected on the network or more processed data from other “sensors” in the system. Processed data might include, for example, log files. Sensors can also be placed on hosts. This does not make it a HIDS system if these sensors are used to create events that are transmitted to a central processing system for understanding network data. Host sensors are useful for processing decrypted network data as otherwise the network sensors cannot understand the encrypted portions.

In the second phase, the data has to go through another level of processing to extract the characteristics of the data that matter to the IDS. Characteristics used as inputs to an analysis engine are often called “features.” Feature selection is important. Leaving out critical features reduces the accuracy and effectiveness of the IDS analysis engine. But having too many features that are not necessary reduces performance sometimes to the point of making the system nonfunctional. Some examples of features from networking data might be source IP, packet type, and data length. Like with firewalls, IDS can use stateful analysis to better contextualize the received data.

Once feature extraction is complete, the third phase begins. This phase runs the IDS analysis engine on the features to produce a result.

Finally, in phase four, the result triggers one or more reactions.

There are different kinds of IDS analyses that can be done for phase three. Most of them fall into one of two categories: *signatures* and *heuristics*. These, of course, are two of the broad categories for malware analysis, and it is not coincidental. Much like malware analysis is a classification problem, so is network behavior. Much of the same limitations, strengths, and weaknesses apply here.

IDS signatures, like malware signatures, can only work based on past experience with known attacks. Instead of scanning a file for matching bytes, an IDS signature, sometimes called an *Indicator of Compromise* or IOC, monitors network events for a known bad pattern. On raw packets, for example, there are various signatures for combinations of patterns that indicate an attack or threat.

A port scan by itself may only be mildly concerning, but a port scan followed by packets sent to specific, relatively unused ports may be far more indicative of an intrusion. It should be noted that these signatures involve a lot more guessing than malware signatures.

Heuristics, on the other hand, try to look for *anomalies* in the events digested by the analysis engine. This approach typically begins by identifying baseline event capture that represents the system in a “good” state. This capture may take several weeks to get a thorough collection of events from which to establish the baseline. Once the baseline is established, the goal of the analysis system is to detect deviations from the standard behavior.

Regardless of the approach taken, the analysis engine will spit out some kind of decision. From there, the IDS will decide on a course of action. Most often, this involves alerting a human operator.

Alternatively, the system can attempt to take proactive action. For example, if a network connection is believed to be malicious, it is possible to terminate it and block reconnections. One way this can be done is if the IDS interfaces with the firewall and can add new rules. When an IDS can proactively protect the network, it is sometimes called an IPS, or *Intrusion Prevention System*.

There are many problems with IDS and IPS systems. IPSs especially are so problematic that they are almost never used. The biggest problem with IDS (and why IPS is generally not viable) is there are just too many false alarms. In the case of IDS, this means that the human operators are getting too many alerts. But in the case of IPS, it may mean interruption of important services that just happen to trigger the signature or heuristic detection.

Story Time: A Deadly IPS in Cyberpunk

The *Shadowrun* role-playing game (RPG) is set in a future time period and involves both advanced technology and magic, science fiction and fantasy. The Internet of the future is called the Matrix (this world was crafted and created before the Matrix movies arrived on the scene, so the RPG did not steal the idea from them!). Deckers (or hackers) plug their computers directly into their brains and enter the Matrix as a VR experience. Supposedly, they interact with the Internet at the speed of thought, so they can customize their hacking faster than a human could without the neural connection.

The problem for the deckers is the Black Intrusion Countermeasures (IC or "Ice"). Black IC is a countermeasure that recognizes "foreign" elements, such as the decker, and will literally try to kill them. In this mythology, if the Black IC can take over the decker's computer, it can cause a biofeedback loop that can blow out the decker's brains or cause their heart to stop.

The book *Shadowplay* is a novelization set in this world. In the prologue, it describes one decker seeing another getting killed by IC. It describes it this way:

> A decker entombed in ice. One of those sights you hope never to see...
>
> He was dying, I knew that. In the real world, it would have been over in an instant. In the matrix the ice picks up the decker's icon, then dumps its signal into his cyberdeck. The deck's filters overload, pouring the signal through the datajack, straight into the decker's brain. And then... who knows? Convulsions, the kind strong enough to break his bones. Or his blood pressure spikes to high one of the vessels in his brain blows out. Or maybe his heart stops, just like that. Biofeedback, it kills you as quick and as sure as a bullet in the head. [107]

Of course, as an RPG, this is a fun setup for amazing adventures in a virtual reality version of the Internet. In real life, such a thing would never, ever work if for no other reason than false positives. IPSs are often seen as too problematic for dropping valid connections that *look* bad to the detection algorithm. Can you imagine if workers, or maybe a customer, or the CEO got killed because the Black IC made a mistake? Nobody, not even in a dystopian corporatist world, would be willing to even use these kinds of systems.

But I like the name Black Ice much better than IPS. I wonder if I could start a trend?

Unlike malware signatures, IDS signatures still tend to involve some amount of guessing and heuristics as well, which is why IDS signatures produce false positives. Unlike malware signatures that match software to software, IDS signatures attempt to identify *misuse*. Misuse detection is inherently more prone to false alarms because it has to capture intent.

Anomaly detection is even worse. This kind of analysis has been tried repeatedly since the 1990s, but there is no really good way to capture “normalcy.” Anomaly detection requires an accurate baseline in order to determine anomalous behavior. But if you think about human behavior, *exceptional behavior is often the rule*. A user might regularly work during business hours, but may occasionally work 16-hour days (or longer) during a crunch time on a project. That kind of behavior is exactly like the kind of behavior an anomaly detector should catch because that might be an indication of an attack too. But most of the time, it is not an attacker, and that just means a lot of false alarms.

Because of the problems with alerts, many times the real value of the IDS is actually after the attack is over. IDS systems tend to store a certain amount of data that permit forensic analysis of what happened when it is determined that something went wrong. Once the human operators know they are looking for something, IDS event data and analysis can be very useful in tracking down what happened.

Defensive Deception

Deception is not just for the enemy. I briefly alluded to this in the summary for Chapter 1. Defensive deception, which is deception designed to *confuse*, *mislead*, and/or *disrupt* malicious actors, has sometimes been classified as a part of IDS/IPS technologies [73, Chapter 15], [233], [60, Chapter 27], [40, Chapter 21]. On the other hand, some authors argue that it should be a distinct "third line of defense." These authors argue that defensive deception systems "could then lie, cheat, and mislead such anomalous users to prevent them from achieving attack goals, even when they have obtained access and fooled the intrusion-detection system" [225, Chapter 1]. Both models have merit. It is true that many defensive deception systems have intrusion detection and identification as a goal. However, cyberdeception is different from most other forms of IDS/IPS. And, as pointed out in the preceding quotation, it has *additional* goals such as making any actions of the attacker within the system benign. It is certainly worth studying deception as a separate topic from, or at least a subtopic of, intrusion detection and prevention. In this section, I will give a short history of defensive deception as well as a survey of some of the state of the art.[6]

One of the best early examples of cyberdeception comes from Cliff Stoll's book *The Cuckoo's Egg* [253]. This book autobiographically describes Stoll's experience related to his discovery and tracking of a group of German hackers in the 1980s. These infiltrators were breaking into American computer systems including Stoll’s computers at Lawrence Berkeley National Laboratory as well as military systems. Stoll, who was working as a computer systems administrator at the time, stumbled upon the intrusion while investigating a 75-cent discrepancy in accounting records.

Stoll ran into many challenges while attempting to solve the mystery. At the time, when computer crime was so nascent, there was little support from law enforcement. Stoll had to invent solutions as he went in his quest to uncover the identities of the cyber-invaders. In fact, he often felt like he had little support from everyone. One exception was his girlfriend Martha.

According to Stoll’s account, Martha was the source for one of the most clever ideas in the book.[7] Stoll tells the story that while discussing the problem of not getting government help to find the intruder’s identity, Martha proposed setting their own trap using deceptive data. And she did so, using a fake foreign accent to sound like a spy revealing her dastardly plan:

> “Boris? Darlink, I hev a plan...”
>
> “Yes, Natasha?”
>
> “Ees time for ze secret plan 35B.”
>
> “Brilliant, Natasha! Zat will vork perfectly! Ah, darlink... vhat is secret plan 35B?”
>
> “Vell, you see, zee spy from Hannover seeks ze secret information, yes?” Martha said. “We give him just vhat he wants–secret military spy secrets. Lot of zem. Oodles of secrets.”
>
> “Tell me, Natasha dahlink, zees secrets, vhere shall ve get them from? Ve don’t know any military secrets.”
>
> “Ve make zem up, Boris!”[8]

Stoll and Martha set to work operationalizing Martha’s idea. In order to track the intruder effectively, they needed the intruder to be connected for a sufficient amount of time. Stoll calculated that to have him in the system for two hours they would need to have about 150,000 words (based off of the download speeds available at the time). How could they possibly come up with that many documents? They would want him connected for multiple two-hour periods, so they would need multiple batches of 150,000 word document sets.

Martha had that figured out too. She explained that it would be easy to take a bunch of nonconfidential government documents and do a very small amount of modification to make them look top secret. Stoll apparently received many nonconfidential government directives from the department of energy. Up until this time, they had just been worthless (as many government documents and directives are). Stoll proposed that they would convert these worthless, bureaucratic mumbo-jumbo documents into deceptive documents that looked like “state secret.”

Again, Martha had a key insight about the deception. If the deceptive documents looked *too* high value (e.g., “TOP SECRET” and “ULTRA CLASSIFIED”), the hacker might get suspicious. She proposed that the documents be just “forbidden” enough to keep the intruder interested, but “low-key” so as to not make him cut and run.

At this point in the narrative, Stoll interjects that another friend was present, Claudia, and she proposed yet another good idea.[9] She suggested creating a fake form letter to submit for more information. Government programs often solicit engagement for various programs; an invitation to get more information would not be unusual from the perspective of the intruder. However, for Stoll and his compatriots, it might be a means of getting the intruder to identify themselves, complete with return address!

Stoll did contribute one very important idea of his own to this plot: a fake user. The hacker was breaking into different people's accounts. There would need to be an account for them to break into and find this data. Stoll proposed creating a fake secretary that was supposedly handling a lot of paperwork. This would also make it easy to have some repetitive documents (reducing the need to generate unique content) because the secretary would have a lot of different drafts of the same document. Stoll also developed a fake mailing list by taking the names of real people in his lab and changing their prefix from "Mr." to "Lieutenant" and the like. A few references to the "Pentagon" were thrown in for good measure.

This story, entertaining all by itself, sets an amazing amount of groundwork for modern defensive deception. Here are some of the key ideas generated by Stoll, Martha, and Claudia:

1. **Decoys** are fake components of the system and can include data, systems, (fake) people, and so forth.
2. **Bait** are decoys meant to attract attackers.
3. **Templates**, perhaps drawn from real data, can be used to generate bait and decoys.
4. **Psychology** is essential both in terms of knowing what will attract and what will repel an adversary.
5. **Active traps** can induce an attacker to reveal information about themselves.

In the 1990s, vendors began to produce products that built on these concepts. Some of the terminology evolved, primarily with the introduction of the term *honeypot*. Unfortunately, the term *honeypot* does not have a universally accepted, formal definition. In 2003, researchers compiled a nonexhaustive list [207]:[10]

- A honeypot is a security resource whose value lies in being probed, attacked, or compromised.
- A honeypot is a resource which pretends to be a real target. A honeypot is expected to be attacked or compromised. The main goals are the distraction of an attacker and the gain of information about an attack and the attacker.
- An Internet-attached server that acts as a decoy, luring in potential hackers in order to study their activities and monitor how they are able to break into a system. Honeypots are designed to mimic systems that an intruder would like to break into but limit the intruder from having access to an entire network. If a honeypot is successful, the intruder will have no idea that they are being tricked and monitored.
- Within the realm of computer security, a honeypot is a computer system designed to capture all traffic and activity directed to the system. While honeypots can be set up to perform simple network services in conjunction with capturing network traffic, most are designed strictly as a “lure” for would-be attackers. Honeypots differ from regular network systems in that considerably greater emphasis is placed on logging all activity to the site, either by the honeypot itself or through the use of a network/packet sniffer. A honeypot is designed to look like something an intruder can attack to gain access to a given system.

The term is thrown about imprecisely in various sources. Many sources identify a honeypot as being very specifically a computer system [178], [73, Chapter 15] or even specifically a server [60, Chapter 27]. Associating the term with a machine is so common that there are different classifications for honeypot systems such as whether or not they have “low” or “high” interactions [169]. For example, a system that just receives connections and logs what happens might be called a “low interaction” honeypot. A system that simulates a full computer, complete with fake users, documents, and programs, would definitely be a “high interaction” honeypot.

I personally prefer the first definition in the preceding list. It comes from Lance Spitzner who wrote a series of papers about honeypots in the early 2000s. According to Spitzner, a honeypot is a “security resource,” meaning that it can be data, systems, people, or any other “resource.” Spitzner explains, “... in the definition... we do not state a honeypot has to be a computer, merely that its a resource that you want the bad guys to interact with.” Spitzner also explicitly defines *honeytoken* to be a subtype of honeypot that is *not* a machine. He describes a honeytoken as “a honeypot which is not a computer... [and] can be any type of digital entity... [such as] a credit card number, Excel spreadsheet, PowerPoint presentation, a database entry, or even a bogus login” [249].

Please note, however, there is an ever-so-slight difference, conceptually, at least, between a honeytoken and fake data. The honeytoken is *the resource the adversary is meant to attack*. Generally, the term honeytoken is meant to refer to data that the defenders specifically want to use against the attacker. From Stoll’s example tracking the German hacker, the invitation to write for more information was definitely a honeytoken. It was fake data from top to bottom. McRae and Vaughn explain that, “The key to using honeytokens is to give the token unique identifiable elements to guarantee that the only access to that token would be by unauthorized parties. If the token could be viewed in normal interaction with a system, the token’s tracking ability is compromised” [177].

Another example of a honeytoken is a *fake email address*. In a 2005 research paper, researchers introduced *HoneySpam*, a system designed to quickly and automatically identify and block spammers. One component of the system was fake email addresses, where by fake I mean that they were not used by any real users. Instead, these email addresses were designed to lure spammers into sending spam to them for automatic identification [38].

In addition to honeypots and honeytokens, there is one other related term: honeynets. According to Spitzner, honeynets are "entire networks of computers to be attacked" [248]. Spitzner goes on to describe some of the advantages of the honeynet:

1. Organizations can insert any application or device normally used on the network.
2. All traffic can be observed without being observed in return (like a "one-way mirror").
3. Infinitely customizable.

Notably, a honeynet does not have to be a set of real machines. In 2004, Niels Provos, then at Google, presented the idea of a virtual honeypot that could be connected to a network [210].

Research within the past ten years or so includes decoys for identifying proprietary software theft [203], deceptive web applications [189], and honeypots for IoT devices [164].

In terms of deployed systems, one early example is *ManTrap*, which was purchased by Symantec and renamed Symantec Decoy Server. The initial system was developed in the early 2000s. By 2005 (post acquisition), it had a long list of features including automatic simulated email generation, host and network honeypot components, and automated responsive capabilities [256].

Another example of an early deployed technology is Fred Cohen’s Deception Toolkit (DTK). This toolkit was meant to be a very generic system that could be programmed to emulate any network service. The goal was that a remote adversary would connect to the toolkit-built service and interact with it. The level of interaction available depended on the programming. The toolkit itself provided little to no programming but, instead, provided a library to enable easy (or easier) programming [77].

More recently, companies like Attivo Networks are actively deploying enterprise-scale deception to clients and customers. The concepts are the same but the virtualization, bait, and realism have all increased [14].

Deception is a very powerful tool for one very simple reason: *no legitimate users should be using the decoy/honeypot*.

> Unlike other intrusion detection measures there are no false positives with a honeypot. All IDS systems produce false positives to varying degrees. This is because there is always a chance that valid traffic will match the characteristics the IDS uses to detect attacks. This is not the case with a honeypot. Any communication with a honeypot is suspect. This is because the honeypot is not used for any purpose other than detecting attacks. There is no valid traffic to produce false positives. [131]

While this quotation is a little bit optimistic (it may be possible for a legitimate user to stumble into a honeypot), the point is correct. Technologies like honeypots are never, or almost never, used by legitimate users. Therefore, *any* use of a honeypot is probably malicious.

Network Architectures

In addition to all of the technologies discussed in this chapter, one of the other important elements of classical network defense is the architecture of the network itself.

One of the most important design elements for classical networks was *segmentation*. Rather than creating a single LAN for an entire organization, computer systems, resources, and servers that were interrelated would be put on a *network segment*. The segment behaved like its own LAN and would often have its own firewall, even for communications within the company.

The basic idea here is that two different groups, such as the marketing department and the research and development team, should not be sharing the same network resources directly. Communication is allowed, of course, but it needs to be mediated (e.g., by a firewall) and a policy enforced on the communications between them.

Segmentation also permits the partitioning of the network into different security levels. The most sensitive resources might need extra security hardening. This may include very limited access, limited services, and limited data transmissions. Those kinds of rules may only apply to a very small number of hosts. In such circumstances, it makes sense to isolate the machines into their own protected subnet.

Even IDS systems work better in segments. By reducing the number of machines and systems on a given network, the IDS deals with a much smaller number of events and a proportionally smaller number of alerts.

One very special kind of segment is called a *demilitarized zone* or DMZ. A DMZ is a segment meant exclusively for high-risk systems, such as those that have a public Internet presence. In the “old days,” this would include companies’ web servers, email servers, and other public systems. Typically, the DMZ segment would be placed next to an “edge” firewall, or a firewall connected directly to the Internet (as opposed to an internal firewall for internal segments). The DMZ would also have connections into the rest of the LAN (or some segments of the LAN), but there would be a separate firewall separating the DMZ from the rest of the company network. The insight that led to the DMZ was that the public-facing systems were the most likely to be compromised, and there should be a security screen between these high-risk systems and all of the others (Figure 8-8).

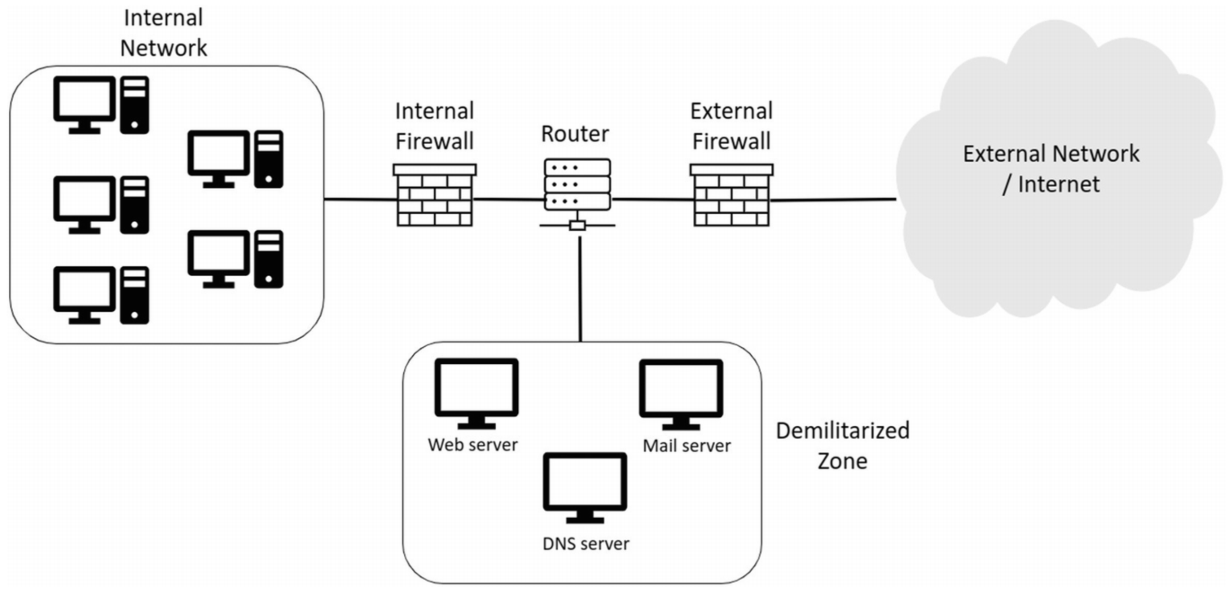


Figure 8-8 Demilitarized zone is placed between the internal network of the organization and the external network which may be the Internet. The DMZ holds public-facing services such as the web server hosting the website, the mail server for emails, the DNS server, etc

If you have been paying attention, you may have noticed that there is a lot of complexity in a company’s network systems. Remember, until just a few years ago, there was no real cloud offerings for most of these services, and companies largely had all of these systems on-prem. How is it possible to manage all of the segments, firewalls, DMZs, IDSs, and other security technologies coherently?

On a network organizational level, it was important to centralize the day-to-day security operations. This led to the development of the *SOC*, or Security Operations Center. The SOC is usually in a central location and includes a dedicated security team that is focused exclusively on keeping the network safe. This typically includes monitoring events in real time, investigating potential incidents, and keeping the system up to date. The SOC tracks host and network security, patch management, signature updates for IDS and anti-malware, and configuration of systems.

The other major innovation was the development of a super IDS known as a *Security Information and Event Management* (SIEM, pronounced "seam"). The SIEM is designed to be a centralized component collecting data from all sources. In many cases, the primary inputs to a SIEM are log files. SIEMs can be configured to collect log files from hosts, IDSs, firewalls, and just about any other system on the network. Like an IDS, it is designed to extract features, run analyses, and report potential problems. Many security vendors have started to provide more advanced sensor systems called *Endpoint Detection and Response* (EDR) for hosts and *Network Detection and Response* (NDR) for networks. These systems have their own operations, but their log files are often usable as inputs into SIEMs as well.

Generally, SIEMs are very expensive, very complex systems that attempt to alleviate some of the burdens on human operators by organizing alerts by severity and priority. These systems are designed to engage human operators interactively, assisting them in their investigation of network events and intrusions.

Emerging in 2018, there is a new approach to system security that appears to be taking over some of the responsibilities of SIEMs. These systems, called *Extended*

Detection and Response (XDR), have a lot of overlap with SIEMs, but were built from a different design philosophy and have slightly different profiles. Like SIEMs, they gather data from a wide variety of sources, including EDR and NDR, and like SIEMs, they are designed to assist human operators conduct investigations into potential incidents.

One of the major differences is that SIEM technology was derived from older technology that processed log files more for compliance than for security. This has influenced SIEM development since inception. On the other hand, XDR was designed from the get-go for threat detection and incident response. Because of this, XDR is optimized for these operations. XDR is also more capable of working with sources beyond log files.

Another major difference is that XDR is typically a cloud-based offering, whereas SIEMs are usually on-prem.

Both SIEMs and XDR also offer “playbooks,” which are prepackaged responses to detected threats or incidents. These playbooks, though automated, generally interface with a human. The issues associated with automated responses are still just too great. This is sometimes called “human-in-the-loop.” These systems can also integrate with centralized servers that update all of the organization’s firewalls with new rules based on the outcome of the investigations.

For example, Palo Alto Networks offers playbooks as part of the “Cortex XDR by Palo Alto Networks Pack.” One of these playbooks is called “Cortex XDR incident handling v3.” The description of this playbook is

> This playbook is triggered by fetching a Palo Alto Networks Cortex XDR incident. The playbook syncs and updates new XDR alerts that construct the incident and triggers a sub-playbook to handle each alert by type. Then, the playbook performs enrichment on the incident’s indicators and hunts for

> related IOCs. Based on the severity, it lets the analyst decide whether to continue to the remediation stage or close the investigation as a false positive. After the remediation, if there are no new alerts, the playbook stops the alert sync and closes the XDR incident and investigation. For performing the bidirectional sync, the playbook uses the incoming and outgoing mirroring feature added in XSOAR version 6.0.0. After the Calculate Severity - Generic v2 sub-playbook's run, Cortex XSOAR will be treated as the single source of truth for the severity field, and it will sync only from Cortex XSOAR to XDR, so manual changes for the severity field in XDR will not update in the XSOAR incident. [1]

This playbook, as described earlier, is designed to semi-automate processing a cybersecurity incident. The playbook has some steps which are automated, such as trigger a subplaybook for each alert type. But it also has steps that integrate with the security operator: asking the analyst whether to continue or if there is a false positive identification. By capturing very common operations as a playbook, it is easier to handle these kinds of investigations across a large organization with a lot of nodes.

Infiltration, Exfiltration, and Advanced Persistent Threats

Although I have identified a few types of attacks in this chapter (e.g., DDOS), you might be wondering, *how does an attacker get through all of this security*? How do they get past the firewall, bypass segmentation and DMZ, avoid triggering the IPS/IDS, and then extract whatever data they're looking for? I wish the answer was "they don't."

Sadly, attackers do bypass defenses and do steal data on a far-too-regular basis.

In this section, I will identify how attackers get into a system (*infiltration*) and how they get data out of a system (*exfiltration*). Some attackers have been described as an *advanced persistent threat* (APT). Although the term has fallen out of use a little, I am going to use some of the old APT definitions and frameworks to talk about all kinds of intrusion. APTs were characterized by stringing together various attacks in order to achieve a final object. In talking about how they assemble these pieces, you will also get an idea of pieces that can be used by themselves.

An APT is characterized by characteristics such as *stealthy*, *patient*, and *sophisticated*. Stealth is important because the attacker may not get into the system on the network that they want even after they "break in." For example, as discussed in this chapter, security-conscious organizations will divide their networks into segments and enforce security policy (e.g., using a firewall) when moving between segments. Even if an attacker breached the DMZ, for example, that does not mean they have access to any of the internal systems. In fact, that is the *point* of having a DMZ.

But what if the attacker got in and did little other than *observe*? By not loudly rattling every door handle to see which rooms are unlocked, they are less likely to set off IPS and IDS detection. With a foothold into the system, just passive observing may reveal signs or clues of a vulnerable system beyond the DMZ internal firewall that is exploitable. Thus, APTs generally are willing to wait and be *patient*.

For the APT to get into a particular system, they may need to use rather sophisticated attacks. Perhaps they might even use a *zero-day* exploit. A zero-day exploit is one that is either unknown or so newly known that there are no patches or defenses available for it yet. In other words, it is an unmitigated vulnerability that is likely open and exploitable by the attacker to get unauthorized access. Attackers like this are either *sophisticated* or they are *renting sophistication*. By that I mean that in the modern, cloud-based Internet, sophisticated attackers often rent out services or software including *DDOS-as-a-service* and *Ransomware-as-a-service* [148, 188]. But one way or another, APTs typically use sophisticated techniques.

Although APTs attack each target uniquely based on both the posture of the defender and the personality of the attacker(s), there are some common characteristics of these kinds of operations. APT operations are typically described as having conceptual stages that capture these characteristics. Different vendors and researchers have proposed different formulations, but the following five-stage version, slightly modified from a survey of APT-related topics [37], is representative:

1. Reconnaissance
2. Establish foothold
3. Lateral movement
4. Exfiltration or other malice
5. Clean up and finalize

Although some of these stages clearly have a dependency on another, the sequence is not a linear one. For example, reconnaissance occurs throughout the entire operation. Moreover, as discussed already, these techniques are used by non-APTs as well. Each one is worth discussing as an individual technique.

Reconnaissance

Military planners and strategists have long known that reconnaissance is crucial to combat operations. Sun Tzu stated:

> Armies remain locked in a standoff for years to fight for victory on a single day, yet [generals] begrudge bestowing ranks and emoluments of one hundred pieces of gold [for spies] and therefore do no know the enemy’s situation. This is the ultimate inhumanity. [268]

The invading attackers in cyberspace have developed and practiced reconnaissance and become very good at it. Reconnaissance is essential to the APT invader at all stages. The first use of reconnaissance is before the intruder has infiltrated any part of the system. During this stage, the attacker may be looking for just about any kind of information including nontechnical information about the company. Business information, executives, offices, policies, products, rumors, and gossips are all helpful. Information like this may be useful in constructing a social engineering attack (see Chapter 1), guessing usernames and/or emails, or just in improving planning for what to do once passed the defenses.

After getting a first foothold in the system (discussed in the next section), reconnaissance will be necessary for looking around for offensive purposes (finding new targets within the system) and defensive purposes (protecting their foothold from detection or ejection). This will continue throughout the entire lifetime of the attack.

When (and if!) the attacker decides to exit the system, reconnaissance will continue especially if the attacker believes that they have been discovered to some degree or another. The attacker will want to observe the actions of the defenders as they try to find out what is going on in order to know how much time they have left, what the defenders have learned, and if they are hopefully (from the attacker’s point of view) missing some of the attacker’s entrypoints.

One big question for an attacker is what services are running on a computer and what are the characteristics of that service? As you learned in Chapter 7, attackers bypass defenses using vulnerabilities, and defenders *must* patch known vulnerabilities as soon as possible. Attackers want to know if a system is running vulnerable software or software with a vulnerable configuration. Accordingly, a big focus for attackers is to identify *all* network accessible services on *every* possible host system. And, they want to know as much about those services as possible. There are often *dozens*, if not *hundreds*, of services running across the collective hosts of an organization. Each host may have a slightly different configuration leading to *thousands* of distinct targets. The attacker often needs *just one* to be weak in order to bypass security controls to one degree or another.

For the defenders trying to patch and reinforce their systems, they often rely on large databases of known vulnerabilities. For example, the MITRE corporation created the Common Vulnerabilities and Exposures (CVE) program in 1999 [170]. The concept is that when somebody finds a vulnerability in software, it is reported to the CVE organization. Once verified, it will be issued a unique number. The CVE contains information on exactly what software is vulnerable and under what conditions. It also includes remediation information such as configuration changes or even patch data.

To illustrate, CVE-2021-44228, also known informally as *Log4Shell*, identifies a vulnerability that was (and unfortunately is) common in many systems on the Internet. It identifies some very serious issues with a library known as *Apache Log4j2*. This library is used in many Internet servers. According to the CVE, attackers that could control log messages could potentially insert and execute their own malicious code on systems with services such as LDAP, which is used for authentication, authorization, and access controls. The CVE also indicates that the vulnerable versions are 2.0-beta9 through 2.15.0. Version 2.16.0 and later have the dangerous functionality completely removed (and are, therefore, safe). The CVE includes a wide range of links to other resources describing the vulnerability in more detail, additional instructions for remediation, and so forth [19].

CVEs like this one are a great resource for defenders to be able to check their software and see if vulnerabilities have been reported and to repair them. Moreover, CVE-compatible products can help with automating the process. Compatible tools can automatically scan an organization's network, detect software with known vulnerabilities, and assist in the patching or remediation process. The good news is that defenders can find out what is wrong on their network.

The bad news is *attackers can find out what is wrong on the defender's network*. The attackers have access to all of the CVE data too. So, when an attacker goes to scan a network, if the attacker can identify if a system is running *Apache Log4j2* that is running version 2.15.0, they can know what kinds of attacks to launch against it.

Often, attackers would like to know what services are running on *other* systems besides the one they have access to. To find out what is running on another computer, they need to interrogate it. Basically, they want to ask all the other systems on the network, "what programs are you running?" They may not be able to figure out all of them, but there is a fairly simple way to figure out most, if not all, of the programs running that are *servers* (i.e., provide services over the network). A *port scan*, discussed earlier in the chapter, will do a good job of identifying which ports have a server listening on them.

Identifying a service running on a port is a good first start, but, as stated earlier, attackers would like to know *what* service is running on the port and, if possible, *what version and configuration* is in use. Just sending correctly formatted data to the TCP port is used to determine if there is any kind of server at all. But once it is known that a server is present, the attacker may launch various probes to that port to try and *fingerprint* the server. Many server programs, even if they cannot be completely accessed (e.g., require a login or other authentication data), can reveal quite a bit about themselves. Some server programs even helpfully identify which version they are running.

Story Time: Realistic Hacking in a Movie!
Most “hacking” in movies is hilariously bad. From viruses that scream in pain when they are being deleted to “fighting” with viruses in an elaborate video game, movies are not known for their technical accuracy in the cybersecurity realm.

One notable exception is *The Matrix Reloaded*. During the movie, the character known as Trinity, played by Carrie-Anne Moss, uses *nmap* to scan a network for vulnerable software. The *nmap* utility is real and is really used to scan ports. In the movie, *nmap* shows the fictional hacker that *SSH* is running on the computer with the address *10.2.2.2*. SSH is a secure access terminal that allows a remote connection to a computer over a text interface.

Although in the next bit, the movie depicts Trinity using a made-up program called *sshnuke*, it begins “[a]ttempting to exploit SSHv1 CRC32.” This is a reference to a real vulnerability discovered in 2001 in SSH. This vulnerability can enable an attacker to bypass the security login and get access to the remote machine without knowing a valid password. Much of what follows is also made-up, but it is still fun to see a more realistic portrayal of hacking in a movie [208]!

In addition to port scanning and fingerprinting, attackers may also passively listen for network traffic. Many system LANs have broadcast traffic, which means that an attacker can hear the data being sent between other machines. Even if broadcast traffic is not available, they can scan the traffic being sent to the machine they control. This can give them information about the other systems on the network and how they interact with the compromised host.

The attacker can also explore information available on the compromised machine itself. This may include log files, data files, and configuration files. If the system is running a web server secured by TLS, as discussed in Chapter 6, it *must* have access to a private key somewhere. The attacker may try to find and steal it. Or, the system may be storing passwords poorly. For example, some systems will be configured with a connection to an SQL database and may include a hard-coded password in a file. The exact amount of damage that an attacker can do to a system depends on their access level on that system.

And, of course, in addition to all of these “high-tech” mechanisms, the attacker can use social engineering attacks to request information. This is another common and effective form of reconnaissance.

It is also worth noting that one of the other purposes for reconnaissance *is to find the target data*. For example, an intruder might know that a hospital has the records of VIPs stored *somewhere*, but no idea where that is. Reconnaissance may help to reveal likely systems or networks that will have the desirable data.

Establish Foothold

Every intrusion has to start somewhere. In very many cases, an attacker can initially get access to a lower-priority system more easily than they could to the system they really want to exploit. For example, an attacker might get access to a company web server. This is not uncommon because often web servers are some of the only company systems that are directly accessible on the open Internet (i.e., the attacker can directly connect to it without having to go through email or some indirect route). The attacker will use their reconnaissance data about the visible server to find an attack. The ideal goal is to get some kind of remote shell access. This kind of access is general and allows them to launch programs, explore the system, or otherwise manipulate it. In other words, they do not compromise a web server to attack the website (probably). Instead, they want general access and general control to the machine in order to create a base of operations. They may achieve this using one of the vulnerabilities discussed in the previous section.

Of course, using a DMZ or other isolation techniques, defenders can try to shield the rest of their network from a compromised web server. But the attacker, if they are simply holding their position, may remain undetected for some time while they continue to conduct reconnaissance. Alternatively, the attacker, using phishing, may try to get a foothold on a user’s computer already inside the organization’s network. Although phishing is often associated with stealing information or money, it can be used to invade a network completely bypassing firewalls and DMZs (because email goes directly to people within the organization).

Even if the DMZ is bypassed, the attacker will want to expand to other systems, but this is the topic of the next section. More relevant to this section, the attackers will also want to improve their foothold on the compromised machine(s). For example, if the attacker compromised a properly configured web server, the compromise should not have given them administrator access. The attackers will almost certainly search for mechanisms to elevate their access to administrator. This will sometimes involve looking for other vulnerabilities on the system that permit *privilege escalation*. Or, it may involve compromising an administrator password through social engineering, brute-force password cracking, or testing for default passwords.

The attacker also will look to create *new* avenues for getting onto the system in case the current vulnerability is discovered and fixed or so that phishing is not required. This might involve installing new software that creates additional access methods (sometimes called a *backdoor*). The backdoor could be inserted as just a stand-alone program, or it could be created through the insertion of vulnerable software. If the machine already has remote access, the attacker could simply create new user accounts.

Another important part of the foothold is to make sure they will remain undetected. If the attacker gets administrative access, one of their objectives will be to delete any of their footprints and ensure their invisibility by corrupting the logging process. They may try to disable logging altogether or, in order to be less obvious, simply edit the log files to remove anything about their activities. For example, most systems log when there is an invalid password attempt. If the attacker manages to get administrator access using this approach, one of their next tasks would be to remove any reference in the log file to all the bad attempts.

Lateral Movement

In order to get “closer” to the desired data, the attacker will typically have to compromise additional systems after the entrypoint system. A system may be closer to another either in terms of network hops or in a semantic network.

Network hops simply refer to how many routers must be traversed to get to the machine. As discussed earlier, many networks are segmented. There is a router between each segment, and, because segments have different security profiles, the router is usually an internal firewall. An attacker may have to make lateral movements from one host to another across segments in order to close in on the true target. Some segments are considered higher security than others. Firewalls often mark the boundaries of these segments. An attacker may need to make a *lateral movement* to gain access to some other system on another segment. This often involves finding a legitimate path through the internal firewall.

A semantic network, on the other hand, represents how the various applications are connected to each other. Typically, for example, a web server is connected to a database. That very same database may be connected to another data entry or data processing system. The web server has no application-level connections to the data entry/processing system. The two machines may never communicate directly to each other even if they are on the same segment. But the web server connects to the database which connects to the data entry/processing system. The three systems are connected in a semantic network where the web server is one semantic “hop” away from the data entry/processing system. The attacker may be completely uninterested in the database by itself but is interested in it in order to gain access to the data entry/processing system.

A very common approach to make a lateral movement is stolen credentials. Compromising the password on one machine may result in knowing the password for the same user on other machines. Sometimes, credentials can be observed or determined from network traffic that the attacker passively observes.

Of course, the attacker can make a lateral movement using the same techniques used to get the initial foothold. Using reconnaissance and identified vulnerabilities, the attacker can compromise other machines on the same network.

Exfiltration or Other Malice

After patiently navigating the victim network, the attacker hopes to eventually be able to achieve their ultimate objective. A common objective is the exfiltration of data. Generally, the attacker will have some kind of Command and Control (C2C) server that can receive the data. If the victim organization has no egress filtering or scanning, this is a simple process that may go completely undetected.

Modern defenses should scan egress traffic. Even basic firewalls can be configured to block outbound connections to suspicious IP addresses. This is a very basic technique for blocking exfiltration but often a minimum first step.

More advanced defenses will attempt to monitor for unusual outbound connections. Large data transfers may be suspicious for many organizations, especially from certain systems. An even more sophisticated system might deploy *data loss prevention* (DLP) techniques. DLP technologies identify an organization’s most sensitive data, where this data should be stored, who or what application(s) should have access to this data, and how to protect from the loss of this sensitive data. DLP is almost by definition an egress scanning and filtration technology directed toward data life cycle management.

DLP was originally created for the purpose of identifying unintended misuse of data by an organization’s employee (e.g., copying confidential data onto a personal USB or external memory device). However, with the increase of data exfiltration by threats like APTs, they are also used to identify and block outside and inside malicious actors trying to access, steal, or destroy data.

A typical DLP deployment can be broken into three components: inspection visibility, inspection capability, and detection response. Inspection visibility is the means by which the DLP technology is able to inspect data for exfiltration. Three common inspection approaches are scanning data at rest, scanning data in motion on the network, and scanning data on endpoints.

Scanning data at rest for DLP works by scanning storage locations for known sensitive data. “These scanning platforms are generally directed towards network share locations, long term storage, database backups, or archive storage locations” [92]. This is helpful with exfiltration because the attacker may need to move the data around in order to get it to a location where it can be sent over the network.

Scanning data in motion is used to describe DLP inspection capabilities that analyze outbound data transfers on the network. Notably, there are versions of this technology that only inspect out-of-band (meaning the data is reviewed in parallel to its transmission) [92]. This obviously may only be able to detect exfiltration and not block it.

Scanning data on endpoints works by installing an agent on hosts. The DLP component can scan data saved to the host or data being transmitted to the host. This is obviously very comprehensive for a given host but must be installed on most, if not all, hosts in order to be effective [92].

Once the DLP has a visibility capability, it requires an inspection capability in order to determine if the data is out of policy-allowed locations and uses. There are various approaches to this process, but three such processes are exact data matching (EDM), indexed document matching

(IDM), and data string matching.” EDM is most often used for data stored in a database or some other structured format. EDM works by comparing data being scanned to the data in the database or structured storage. IDM, on the other hand, requires creating an index from unstructured (free-form) documents for use by the DLP technology when scanning. Because only part of a document may be exfiltrated, the IDM process typically breaks up sensitive components within a document as individual elements to be matched. Finally, data string matching looks for known patterns like 16-digit credit card numbers or 9-digit social security numbers [92].

The final DLP component is the responsive component that either raises alerts or blocks the exfiltration depending on the configuration [92].

Generally, however, if the attacker has reached this stage, they will probably succeed in getting the data out. A wise organization will put in as many of these technologies as possible to prevent it, but attackers can almost always find a way out. Previously, in Section , I discussed using DNS as a covert channel. As I explained in that section, attackers can get data out using DNS. There were at least half a dozen high-profile exfiltrations between 2014 and 2017 that used this exfiltration-over-DNS technique [33].

In addition to exfiltration of data, an attacker could do other kinds of mischief including data destruction, system disruption, or other malicious activities. The attacker could, for example, deploy malware. The attacker could have deployed other malware as part of any of the previous stages in order to better conduct reconnaissance, establish a foothold, or make a lateral movement. But the attacker may wish to deploy malware on a specific server for a specific purpose. For example, perhaps they want to lock up very important records using ransomware. Refer to Chapter 7 for some examples.

Clean Up and Finalize

Once the attacker has completed their objectives, they may perform a number of cleanup tasks. For example, if the defenders become aware of the adversary, they may be searching their systems for the intrusion. The attacker may spend some time cleaning up their tracks and hiding their presence. They may wish to remain in the system in a kind of "low profile" mode. This would require them removing many of their tools and accounts, forging false log files, and otherwise producing a minimum amount of network traffic. The goal would be to attract as little attention as possible while the defenders are searching for the intrusion. With any luck (for the attacker), the defenders will not be able to find the attacker or the vulnerabilities that got them into the system. Once things have "cooled down," the attacker can reassert themselves on the network.

On the other hand, if the defenders are still clueless and have no idea there is an intruder, the attacker can set to work strengthening their hold on the system, extracting additional data, and looking for new opportunities to spread.

Story Time: DigiNotar—Anatomy of an Intrusion

I already mentioned DigiNotar in Chapter 6. This Certificate Authority (CA) was infiltrated by an Iranian hacker. A CA is supposed to keep this kind of data carefully protected.

In terms of defenses, the DigiNotar network was configured with segments and DMZs. The critical servers with cryptographic secrets had those secrets secured by hardware security modules (HSMs). So far, so good. So how did the attacker bypass all of this security?

According to the investigation, the first system compromised was web servers in the external DMZ. These web servers were running outdated software. As discussed in this chapter, that is the quickest way to get compromised. Once the attacker had control of the machines, they turned these web servers into their own personal file server. These systems became the attacker’s go-between with the inside and outside of the organization. Data would be moved out of the sensitive inner segments to these servers where the data could be exfiltrated to the attacker’s computers.

The first lateral movement of the attacker was from the external DMZ to a segment called *Office-net*. It appears that there was a database on an Office-net computer that was accessible from the web server, and the web server had the username and password stored in a file that was readable by the attacker. Using the username and password, the attacker accessed the Office-net computer to get foothold outside of the DMZ. The attacker found other credentials using brute-force cracking of password hashes. Because the passwords were weak, they were easily broken by this brute-force attack.

Using these and additional techniques, the attacker made a subsequent lateral move into the *Secure-net* segment that had the critical cryptographic secrets. These computers were not even supposed to be directly connected to the Internet, but apparently the attacker created some special tools to create a tunnel connection that bridged the gap. The attacker had full remote desktop capabilities on the compromised machines. Using these machines, the attacker was able to issue at least 531 rogue (fake) certificates. There may have been more.

One reason why it is not known how many certificates were created is because the attacker was editing log files. The complete record is destroyed and impossible to completely recover.

This story illustrates many of the concepts discussed in this section as they played out in real life.

Summary

Network security has historically been defined by perimeter security—protecting the resources inside a network from both threats attempting to infiltrate from outside and from threats already inside now attempting to propagate and cause harm. There are many reasons this is now changing. For one thing, it is hard to identify the “inside” anymore with the rapid shift to the cloud. With that said, understanding the basic principles and classical approaches is an important foundation.

Much of perimeter security is defined by firewalls. Firewalls sit at the interface between two networks and regulate traffic between them. One of the most basic approaches is filtering traffic based on the source and destination of traffic, but modern firewalls can analyze different aspects of the traffic up to and including the application data, the so-called layer-7 data.

Proxies of various kinds enable additional security on traffic entering and leaving a network, such as authorization, scanning, load balancing, and data loss prevention. Virtual Private Networks (VPNs) enable connecting remote networks to each other or securely granting access to protected resources to remote computers.

These components can be used to create a defensive network architecture. Firewalls are placed between an organization’s systems and the outside Internet. But they are also used internally to divide the network into segments that can be used to create different security zones. A demilitarized zone (DMZ) is a special kind of segment that separates out an organization’s public-facing systems from all other systems. That way, if one of the public systems is compromised, the attacker does not have direct access to the other systems.

If threats manage to bypass the network perimeter defenses, technologies like Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) allow for the detection of active threats and the prevention of attempted threats. One of the most important intrusion detection technologies in use today is honeypots, defensive deception technologies that are used to identify, confuse, and disrupt intruders.

Despite all of these defenses, attackers do manage to get through. A very nasty type of attacker is an advanced persistent threat (APT). These attackers are stealthy, patient, and sophisticated. They will carefully get a foothold in a network and then methodically expand their control until they reach the systems that have the data they are looking for.

Further Reading

Readers should remember that the solutions described in this chapter do not solve any problems on their own. These are *mechanisms* for enforcing policy. Unfortunately, many organizations just throw technology and architectures at problems and hope that it keeps the adversaries out. The proper starting point is *policy*. Although I have already cited Anderson's chapters on policy in Chapter 3, now that you have read a mechanism-heavy chapter I suggest going back and rereading Chapters 1 and 9 to understand how policy and mechanism fit together [40, Chapter 1], [40, Chapter 9]. I also like Peterson and Davie's book *Computer Networks*. This book is deeply technical and digs into how the Internet works from top to bottom and also includes a chapter specifically about security [205, Chapter 8].

Another good starting point for thinking about how to actually deploy network security components is to refer to NIST's guide on risk management. There is never perfect security that can prevent all possible intrusions. Instead, it is important to investigate risk to determine what the risks are and how they should be handled [109]. After reading this document, I suggest reading NIST's guide to security and privacy controls. The controls that it lists go beyond what you have learned in this book but should have a number of familiar elements such as authentication, authorization, host security, and network security [121].

NIST also provides an entire cybersecurity framework. You can visit their website at www.nist.gov/cyberframework. There is a lot on this website, however, which is why I recommend starting with risk management and security controls. Those two are a great start for a more thorough study of developing a full cybersecurity strategy.

Digging into firewalls can be fun. The colorfully named *Best Damn Firewall Book Period* is, in fact, quite comprehensive [241]. A more up-to-date book is *Network Security, Firewalls, and VPNs*. This book explicitly tries to tackle APTs and incorporate their methods into creating appropriate defenses [252]. At a slightly broader granularity, Bishop provides a full chapter on "Network Security," including a discussion of network organization, DMZs, and firewalls [60, Chapter 28].

Another area of important reading is *zero trust*. I have mentioned it repeatedly as the topic I will not discuss in this book. Zero trust would need several chapters to really provide sufficient background, principles, and applications. This book, instead, focuses on the classics. If you wrap your head around perimeter security concepts, you are ready to move on to all the reason *perimeter security is bad* and how zero trust attempts to fix the problem. Google's

BeyondCorp is reasonably understandable and a good place to start [58, 199, 279]. NIST also provides an architecture document for zero trust that is also helpful [222].

To further investigate the fascinating topic of defensive (and offensive) deception, I suggest the book *Introduction to Cyberdeception*, of which I recommended a single chapter in Chapter 1. The entire book is a great read, however, discussing theoretical principles as well as practical applications for all kinds of deception [225].

References

1. Cortex XDR incident handling v3.

14. Active deception to combat advanced threats. Technical report, Attivo Networks, 2019.

19. CVE-2021-44228. 11 2021.

21. Packet flow sequence in PAN-OS. 06 2021.

33. Ahmed, J., H.H. Gharakheili, Q. Raza, C. Russell, and V. Sivaraman. 2019. Real-time detection of DNS exfiltration and tunneling from enterprise networks. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 649–653.

37. Alshamrani, A., S. Myneni, A. Chowdhary, and D. Huang. 2019. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys and Tutorials* 21(2): 1851–1877.
[Crossref]

38. Alessandro Bulgarelli, M.A., and M.C. Francesca Mazzoni. 2005. Honeyspam: Honeypots fighting spam at the source. In *Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI'05)*, Cambridge, MA, ed. by D. Katabi and B. Krishnamurthy. USENIX Association.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing.
[Crossref]

53. Bellovin, S.M., and W.R. Cheswick. 1994. Network firewalls. *IEEE Communications Magazine* 32(9): 50–57.
[Crossref][zbMATH]

58. Beyer, B.A.E., C.M. Beske, J. Peck, and M. Saltonstall. 2017. Migrating to beyondcorp: Maintaining productivity while improving security. *Login* **42**(2). ISSN 1044-6397.

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

64. Braden, R., and J. Postel. 1987. Requirements for internet gateways (1009).

71. Chatel, M. 1996. Classical versus transparent IP proxies (1919).

73. Cheswick, W.R., S.M. Bellovin, and A.D. Rubin. 2003. *Firewalls and Internet Security*, 2nd ed. Addison Wesley Professional.

77. Cohen, F. 2004. The use of deception techniques: Honeypots and decoys.

92. Devlin, R. 2016. Data loss prevention—Devlin. Technical report, SANS Institute.

100. Economy, E.C. 2018. The great firewall of China: Xi Jinping's internet shutdown. *The Guardian*

107. Findley, N. 1993. *Shadowplay*. Penguin Group.

109. Force, J.T. 2018. Risk management framework for information systems and organizations. Special Publication (NIST SP) 800-37r2, National Institute of Standards and Technology, Gaithersburg.

111. Garber, M. 2014. There are 64 tiananmen terms censored on China's internet today. *The Atlantic*.

121. Group, J.T.F.T.I.I.W. 2020. Security and privacy controls for federal information systems and organizations. Special Publication (NIST SP) 800-53r5, National Institute of Standards and Technology, Gaithersburg.

131. Hernacki, B., J. Bennett, and T. Lofgren. 2004. Symantec deception server experience with a commercial deception system. In *Recent Advances in Intrusion Detection*, ed. E. Jonsson, A. Valdes, and M. Almgren, 188–202. Berlin/Heidelberg: Springer.
[Crossref]

148. Karami, M., and D. McCoy. 2013. Understanding the emerging threat of DDoS-as-a-Service. In *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 13)*, Washington, DC. USENIX Association.

164. Luo, T., Z. Xu, X. Jin, Y. Jia, and X. Ouyang. 2017. Iotcandyjar: Towards an intelligent-interaction honeypot for IoT devices, 1–11.

167.

Luotonen, A., and K. Altis. 1994. World-wide web proxies. *Computer Networks and ISDN Systems* 27(2): 147–154. Selected Papers of the First World-Wide Web Conference.

169. Mairh, A., D. Barik, K. Verma, and D. Jena. 2011. Honeypot in network security: A survey. In *Proceedings of the 2011 International Conference on Communication, Computing and Security (ICCCS'11)*, New York, 600–605. Association for Computing Machinery.

170. Mann, D.E., and S.M. Christey. 1999. Towards a common enumeration of vulnerabilities. In *2nd Workshop on Research with Security Vulnerability Databases*, West Lafayette.

177. McRae, C.M., and R.B. Vaughn. 2007. Phighting the phisher: Using web bugs and honeytokens to investigate the source of phishing attacks. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 270c.

178. Meadows, C. 1995. Applying the dependability paradigm to computer security. In *Proceedings of 1995 New Security Paradigms Workshop*, 75–79.

188. Moussaileb, R., B. Bouget, A. Palisse, H. Le Bouder, N. Cuppens, and J.-L. Lanet. 2018. Ransomware's early mitigation mechanisms. In *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES 2018)*, New York. Association for Computing Machinery.

189. Mphago, B., O. Bagwasi, B. Phofuetsile, and H. Hlomani. 2015. Deception in dynamic web application honeypots: Case of Glastopf. In *Proceedings of the International Conference on Security and Management (SAM'15)*, Las Vegas, ed. by K. Daimi and H.R. Arabnia.

199. Osborn, B., J. McWilliams, B. Beyer, and M. Saltonstall. 2016. Beyondcorp: Design to deployment at Google. *;login:* 41: 28–34.

203. Park, Y., and S.J. Stolfo. 2012. Software decoys for insider threat. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*, New York, 93–94. Association for Computing Machinery.

205. Peterson, L.L., and B.S. Davie. 2021. *Computer Networks*, 6th ed. Morgan Kaufmann.
[zbMATH]

207. Pouget, F., M. Dacier, and H. Debar. 2003. White paper: Honeypot, honeynet, honeytoken: Terminological issues. Technical Report RR-03-081, Eurecom.

208.

Poulsen, K. 2003. Matrix sequel has hacker cred. *The Register*.

210. Provos, N. 2004 A virtual honeypot framework. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego. USENIX Association.

213. Ranum, M.J. 1994. Thinking about firewalls. In *Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II)*.

222. Rose, S., O. Borchert, S. Mitchell, and S. Connelly. 2020. Zero trust architecture. Special Publication (NIST SP) 800-207, National Institute of Standards and Technology, Gaithersburg.

225. Rowe, N.C., and J. Rrushi. 2016. *Introduction to Cyberdeception*, 1 ed. Springer International Publishing Switzerland. [Crossref]

233. Scarfone, K., and P. Mell. 2007. Guide to intrusion detection and prevention systems (IDPS). Special Publication (NIST SP) 800-94, National Institute of Standards and Technology, Gaithersburg.

241. Shinder, T.W. 2008. *The Best Damn Firewall Book Period*, 2nd ed. Syngress.

248. Spitzner, L. 2003. Honeypots: Catching the insider threat. In *19th Annual Computer Security Applications Conference, 2003*, 170–179. IEEE.

249. Spitzner, L. 2003. Honeytokens: The other honeypot.

252. Stewart, J.M., and D. Kinsey. 2020. *Network Security, Firewalls, and VPNs*, 3rd ed. Jones & Bartlett Learning.

253. Stoll, C. 1989. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. New York: Doubleday.

256. Suljkanovic, S. 2005. Honeypots or honey delusions. Technical report, SANS Institute.

268. Tzu, S. 2002. *Sun Tzu: Art of War*. Trans. Ralph D. Sawyer. Basic Books.

279. Ward, R., and B. Beyer. 2014. Beyondcorp: A new approach to enterprise security. *;login:* 39(6): 6–11.

286. Yang, S. 2022. As China shuts out the world, internet access from abroad gets harder too. *LA Times*.

Footnotes

1 This includes time setting up, configuring, and deploying the network architecture and security policy enforcement mechanisms. It also includes time monitoring the network for intrusion, audits to ensure changes remain compliant, and reconfiguring based on changing IT and security needs.

2 In a historic irony, the view that network security is a bad idea and the only solution is secure hosts has returned in at least some form with the advent of *zero trust security*. Zero trust security is not considered in this book, which focuses on more classical security technologies. However, suffice it to say that in this zero trust model, the internal network is no longer trusted [279]. Unfortunately, most classical network security, the subject of this chapter, is quickly becoming obsolete. You have to start with the classic tech first, however, in order to understand what is changing and where computer security is going.

3 These devices are, in fact, still implemented on general-purpose computer technologies, but the systems are (supposed to be) stripped down to only include components necessary for the security operations.

4 Again, Appendix C provides an overview of the differences between UDP and TCP. Those differences are not particularly important here; the point is that a firewall that examines layer-4 data can tell the difference between layer-4 protocols.

5 This is a specially reserved port number that should almost never be used but can return results in some circumstances.

6 Please note, I am going to cite other papers and sources more heavily in this section than I have elsewhere. I am providing only a relatively brief overview of a very important technology. I figured it would be better to include a wide range of reference material here for those that want to investigate other sources instead of filling the “Further Reading” section with all of them.

7 I will refer to this as Martha’s idea, not Stoll’s. Most references that I read do not properly attribute the idea to her. For example, see Bishop’s book on computer security [60, Chapter 27].

8 I have very slightly edited this quote from the book for clarity.

9 Again, you rarely see attribution when Stoll’s work is discussed.

10 Each of these definitions comes from a different source. Refer to the survey [207] for attribution.

9. World Wide Web Security

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
The Internet and the World Wide Web have specific security needs and challenges, many of which relate to how applications are built on top of the original stateless HTTP protocol. We will examine these issues and the many solutions that address them.

Key Concepts

1. The Web was originally designed for sharing documents and linking them together semantically.
2. Additional systems have been built on those foundations, such as encrypted communications (TLS and HTTPS) and web applications.
3. These developments have required new technologies, like cookies for state persistence, JavaScript for interactivity, and OAuth for authorization.
4. Security technologies have been developed to make these advances secure and reliable and to address the additional vulnerabilities they introduce.

Common Pitfalls and Misunderstandings

1. HTTP is a stateless protocol, but modern web applications rely on keeping the user’s state between individual requests. Even though these needs are not accounted for in the original protocol, technologies have been developed on top of HTTP to make them possible.
2. The Internet refers to the interconnection of network resources around the globe. The Web refers to the semantic linking of these resources in useful ways that enable higher-level constructions like web search, applications, and human- and machine-readable context.

Useful Vocabulary

- **URL**: Uniform Resource Locator. A format for describing how to access a document or resource on the Internet
- **HTTP**: HyperText Transfer Protocol. A protocol for requesting and transmitting documents over the Internet
- **HTML**: HyperText Markup Language. A language for describing the structure and contents of a web page
- **JavaScript**: A programming language for building interactive applications within HTML pages
- **API**: Application programming interface. A collection of operations for requesting and receiving data from a web application
- **SQL**: Structured Query Language. A language for reading and writing data in a database
- **WAF**: Web Application Firewall. A filtering system for protecting web servers from malicious inputs

- **Regular Expression**: A syntax for matching patterns in text
- **SSO**: Single-Sign On. A system allowing a user to authenticate once and subsequently receive access to various other systems automatically

---

In the previous chapter, I introduced you to some of the concepts of what I call "classical" network security. Primarily, these technologies are focused on protecting the Local Area Network, or LAN. In many ways, this kind of security is *perimeter* security. The goal is for an organization to defend itself from "outside" threats. As I indicated, perimeter security, though still widely used, is already seen as outdated and obsolete. The cloud, bring-your-own-device (BYOD), mobile use, working from home, and other changes render perimeter thinking less effective, or perhaps not effective at all. This book does, however, provide a good foundation for understanding those components.

In this chapter, I will dig into the security components that have developed in order to secure the World Wide Web. The topics in this chapter are both still relevant *and* provide good foundation for more advanced topics including zero trust. Web security is a good topic to discuss after perimeter security for a couple of reasons. First, web security is a complementary concept to perimeter security. Instead of isolating, web security is about securely collaborating, interacting, sharing, and participating. Second, web security operates primarily at the application level of the network protocol stack. The security typically needs to be *end to end*, meaning from one endpoint to another rather than just at the boundary of a network.

To get started, some background will be useful. One of the most common misconceptions is that the Internet and the World Wide Web are the same thing. As discussed in Chapter 8, an "internet" is a network composed of smaller networks that are connected together. The Internet (with a capital "I") refers to *the* singular Internet, the worldwide interconnections of networks. The Internet was born, from a certain point of view, in 1969 when Stanford and UCLA created a network connection between their two networks.

The World Wide Web, on the other hand, is a suite of interconnected, but distributed, applications built on top of the Internet using a common set of protocols, primarily a protocol named *HTTP* (HyperText Transfer Protocol). The Web is not centrally planned or coordinated. Instead, it is a cooperative construct created by all the individual websites spread throughout the Internet. It is the whole ecosystem of search engines, ecommerce technologies, web applications, and other services. Cloud services such as AWS and Azure, mobile apps, and social media are all examples of technologies built within this environment.

An Overview of Basic Web Components

The architecture of what became the World Wide Web was proposed in 1989. According to Tim Berners-Lee, one of the people behind its development, the primary purpose of the Web was to "be a shared information space through which people and machines could communicate." By 1994, "browsers" started to become more widely available, and the Web began to really grow [56].

Resources and URLs

Data in the World Wide Web is organized into named *resources*. Each resource has a specific identifier known as a *Uniform Resource Identifier* (URI), sometimes more commonly called a *Uniform Resource Location* (URL).[1] A URL has a defined format:

```
<scheme>://<authority><path>?<query>
```

There are four elements in the URL:

1. **Scheme** defines the identification scheme. In practice, this is usually the protocol for requesting the resource. This is almost always http or https.
2. **Authority** is almost always the server where the resource is hosted. The host could be an IP address, but is most often a host name, such as “google.com”.
3. **Path** identifies to the server the specific location or other identification information of the resource within the server.
4. **Query** enables the request to be parameterized. This means the request can change based on the query provided.

So, for example, here is a very simple URL:

```
google.com
```

This URL does not specify the scheme, path, or query. When schemes are not specified, it is figured out from context and is usually http or https. When no path is specified, a default, or root, path is used. Query is only used for queries that can have parameters. An example with all four elements is shown as follows:

```
https://www.google.com/search?q=cat+dog
```

This URL identifies a Google resource for searching for cats and dogs. The path, in this case, is “/search” and the query is “q=cat+dog”.

The data identified by a URL does not have to take any particular form. It could be a web page, a file to download, an image, an action (e.g., signing out), or even data in formats meant more for machines than people. For this reason, data identified by a URL is called a *resource*. Notice also that a query is considered part of a resource’s identifier. So the preceding example for searching cats and dogs is a *unique resource* separate from a search for something else or even the initial Google search page.

URLs enable a crucial concept of the Web: *semantic* connections between resources. The Internet, of course, links network resources. On LANs, resources share a common medium for communication. Over the Internet, routers and protocols enable logical connections between any two endpoints. This is absolutely necessary. The problem is, however, that none of these connections have any meaning or purpose associated with them. There is no indication of why someone would use a connection or what the result will be. In other words, there are no semantics.

The Web, on the other hand, introduces a level of linking higher in abstraction. Resources, like web pages, are able to link to other web resources using *hyperlinks*. A hyperlink, or just *web link* or *link*, is designed to connect two web resources together semantically. A link includes a URL to another resource, linking the current resource and the linked resource together. There was no need to create network links, logical or otherwise, or provide routing and a protocol stack in order to link the resources. Instead, resources, like web pages, can simply include a hyperlink, and the connection is created.

Moreover, the connection is semantic. The relationship between the two resources is not created because they are in the same LAN or because they have some kind of network connection. Instead, the web resources are meant to be connected because they have a relationship in the information, data, or other concepts they represent. This is enhanced further by free-form text that can be associated with web links on web pages. This text can communicate human-readable or even machine-readable information about what the link is (e.g., “pictures of my kids!”) and/or what the link does (e.g., “click to sign out”). Using these kinds of links, web pages are connected together by ideas and concepts into a mesh that is like...well, like a *web*.

So, the Internet can be thought of as interconnected computers and computer processes, and the Web can be thought of as interconnected ideas and information.

The nature of the Web also enables *resource discovery* by *browsing*. From one link to another, a person (and even machines!) can “crawl” the Web, discovering more and more resources. Even search engines such as Google answer search requests by using massive databases of indices created by crawling the Web and adding data about each resource.

Web browsers are programs that make the Web accessible to humans. Browsers take the raw information from the Internet and convert it into the visual representation the human sees on the display. This process is called *rendering*.

HTML

When a browser renders a website or other web content, there are a lot of components that go into the process. The starting point for most of these web pages is HTML: HyperText Markup Language. A “markup” language is a text-based description system wherein special control codes or other “markup” data indicate how the text should be rendered when displayed.

To illustrate, here is a very simple example of HTML:

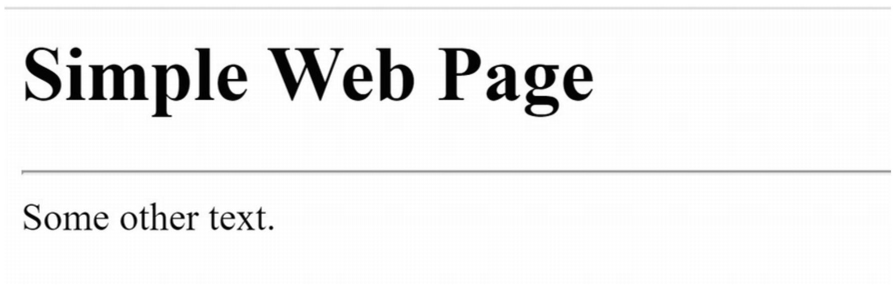
```
<HTML>
```

```
<BODY>
<H1>Simple Web Page</H1>
<HR>
Some other text.
</BODY>
</HTML>
```

When this data is loaded into a web browser, it is *rendered*. Each browser might render the information slightly differently, but it will look consistent for the most part. One rendering is shown in Figure 9-1. You can see that "Simple Web Page" is quite large and in a bold type, while "Some other text" is smaller and is not bold.

In the HTML, "Simple Web Page" is marked with a preceding " $<$ H1 $>$ " and followed by a " $<$ /H1 $>$ ". In HTML, these are called "tags." The H1 tag indicates that the text enclosed between the H1 tag and the closing H1 tag (" $<$ /H1 $>$ ") should be rendered as a level 1 header. The rendering in Figure 9-1 is a default rendering of a level 1 heading for this browser. The exact font, size, and other attributes for the H1 heading can be customized using, for example, style sheets.

Some tags do not have a closing tag. The " $<$ HR $>$ " tag, for example, does not. This tells the browser to put in a "horizontal rule," which is why the rendered page has a line between the two text sequences.


Simple Web Page

Some other text.

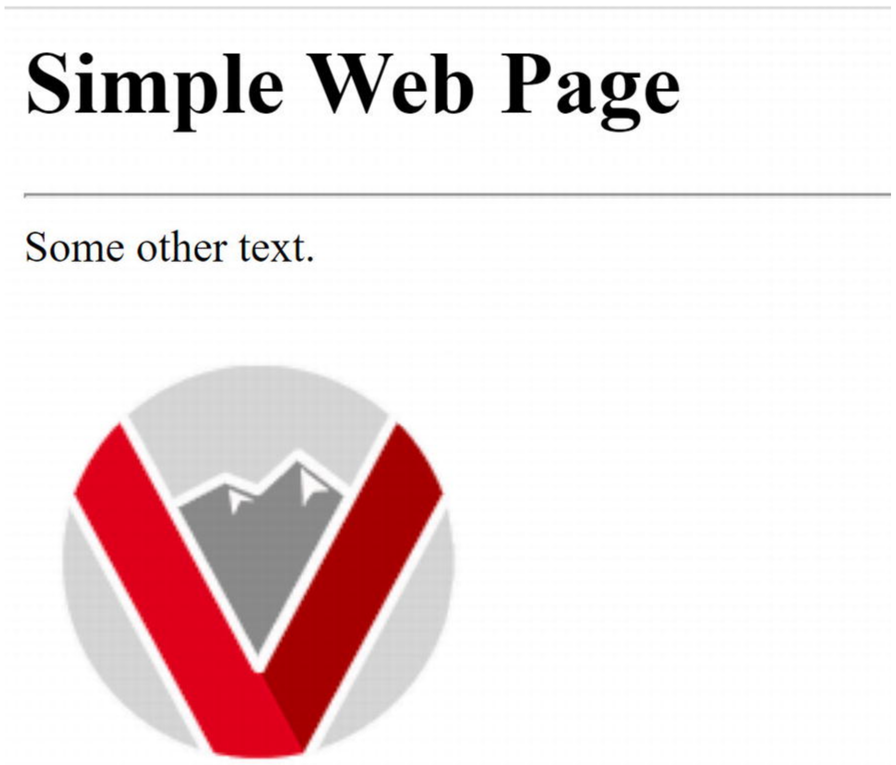
Figure 9-1 A very simple web page

Not only are the contents of the web page rendered in Figure 9-1 simple, but so is the structure. This web page is rendered based on the contents of this HTML all by itself. But most web pages have to identify additional resources at other URLs that the browser must also download as part of the rendering process. For example, an image is generally included in this way. In the following listing, a new “IMG” tag has been added:

```
<HTML>
<BODY>
<H1>Simple Web Page</H1>
<HR>
Some other text.
<P>
<IMG SRC= "https://www.crimsonvista.com/img/logos/CV_icon.png">
</BODY>
</HTML>
```

The IMG tag tells the browser to insert an image into the web page. The “SRC” parameter tells the browser where to go to find the image. The location is specified as a URL. Figure 9-2 shows a rendering of this page.

The image displayed is a logo for my consulting company, Crimson Vista. Importantly, the image is hosted at a different domain than the HTML. In other words, the browser is getting the HTML listed earlier from one web server and the image *referenced* in the HTML from another. Most web browsing works this way. Usually, a person navigates to a website URL which downloads initial HTML to the browser. The browser then begins rendering the browser while simultaneously downloading any referenced content. This might be images, programming instructions, style sheets, other HTML, or most other types of web content. In short, rendering a single web page almost always means making multiple requests and often from multiple servers.


Simple Web Page

Some other text.

Figure 9-2 Another simple web page with an image in it

Story Time: Personal Web Pages

On a bit of a personal note, I started my undergraduate degree at Brigham Young University in 1994. This means I hit college just as the World Wide Web was taking off. During these early days, you could watch the rapid development of this new technology. Browsers improved, the Java programming language was released, and both personal and commercial websites emerged.

As a college student in computer science, I could create my own web page and host it with the university. I started learning a little HTML and some other basic elements. Certainly, the web page I created at the time was clunky, awkward, and not aesthetically pleasing. But it was an amazing experience to be able to create a “home page” that was visible to anyone else using the World Wide Web. This was before the advent of social media; so those with a web presence would sometimes use their home page as a place to share about themselves.

For example, during my senior year in 1999 (I had taken time off from 1997–1999 for religious missionary service), I became engaged to the brilliant and beautiful Amy Nicole Quist. I learned just a little bit of web scripting and was able to create a countdown timer that would automatically display the days, hours, and minutes left until the wedding. I wanted the whole world to know how much I was looking forward to tying the knot with this lovely young lady. Speaking of which, we were married on April 20, 2000. We are still happily married 23 years later!

The protocol used for making these requests and getting the responses is called *HTTP*.

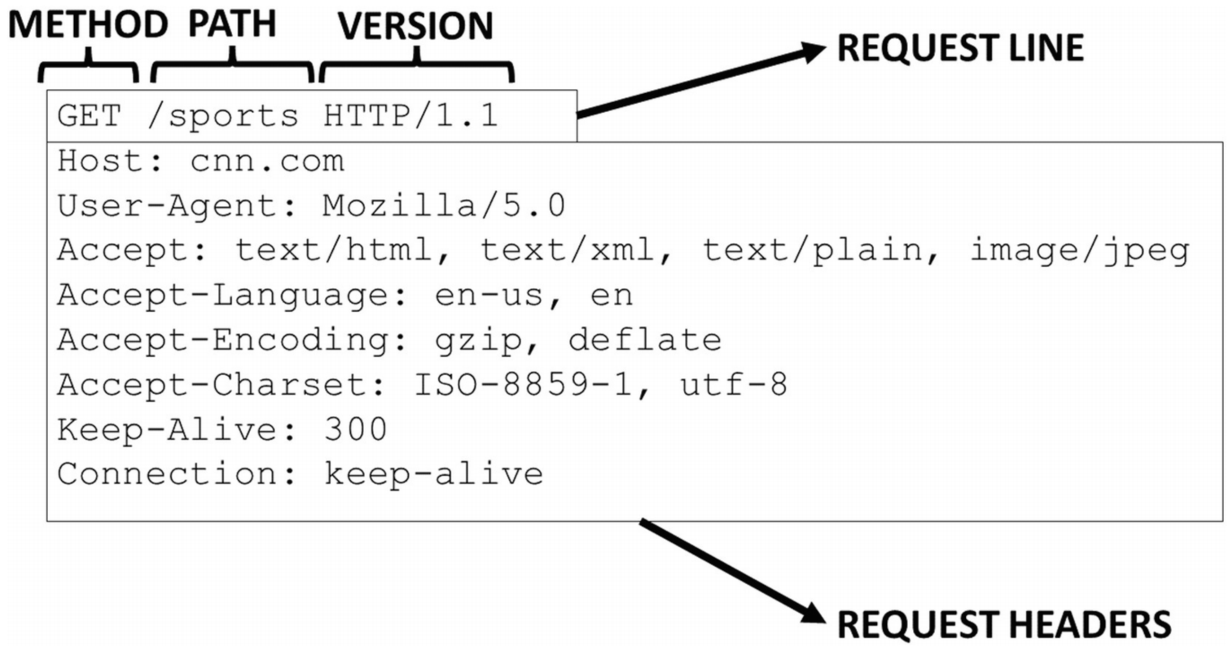
Overview of HTTP

I briefly mentioned HTTP in Chapter 8. In that chapter, I was talking about firewalls and how firewalls can examine different parts of the network stack. HTTP was an example of a layer-7, or application layer, protocol. In this chapter, I will explain how this protocol works and how it is used to exchange information on the Internet.[2]

Any computer program can use HTTP, but the most common example of a program that does so is your average web browser. When a browser is directed, either because a user typed a URL into the URL bar or because a hyperlink was clicked, the browser sends an *HTTP request* to the host identified in the URL. There are different types of HTTP requests. Figure 9-3 from Appendix C is an example of an HTTP *GET* request.

```
GET /sports HTTP/1.1
Host: cnn.com
User-Agent: Mozilla/5.0
Accept: text/html, text/xml, text/plain, image/jpeg
Accept-Language: en-us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
```

Figure 9-3 An HTTP GET request is commonly sent by browsers to get a web page



```
GET /sports HTTP/1.1
Host: cnn.com
User-Agent: Mozilla/5.0
Accept: text/html, text/xml, text/plain, image/jpeg
Accept-Language: en-us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
```

Figure 9-4 The first line of an HTTP request is called the “request line.” Within the request line is a method, a path, and a version

HTTP requests have different parts that are called *the headers* and *the body*. Figure 9-4 shows an example of this. The first line of the HTTP request is called the “request line," and it has three components: the method, the path, and the version. The remainder is the request headers. This HTTP request has no body.

The request method tells the web server what kind of communication the request is. A *GET* request, such as the requests you have seen so far, is probably the most common type. It indicates that the browser is specifically requesting, *and not sending or uploading*, information. Any time you have popped a URL into a browser and hit enter, the browser has almost certainly sent a GET request. As shown in the figure, the text sent is literally the method type, so in this case it is the actual word "GET."

There are other types of HTTP requests including requests that upload data. The most common HTTP method for upload is called a *POST* method. As shown in Figure 9-5, this request looks almost the same. The "GET" text has been replaced with "POST" text in the method location of the request line, and there is a new header for "Content-Length." This new request header is required so that the web server knows how much data is being uploaded. If the HTTP request did not have a "Content-Length" header, the server would not be able to tell when the upload was complete.

```
POST /homework/submit HTTP/1.1
Host: utexas.edu
User-Agent: Mozilla/5.0
Accept: text/html, text/xml, text/plain, image/jpeg
Accept-Language: en-us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
Content-Length: 2000

<2000 bytes of data>
```

Figure 9-5 An HTTP POST request

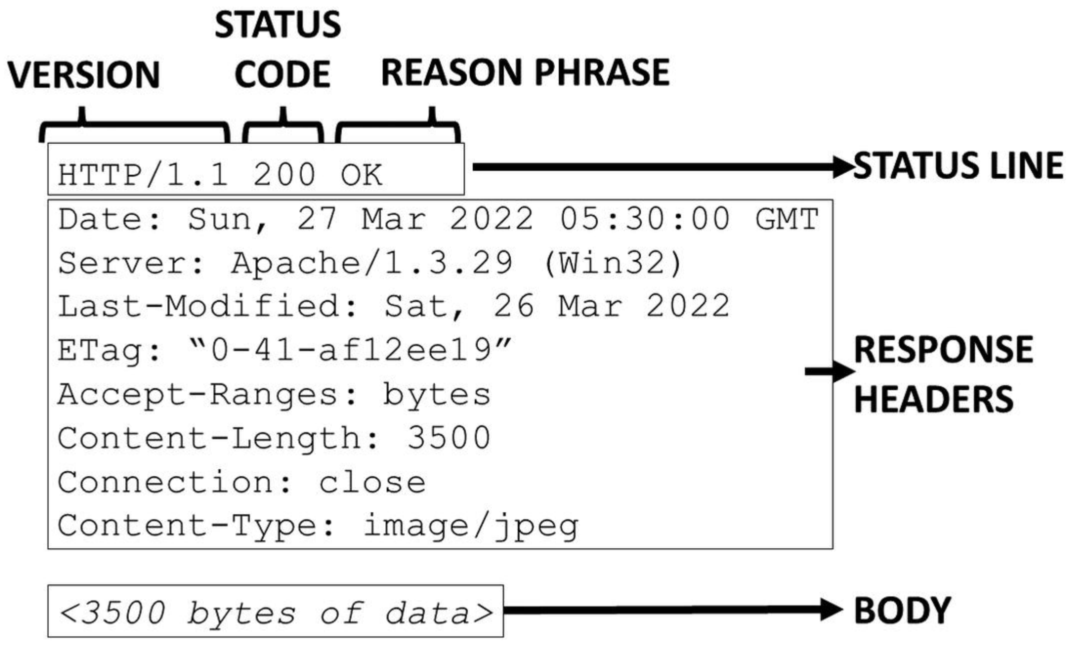
```
HTTP/1.1 200 OK
Date: Sun, 27 Mar 2022 05:30:00 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 26 Mar 2022
ETag: “0-41-af12ee19”
Accept-Ranges: bytes
Content-Length: 3500
Connection: close
Content-Type: image/jpeg

<3500 bytes of data>
```

Figure 9-6 An HTTP response indicating that the resource was located

There are other request methods including HEAD, PUT, DELETE, and others. GET and POST are the most common for “ normal” web browsing. The other methods are more common in machine-to-machine communications. Although beyond the topics covered in this book, REST APIs, for example, make use of the different request methods and define semantics for each one.

As discussed in Appendix C, when the web server receives an HTTP request, it is expected to send back some kind of HTTP response. The response includes a code that indicates some kind of response code. The response code “200 OK” is the standard code for reporting that the resource was found and is included in the response message. An example of this response is shown in Figure 9-6.



```
HTTP/1.1 200 OK
Date: Sun, 27 Mar 2022 05:30:00 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 26 Mar 2022
ETag: "0-41-af12ee19"
Accept-Ranges: bytes
Content-Length: 3500
Connection: close
Content-Type: image/jpeg

<3500 bytes of data>
```

Figure 9-7 The first line of an HTTP response is called the “status line.” Within the status line is a version, a status code, and a response phrase

As with the HTTP request, it is worth going into just a little more detail. As shown in Figure 9-7, an HTTP response has a “status line” that is analogous to the “request line” in an HTTP request. The status line has an HTTP version followed by a status code and response phrase pair. The status code indicates the result of the request, and the response phrase provides a human-readable string for explaining what happened to a user.

Some of the more common HTTP status codes and corresponding response phrases are

- **200 OK**: The resource was found.
- **301 PERMANENT REDIRECT**: The resource found with this URL has permanently changed to a new URL. The browser will usually refresh automatically to the new URL and will cache the change.
- **302 TEMPORARY REDIRECT**: The resource is only temporarily relocated. The browser will usually refresh automatically to the new URL, but will not cache the

change. That means the next time the browser tries to access this resource, it will still use the old URL.
- **404 NOT FOUND**: The resource could not be found.
- **410 GONE**: The resource used to be here, but is permanently shut down.
- **500 INTERNAL SERVER ERROR**: The server did something wrong.
- **503 SERVICE UNAVAILABLE**: The server currently cannot response.

Server response codes follow a schema wherein all codes in the 100s are informational, 200s are success messages, and 300s are redirections (meaning the browser should be sent to a different URL or web address). In terms of errors, codes in the 400s are meant to capture client-side errors (i.e., errors from the browser such as going to a nonexistent URL), and the 500s are server-side errors. Server-side errors come from misconfigured servers, servers experiencing outages, and so forth.

Story Time: The Mythical Birth of 404 Not Found

Humanity has always loved legends and mythology. Everything *must* have come from somewhere, and behind every common experience, we *want* to find a story behind it. The World Wide Web is no different in this regard. Myths and legends abound! One particularly interesting legend surrounds the origin of the error message “404 Page Not Found.” It goes something like this.

Much of the development of the World Wide Web came out of work done by scientists of CERN in Switzerland. In many ways, this was the birthplace of the Web. During the 1980s, these scientists worked together in an office building and stored a central database on the fourth floor in a room numbered, you guessed it, 404. This central database practically *was* the early Web. As requests came in from around the world for documents, they inevitably had errors, either requesting documents that did not exist or not formatted correctly. Eventually, the scientists worked up a standard error message: “Room 404: file not found.” As the Web standardized under the direction of the World Wide Web Consortium (W3C), 404 was maintained as the code for resources that could not be found.

It is a fun story, but according to those involved, it simply is not true. When asked for comment, one of the CERN scientists that worked with Tim Berners-Lee, Robert Cailliau, was explicit. “404 was never linked to any room or any physical place at CERN,” he said in an email interview with Wired magazine. “That’s a complete myth.” I think it is only fair to Cailliau to point out he is not fond of mythology at all. He also told Wired:

> I don’t even have a hunch about the 404 fascination. And frankly I don’t give a damn. The sort of creativity that goes into 404 response pages is fairly useless. The mythology is probably due to the irrationality, denial of evidence, and preference for the fairy tale over reality that is quite common in the human species... These human traits were relatively innocent in the past, when individual influence was small and information spread slowly. Today, and in no small way due to the existence of the net, these traits have gained a power that is dangerous.

Apparently, he tied this kind of mythology with the political issues of Donald Trump, the EU, gun violence, and climate change [284]. I find it fascinating that questions about the 404 error message escalated to some of the most divisive political problems of our time. But perhaps this is also a reflection of how powerfully myths and legends drive the human experience.

As mentioned in the previous section, rendering a single web page almost always means multiple HTTP requests. Importantly, these multiple requests may very well be to *multiple servers*, as depicted in Figure 9-8.

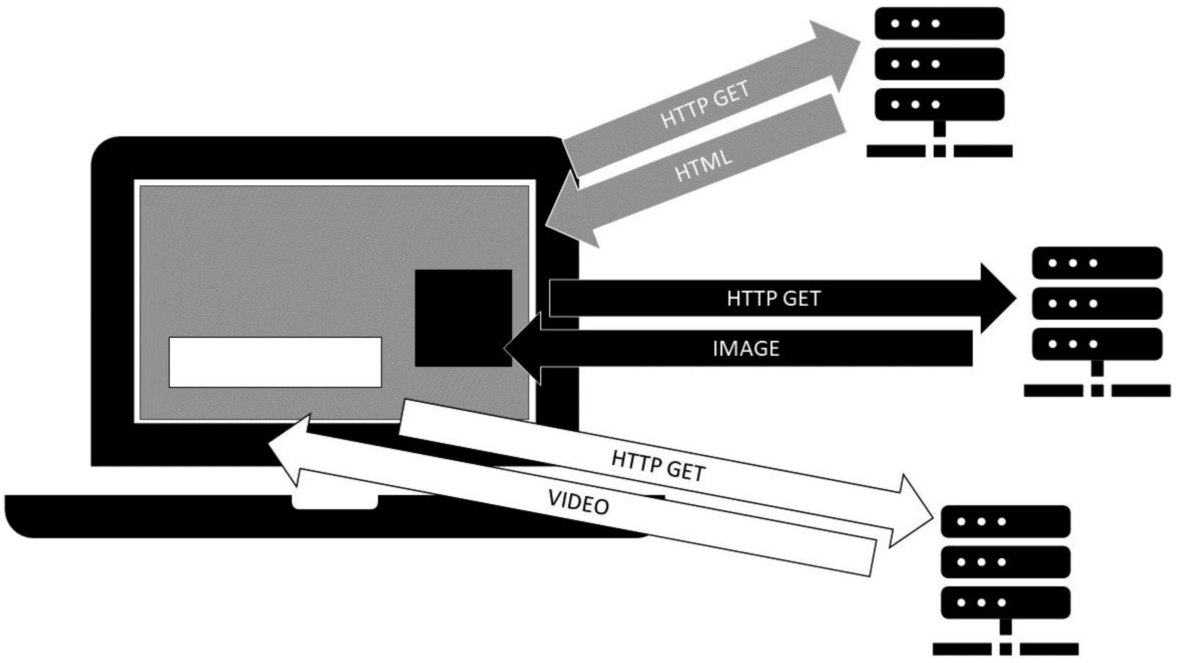


Figure 9-8 Rendering a web page requiring three requests. First, the browser downloads the root HTML page. It must then download all of the components referenced in the HTML, such as images and video

Cookies and State

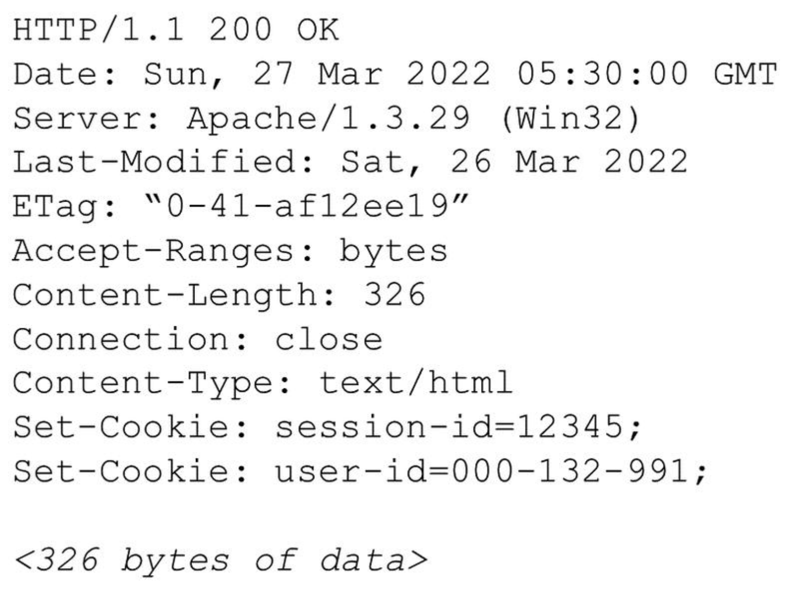
Other than the types of messages browsers and web servers send back and forth to each other, there is one other really critical aspect to HTTP that you need to know. *HTTP is a stateless protocol!*

A stateless protocol is one wherein every message is evaluated *independently from every other message*. Applied to HTTP, this means that a web server evaluates each HTTP request based *solely* on what is in the HTTP request. Relying on the HTTP protocol alone, the web server cannot link the HTTP request to any previous HTTP request. Therefore, it cannot process an HTTP request differently based on what has come before it.

The problem is that we often want our websites to have state. That is, we want our browsing experience to depend on our previous interactions. Here is a very simple example: signing in. If you sign in to the Amazon store, you expect it to be in a different *state* than when you were not signed in. If the web server cannot connect any HTTP requests together, it cannot tell if the sender of an HTTP request is signed in or not. The same is true for maintaining a shopping cart or doing any other kind of operation that needs to "remember" what the user did previously.

The reason for this limitation of HTTP is because of HTTP's origin as a protocol for *document retrieval*. The designers of HTTP did not foresee HTTP being used for stateful operations.

Nevertheless, if you have used the Internet even a little, you are certainly aware that you *do* sign in to websites such as Amazon, and people can, in fact, put things into their Amazon shopping cart. How can a web server maintain state when the HTTP protocol does not do so? The answer is *cookies*.



```
HTTP/1.1 200 OK
Date: Sun, 27 Mar 2022 05:30:00 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 26 Mar 2022
ETag: "0-41-af12ee19"
Accept-Ranges: bytes
Content-Length: 326
Connection: close
Content-Type: text/html
Set-Cookie: session-id=12345;
Set-Cookie: user-id=000-132-991;

<326 bytes of data>
```

Figure 9-9 This is an HTTP response with some "Set-Cookie" headers. These instruct the browser to set cookies for this domain

The term *cookie* just means an opaque identifier. When I say *opaque*, I mean that the identifier is not required to have any structure or meaning. It may just be a random number. By marking all related requests with the same cookie, the web browser can keep track of the state of the session.

The typical flow goes something like this. When a browser visits a website like Amazon, the web server sends a response header back called "Set-Cookie." The Set-Cookie header instructs the browser to set a key-value pair for *any web page in the same host domain*.

In Figure 9-9, the last two response headers are Set-Cookie headers. Notice that the first has a value of "session-id=12345;" and the second has a value of "user-id=000-132-991". This tells the browser to create two cookies. The first, identified by "session-id," is set to a value of 12345. The second, identified as "user-id," will have 000-132-991 as its value.

Once a cookie is set, the browser will include it in a "Cookie" request header with all subsequent requests to the same host domain. This is illustrated in Figure 9-10. If the HTTP response with the “Set-Cookie” headers came from amazon.com, these Cookie headers will generally be sent back in any HTTP request to any amazon.com web server, no matter the path. The Set-Cookie field can include additional information that limits where and how cookies are used, but this is the general approach.

On the server side, when an HTTP request is received with a cookie, the server can look up in a database the state of the requester. So, for example, Amazon would use a process like this. Internal to Amazon web servers, a database keeps track of the cookie. As the user signs in, or puts items in their cart, the information is recorded to the database. Each page the user visits (i.e., sends an HTTP request for with the cookies) is customized based on the information in the database for the given cookie.

```
GET /shopping_cart HTTP/1.1
Host: amazon.com
User-Agent: Mozilla/5.0
Accept: text/html, text/xml, text/plain, image/jpeg
Accept-Language: en-us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
Cookie: session-id=12345;
Cookie: user-id=000-132-991;
```

Figure 9-10 After setting cookies, subsequent HTTP requests to the same domain will include “Cookie” headers, as shown here

You may have read news stories or other reports about cookies and their privacy implications. The privacy implications of cookie use are a serious issue. There are also security issues related to cookie theft or manipulation by attackers. I will get back to these security issues after introducing some of the other technical components.

HTTPS

You learned about TLS in Chapter 6. Secure web communications work by tunneling HTTP over TLS. Even though a web browser differentiates between HTTP and HTTPS as if they were separate protocols, HTTPS is not, for the most part, any different from HTTP. HTTPS is just HTTP over TLS.[3]

I also mentioned in Chapter 6 that exactly where TLS fits in the idealized network stack (i.e., the OSI model) is debated. Most sources place it between layer 4 (transport) and layer 7 (application). The *presentation* layer is supposed to handle things like encryption. However, the TLS protocol in most circumstances does *not* operate like a separate component. Instead, it is built into the applications that are using it. Web browsers, for example, do not have their own TCP (layer-4) or IP (layer-3) components. These are built into the operating system. But web browsers *do* have the TLS code running within the browser itself. The web browser's TLS component is *different* from TLS running in another computer program. In other words, the browser integrates the TLS protocol into itself. This is why I prefer to think of it as an application layer protocol. My interpretation is depicted in Figure 9-11.

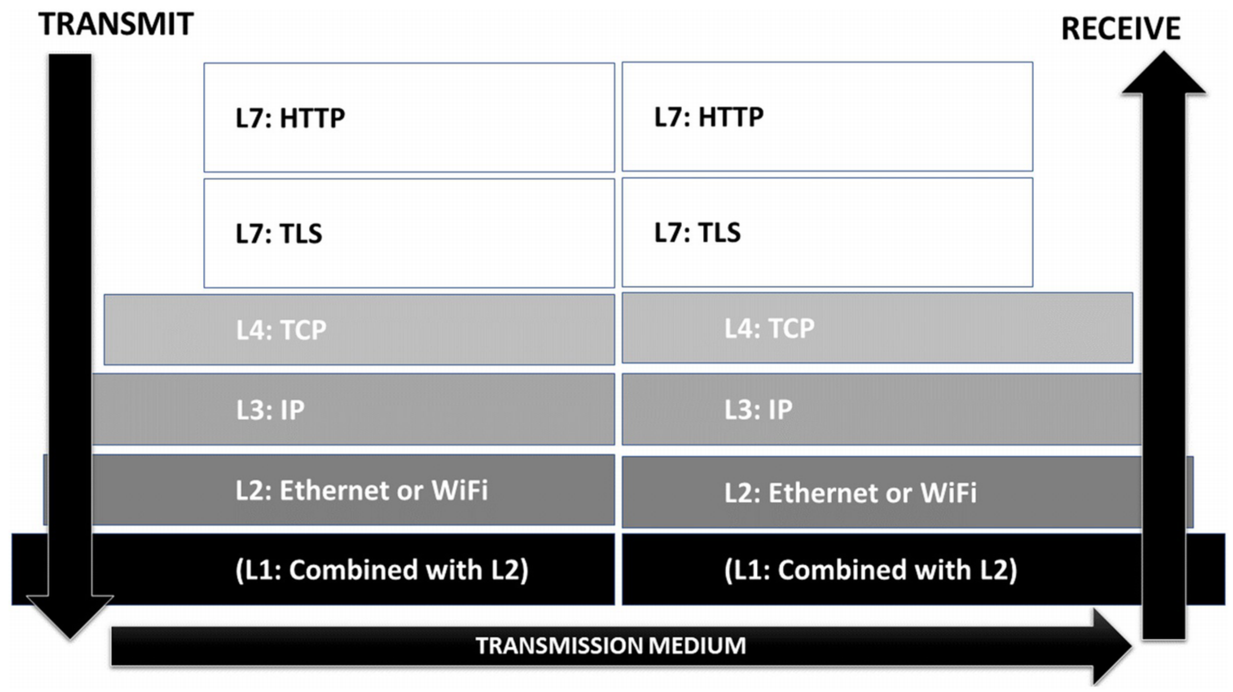


Figure 9-11 HTTPS (HTTP+TLS) when TLS is viewed as a layer-7 protocol

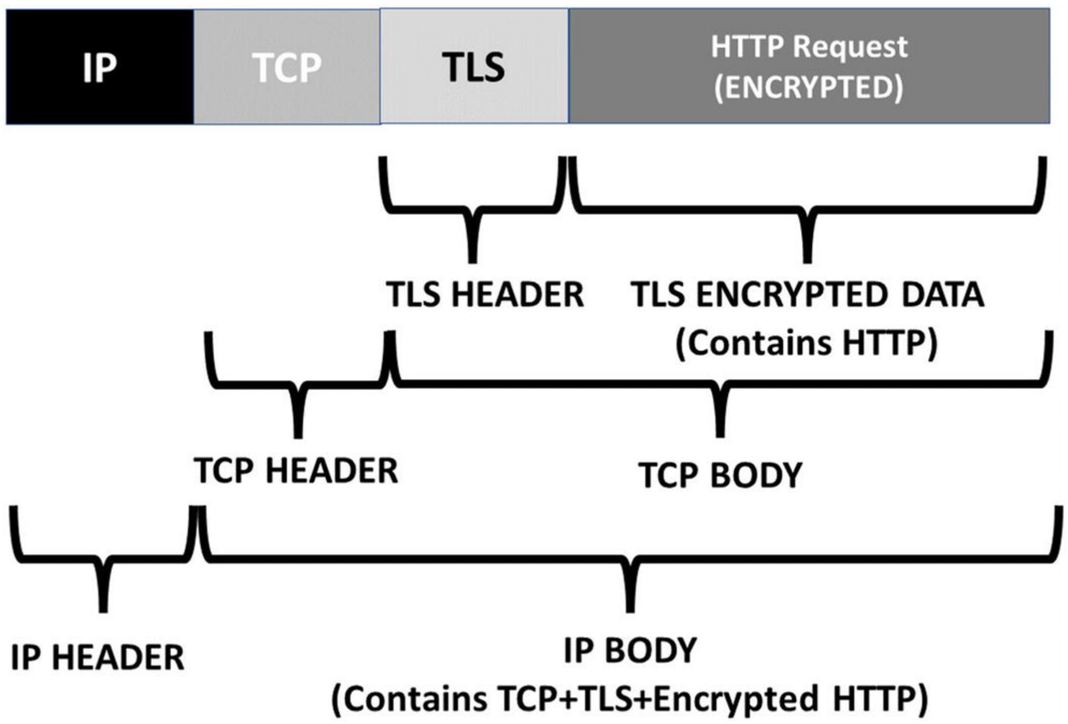

IP
TCP
TLS
HTTP Request (ENCRYPTED)
TLS HEADER
TLS ENCRYPTED DATA (Contains HTTP)
TCP HEADER
TCP BODY
IP HEADER
IP BODY (Contains TCP+TLS+Encrypted HTTP)

Figure 9-12 A representation of an HTTPS packet within the TCP/IP protocol stack. The TLS packet now encapsulates the HTTP data in an encrypted, authenticated envelope

Regardless of how one chooses to taxonomize TLS, it sits between HTTP and TCP and creates a “secure tunnel.” Figure 9-12 illustrates the TLS data in context with TCP and IP. This is simplified. It is not showing data like IV, MAC, or padding data. But generically, there is the readable TLS header information followed by an encrypted chunk.

To help illustrate how TLS creates a secure tunnel, take a look at Figure 9-13. As shown here, the browser prepares an HTTP packet. The HTTP packet is sent through TLS processing before being sent to the TCP/IP protocol stack. *Importantly, the HTTP packet is not sent until the TLS handshake has already taken place!* So both the browser and web server have keys that the man-in-the-middle does not have. From the perspective of the attacker, the encrypted data is completely “opaque.” The attacker can get no information out of the encrypted data.

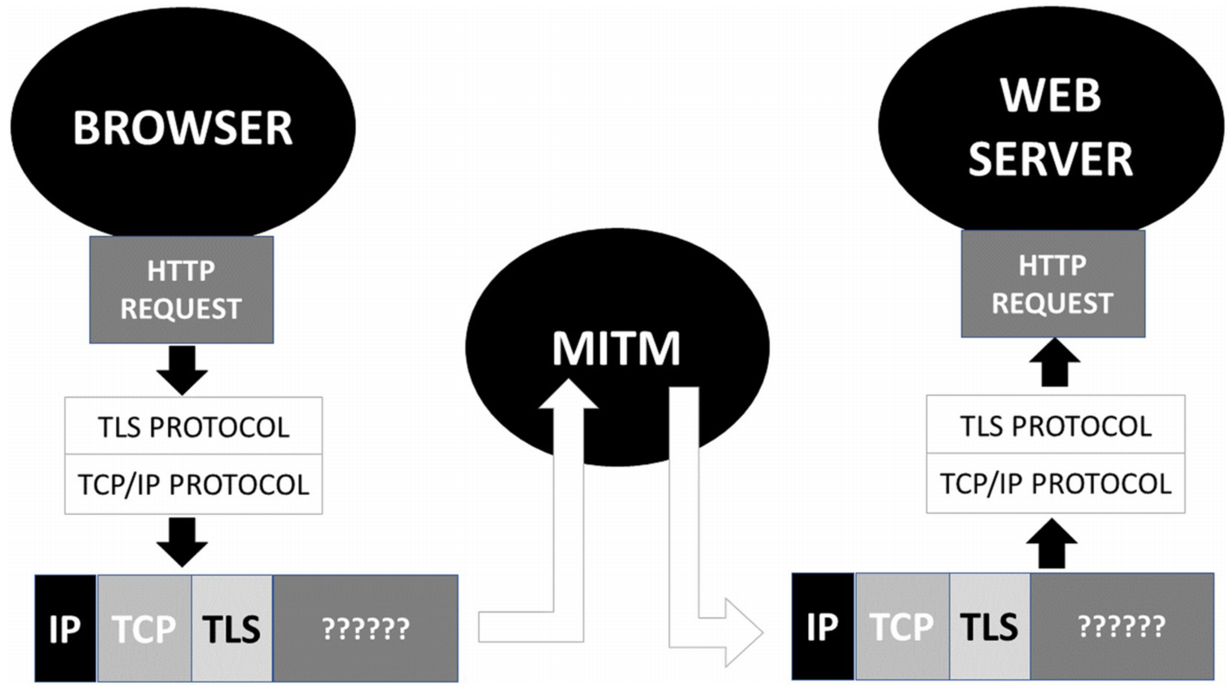


Figure 9-13 A TLS tunnel between a browser and a server. The man-in-the-middle attacker cannot see any part of the HTTP data

Because the HTTP requests and responses are completely encrypted, the attacker cannot see any of the HTTP headers. *Importantly, the attacker cannot see the cookies used for state!* If the attacker could see a cookie, they could use it to steal the victim’s session, meaning if the cookie was used for logging in to Amazon, an attacker that could read the cookie in the HTTP could steal it and use it to access the victim’s account.

What can the attacker see? The attacker can still see the IP address and port. So, for example, a snooping government could still see if data was going to an unauthorized or “illegal” server. And it could use the TCP port number to make guesses about the application (e.g., port 443 would imply web traffic).

The attacker can also gain a little bit of information from the unencrypted parts of the handshake exchange. Normally, the TLS handshake includes extra data I did not describe in the walk-through because it is included in “extensions” to the protocol, and I did not want to get lost in that level of details. But this extra data can give clues to attackers about important information. For example, one very common extension includes the *host name* requested (e.g., “google.com” or “amazon.com”). Note that this does not include the entire URL, just the host name itself, but this does *leak* a small amount of information.[4] Some updates to TLS get rid of this problem, but it is still a common problem.

In terms of operational security, administrators of web servers need to ensure that their private keys are secured and that TLS is correctly configured. With respect to the one or more private keys used for the website, the keys should have limited access within the organization (i.e., to the fewest number of employees possible), stored securely, and password protected. As for TLS, only versions 1.2 and 1.3 should be enabled. Additionally, the TLS (and HTTPS) configuration should be “hardened,” meaning that the configuration should maximize security. While such configurations can be very technical and are beyond the scope of this book, there are plenty of guides and recommendations for this hardening process.

Web Applications

You have now been introduced to the most common and basic web components: URLs, HTML, HTTP, cookies, and HTTPS. Web development teams use these elements (directly or indirectly) to build more advanced systems that I generically refer to as *web applications*. While the term “web application” is not clearly defined, it is typically described as more than just “content.” CNN’s news page hosts “content,” but it would generally not be thought of as an “application.” On the other hand, an online banking system is an application wherein customers can check balances, transfer funds, and pay bills.

Securing web applications is almost universally harder, and usually *much* harder, than securing more general web content. Many of the security principles, however, are broad and can be applied to any kind of web content whether it is a full application or not. For this reason, I am going to talk about “web application security” with the understanding that it can generally apply to other content. If there are exceptions, I will attempt to identify them.

This chapter has, so far, focused on securing individual HTTP communications. HTTPS enables the web browser and the web server to exchange data over a secure tunnel. When everything is working correctly for TLS, anything *in between* the browser and the web server should be unable to compromise the security of the communications.

TLS cannot, of course, prevent bad things from happening at either or both endpoints. If there are security issues with the web server, TLS cannot do anything about it. Likewise, TLS provides no security against bad things happening on the client machine, and specifically in the browser itself. In other words, TLS enables a web application to have security between the browser and server components, but not for the overall application itself. An illustration of this issue is shown in Figure 9-14. As you can see in this figure, TLS connections are a very small part of the entire web application architecture.

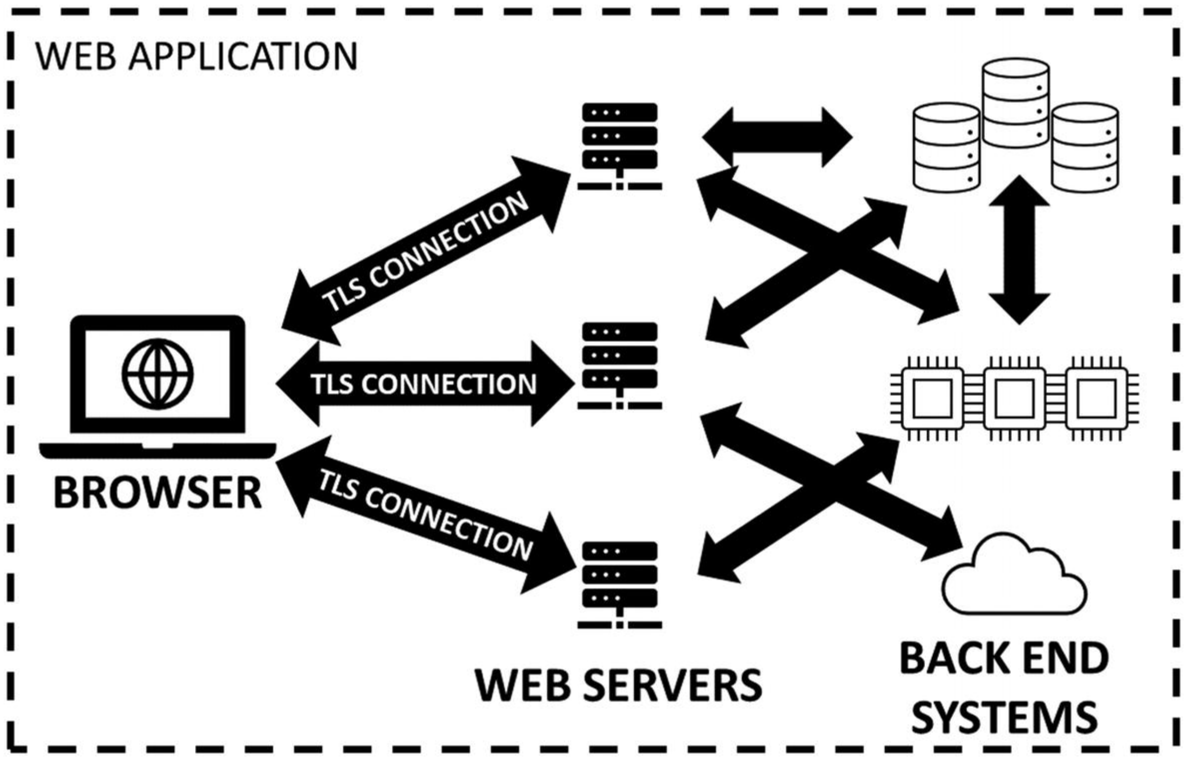


Figure 9-14 A web application is an example of a "distributed system." It works through the cooperation of many systems. TLS is used to protect the communications between remote components like between the browser and the web servers. TLS or other secure protocols are used between the web servers and other backend systems

Figure 9-14 also illustrates another important characteristic of web applications. I have used simplified language any time I have said "browser and web server." In truth, there are often many web servers that help put together a web application. Perhaps more importantly, the web servers often do not have all of the information. Web servers may draw on any number of “backend” servers for data and operations. The backend servers include databases, application servers, and even other third-party cloud systems.

Clearly, securing the communication from the browser to the web server is just a small part of the overall security of the system.

Some of the security issues associated with securing a web application have to do with the underlying technologies. As with many of the other technologies discussed in this book, the technology used in modern systems was not developed for its current purposes but has evolved beyond its original design to handle increased and unpredicted functionality. The original concept of the “Web” was document retrieval. The original designers were not anticipating advanced web applications.

The technology behind a web application can be thought of as having three major categories. The first is the communications protocol between browsers and web servers. The second category is the technologies used to enable the client web browser to render content. Finally, there are other technologies used by web servers in order to provide the contents to the browser.

Client-Side Technologies: Collaborative Websites and JavaScript

Earlier in this chapter, I introduced the concept that a web page is often constructed from multiple HTTP requests. The browser, for example, will get an initial HTML skeleton, and then, based on additional URIs within the HTML, the browser will request additional data such as images and other content. These URIs may be to other servers. This means that a single website *might be constructed from the servers of multiple organizations*. Or, said another way, a single website may be a collaborative construction from multiple parties.

This ability to construct web pages from multiple parties also enables most of modern web advertising. A website can link in a URI to an advertiser component, thus providing space to an ads platform such that the ads are displayed in a space on the page. The images, text, and other content for the ads are downloaded from the ads platform directly to the web page without the website owner having to do much more than insert a link. In modern web pages, most websites do not host an individual advertiser directly. Instead, the website web page includes a link to an advertising broker or ads platform. Individual advertisers sign up with the platform and pay to have their ads shown based on contextual information from the user’s browsing.

The security implications of this collaborative page generation are important. Having multiple sources of content on the same web page requires careful management of security data. One of the big issues is how to handle cookies from multiple sources in a single page. Another big issue is handling programming instructions from multiple sources on the same page.

I have not yet explained to you how programming instructions work in a web page, although I actually introduced the concept in Chapter 7. As illustrated, HTML is a “markup” language. It is not really a programming language. I have seen a very small number of sources that generically refer to it as a programming language, but because it has no real control flow, it does not qualify. Basically, HTML is *static*. For the most part, HTML downloaded to a browser is unchanging. User inputs are limited to new requests to the server, such as clicking a link or submitting a form. The HTML itself is not responsive to inputs.

But as I explained in Chapter 7, browsers do allow snippets of programming instructions, generically called *scripts*, to be downloaded and used in the rendering of a web page. The universal scripting language is called *JavaScript*. This language is used in almost all circumstances to make a web page dynamic.

JavaScript can be inserted directly into HTML between "$<$ script $>$ " tags. For example:

```
 <HTML>
 <BODY>
 <H1 id= "headline" onClick= "addUnderline()">Dynamic Web Page</H1>
 <HR>
 Click the heading above to add an underline.
 <SCRIPT>
 function addUnderline() {
   document.getElementById( "headline").style.textDecoration =
"underline";
 }
 </SCRIPT>
 </BODY>
 </HTML>
```

This HTML looks more complicated, but it is actually very simple once you know what the components mean. Starting with the "H1" tag, you will notice that some extra information has been put into it. It is still an "H1" tag for defining a heading to the page; that has not changed. But we have added some parameter information to it. The first new parameter assigns the tag an ID. The ID can be anything, but I chose "headline" because I thought that was a good description of what this part of the HTML is.

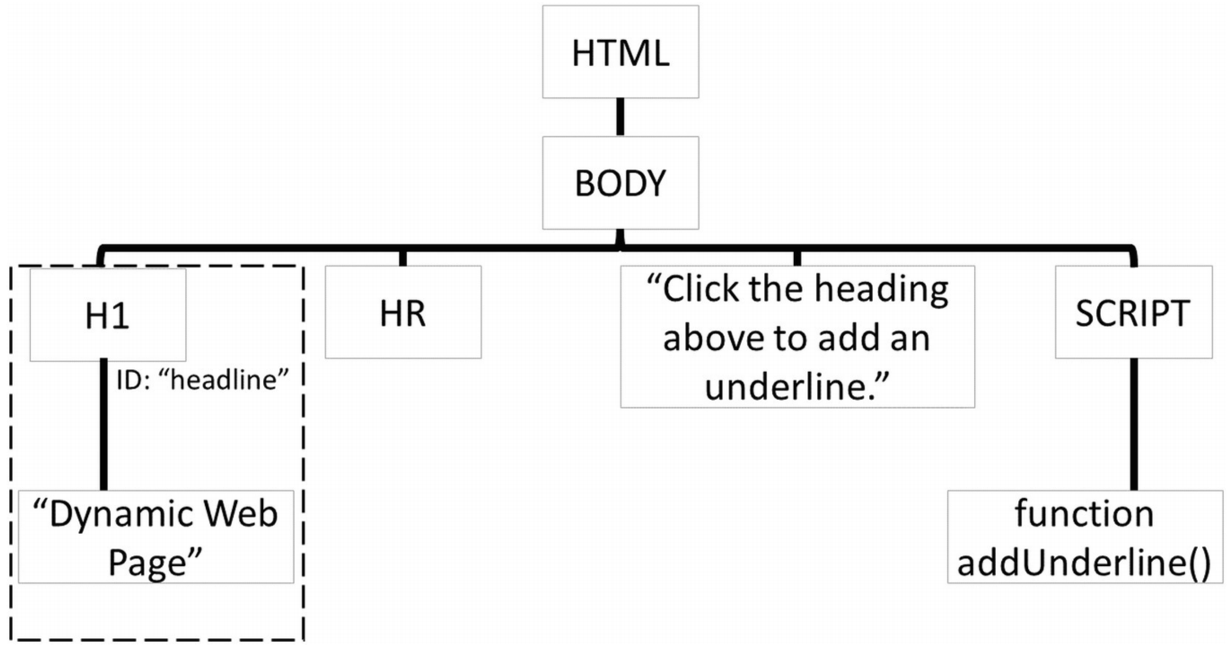
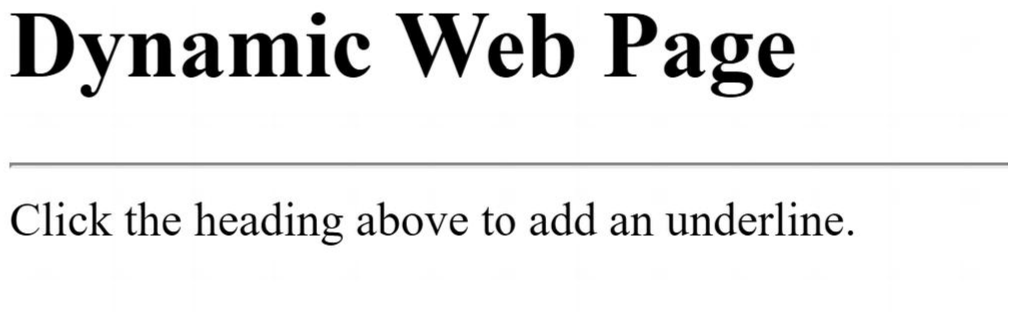


Figure 9-15 An HTML hierarchy for a web page. The HTML is the root, and then the BODY tag next. Note that the H1 tag has text as a subbranch

It is worth noting that HTML is hierarchical. By putting an ID into the "H1" tag, I am identifying that tag *and any subelements*. In this case, the only subelement is the text "Dynamic Web Page," but the point is I am putting an ID on the entire chunk of HTML, not just one tag. Figure 9-15 illustrates how the HTML hierarchy is built. As illustrated, the ID marks the entire branch.

The "H1" tag is also parameterized with an "onClick" value. The "onClick" component tells the browser that if the "H1" tag, or any of the data enclosed within that part of the HTML, is clicked, it should execute a JavaScript function named "addUnderline." I also introduced the concept of a *function* in Chapter 7. By way of reminder, a *function* is a subroutine or subset of programming instructions that is assigned a name. That part of the code can be run by *calling* the assigned name. In this case, the assigned name is "addUnderline," and it will be *called* whenever the mouse is clicked on any part of the data enclosed by the "H1" tag.

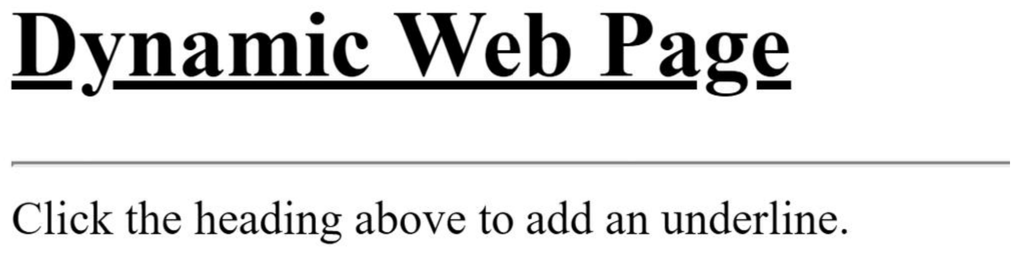
The function must be defined (i.e., created and assigned its name) within a "SCRIPT" tag, but that can happen anywhere in the HTML. In this example, the JavaScript is defined after the "H1" tag with the "onClick," but it could have been defined above it. Even if you cannot understand how programming or JavaScript works, hopefully at least some of the naming conventions are helpful. First, the JavaScript function gets the document. In this case, the document refers to the entire web page.



Dynamic Web Page

Click the heading above to add an underline.

Figure 9-16 A web page with some simple JavaScript before clicking the headline



Dynamic Web Page

Click the heading above to add an underline.

Figure 9-17 A web page with some simple JavaScript after clicking the headline

Without getting too deep into detail, the next part of this JavaScript searches the HTML for an element with an ID called "headline." This will, of course, locate the "H1" tag and its subelements. The JavaScript says to take whatever tag we found (i.e., "H1") and extract its style information. Within the style information, the text "decoration" is set to "underline." This changes the appearance of all content enclosed by the "H1" tag to have underlining turned on.

Figure 9-16 shows the web page when it is first loaded. Figure 9-17 shows the same web page after clicking the heading.

JavaScript does not need to be included directly between "SCRIPT" tags. The "SCRIPT" tag can identify an external JavaScript file that can be downloaded and incorporated. For example, the "addUnderline" function could be stored in another file on the web server called "font_functions.js", and the "SCRIPT" tag could be modified to load it, like this:

```
<HTML>
<BODY>
<H1 id= "headline" onClick= "addUnderline()">Dynamic Web Page</H1>
<HR>
Click the heading above to add an underline.
<script src= "font_functions.js"></script>
</BODY>
</HTML>
```

As long as "addUnderline" really does exist in "font_functions.js", this works just as well. This approach is very common because it allows authors to put many functions, maybe even hundreds of functions, into a single JavaScript file. Web pages can just reference the file with all of the functions, rather than having to make the HTML more confusing and cluttered.

JavaScript is very powerful, but it can be dangerous. As shown in this simple example, it is possible to look up information within the web page itself. JavaScript can *read* its own web page. JavaScript can also send data out over the network. In other words, if an *attacker* inserted JavaScript into your page (say your online banking page), the JavaScript could *read* any of your private information and then send it off to their own website.

Not only that, but JavaScript can *change* just about any part of the web page as it is displayed in the browser. Literally, the JavaScript in a web page could completely rewrite the entire page. Applications like Gmail work by using JavaScript to rewrite the page as needed to expand emails, load settings, and so forth. When you use Gmail, your browser is downloading an entire software system that basically writes the web page contents in real time.

For this reason, there are a lot of rules put into browsers about what JavaScript can and cannot do, where it is allowed to come from, and how it is allowed to interact with the page, the browser, and the system. More on this in a moment.

Server-Side Technologies: Databases, Applications, and Server-Side Scripting

It is common, with respect to web applications, to refer to the *frontend* and *backend* of the system. The frontend deals with everything the user interacts with. All of the HTML and JavaScript you learned about in the previous section are used in creating the frontend experience.

On the other hand, the application's *backend* deals with all of the invisible (to the user) machinery that makes the application work. For example, most web applications require some kind of *database*, a system for systematically sorting, organizing, and operating on data. The user, however, does not *see* the database. At most, they only see information loaded from the database and put into their web page.

For example, to reuse the banking example, when a person checks their balance, the web application must query the banking database and extract the current balance. On the *frontend*, the user is shown their balance. But it was on the *backend* that their balance was stored and processed.[5]

In Figure 9-14, which I introduced earlier in the chapter, I described web servers and *backend* servers. In actuality, the web server itself does, or can do, *backend* processing. In some simple applications, for example, the web server may have a local database running on the same machine. Generating the content for the web server and other operations are all *backend* operations. However, it is not uncommon to refer to dedicated (and invisible to the user) servers that only perform backend tasks as *backend servers*.

Databases

The interactions between a web server and a database are an important part of backend operations. A common workflow is for the web server to receive some kind of input from the user through their browser. The web server responds to this information in part by querying the database and getting results. These results are fed into the response to the user, typically by generating some HTML based on the results. The online banking example illustrates this workflow.

But there are many applications that use the same workflow, but in a less obvious way. Users of Twitter execute this same process when retweeting or commenting on a post. Any of these Twitter operations transmit data to Twitter's servers, which in turn query and update the data stored in databases. These databases store the tweets, who has retweeted them, who has commented on them, and so forth. For any particular tweet, the database has to extract the comments, retweets, and other information in order to build the complete web page for that tweet.

There are a number of different database technologies in use today. One of the older, but still widely used, database formats is called SQL, or Structured Query Language (SQL is often pronounced "sequel"). For these kinds of databases, *queries* are used for any type of operation, including requests, updates, new data, or deletions. Other database approaches may work differently, but just about every kind of database has read and write operations. The write operations, however they are set up, are often a source of security headache for web applications.

Application Servers

Web servers may make use of other application servers as well. The online banking example, which I use and reuse because of how many of these ideas it brings together, needs servers that will handle all of the actual bank processing. For example, when a bank customer uses the online bill pay and the result is a check that gets mailed, the web server will send commands to application servers that carry out this operation. Web servers also have to process the responses that come from the application servers that range from confirmations (e.g., “the check has been sent”) to data generated by the servers.

In the modern Internet, it is quite common for one organization to use another organization’s servers for processing. An interesting example of this is actually two-factor authentication. One common two-factor authentication service is Duo. Duo provides a “something-you-have” authentication service through their mobile app. Other websites can build Duo into their authentication system by making Duo a backend server.

Typically, this works by having the customer (in this case, the website operator) create a contract with Duo and set up an account. The customer usually agrees to pay on some kind of per-verification basis or with larger volume pricing for large-enterprise customers. Once the account is set up, the customer website is provided with an *API*, or application programming interface, that can be integrated into the web server, or an application server that services the web server. When a user logs in to the website, the website will check its own database servers to verify the password, but the website sends a query to Duo to authenticate the user with the Duo Mobile app. Duo does its own authentication with the user and sends the result back to the website operator. This process is illustrated in Figure 9-18. It should be noted that Duo must also be communicating with its own databases and application servers, but this has been left out for simplicity.

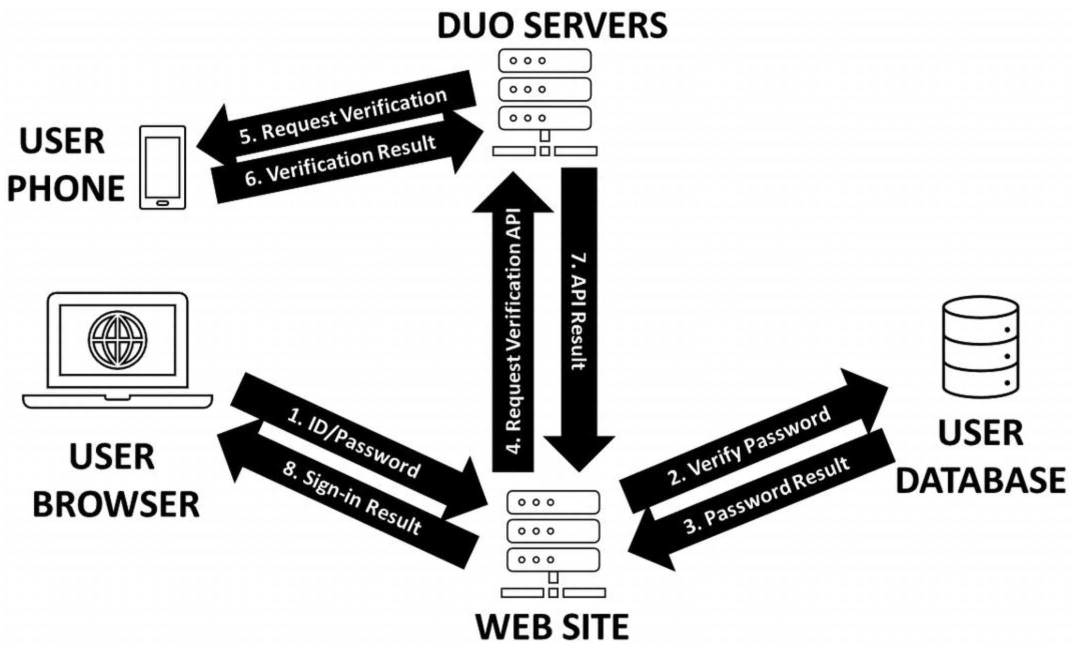


Figure 9-18 A simplified and generic depiction of how services like Duo provide 2FA services to other websites

Although not required, it is quite common for application server APIs, especially those provided to third parties such as in the Duo example, to use HTTPS as the communications protocol. As I mentioned earlier, HTTP can be used for machine-to-machine communications and not just for human-operated browsers. By using HTTPS, the application servers can build on top of all the HTTP machinery that is already commonly in use as well as all of the security of HTTPS. It also provides a very common and standard framework for interfacing as “everybody” (in IT) knows how HTTP works.

When creating an HTTP-based API, there are several “styles” that have emerged. The most commonly used style in today’s Internet is called REST, or Representational State Transfer. REST is not a standard; instead, it is a set of guidelines that suggest how the APIs should be built. The details of these guidelines are not particularly important, but it is helpful to know the term because it is used so widely.

Duo, continuing our example, states in their online documentation that they provide a “RESTful” API [3]. These APIs are described in detail in the documentation. For example, Duo provides an API for querying all of the users that are enrolled for the organization. This is done by sending an HTTPS request to the Duo API web server of the following form:

```
GET /admin/v1/users
```

On the other hand, a specific user can be deleted using the following API:

```
DELETE /admin/v1/users/[user_id]
```

Notice that the API uses the HTTP method as a way of indicating what kind of operation it is. Although there are only a few HTTP methods, such as GET, POST, PUT, and DELETE, these methods really can communicate quite a bit as far as APIs are concerned.

The API uses the path as a type of command rather than a document or web page. The path “/admin/v1/users,” for example, tells the API server what *resource* to get (do you see why “resource” is a good name for this concept?).

These API requests are sent as full HTTP requests, even though they are listed on the documentation page with just the method and path. The documentation assumes the sender of the request can automatically configure the HTTP version. Different HTTP versions should not affect the results of the API request.

The HTTP responses typically include the data formatted into a machine-readable data format. But one of the most common formats is JSON, or JavaScript Object Notation (pronounced "JAY-sahn"). But even though JSON can be read by machines, it is reasonably understandable by humans. For example, the response to the delete user API might look like this:

```
{
   "stat":  "OK",
   "response":  ""
}
```

You do not have to be a machine to guess that this means the server correctly deleted the requested user.

Server-Side Scripting

The final server-side technology to review is server-side scripting. I already discussed client-side scripting wherein JavaScript can be used to make web pages dynamic and interactive. Server-side scripts, on the other hand, are used *to customize or completely generate the contents of a web page in response to an HTTP request*. In fact, server-side scripting is largely necessary to take advantage of the results from databases and application servers.

Although you are probably tired of it, I will return to the example of the online banking system. When the customer requests the balance of their accounts, a simple HTML page to display the balances might look like this:

```
<HTML>
<BODY>
<H1>Account Balances</H1>
<HR>
Your account balances are:
<UL>
  <LI> Checking Account: $100.00
  <LI> Savings Account: $200.00
</UL>
TOTAL: $300.00
</BODY>
</HTML>
```

Figure 9-19 shows a rendering of this HTML. This is clearly not a very sophisticated user interface. A real bank would, of course, use better styles and display things in a far more aesthetically pleasing way. But this page is good enough for what we need to talk about. (Note: The “UL” tag stands for “unordered list” and creates something like a bulleted list. “LI” identifies a single item in the list.)

The problem is hopefully obvious. How does the web page (HTML) get written? After all, the values of the different accounts were presumably pulled from the database at the time of the request.

One solution is to have programs or scripts (generically all called scripts now) on the server generate the HTML from a template. For example, the template might look like this:

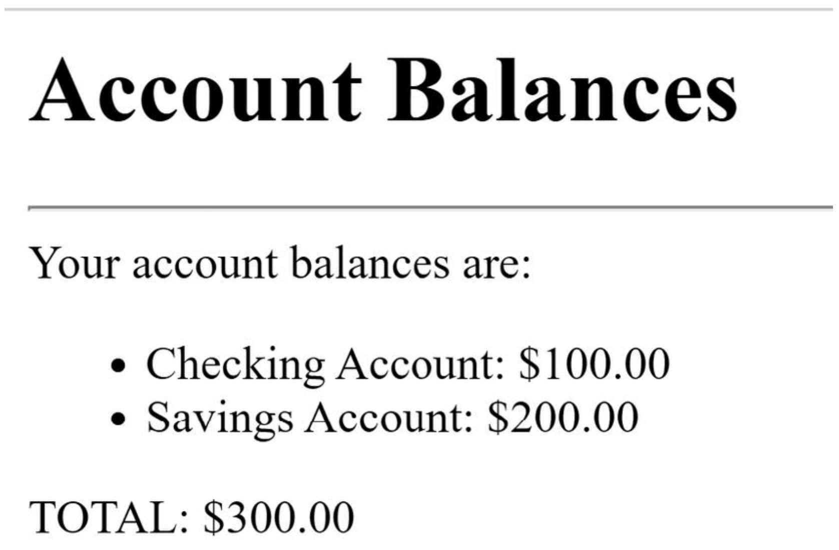
```
<HTML>
<BODY>
<H1>Account Balances</H1>
<HR>
Your account balances are:
<UL>
  <!-- TEMPLATE: BALANCES GO HERE -->
```

```
</UL>
TOTAL: <!-- TEMPLATE: TOTAL GOES HERE -->
</BODY>
</HTML>
```

In this example, the HTML includes “comments.” A comment is information that is ignored by the browser. It is usually for information purposes only. But sometimes template systems put template instructions in the comments. In this example, a computer program would look for any comments marked with “TEMPLATE:” and follow the instructions.

In real systems, template instructions do not say things like “BALANCES GO HERE.” I have put that in to make it understandable for readers not familiar with programming languages. Real systems are written with *identifiers* that tell it what kind of information (e.g., extracted from a database) should be inserted into what part of the HTML.

In a database, an account might be identified by “account_name”, and the balance might be identified (unimaginatively) by “balance.” When extracted, the template system would enable the programming language to write out “ $<$ LI $>$ ” followed by the information identified by “account_name”, the text “: ”, and then the information identified by “balance.”



Account Balances

Your account balances are:

- Checking Account: $100.00
- Savings Account: $200.00

TOTAL: $300.00

Figure 9-19 A very basic and simplified web page display of account balances

Putting it all together, in this example the web server receives an HTTP request from the browser requesting the web page that displays the balance for the accounts. The web server sends a query to the database requesting all accounts belonging to this user and gets back (in our example) a checking account with a balance of $100.00 and a savings account with a balance of $200.00. The server uses a templating system to dynamically generate the response HTML. It builds the HTML listing each account and the value, as well as computing a total value for all accounts. The total is also inserted into the template. When finished, the server has generated all of the HTML required for the browser. It now responds to the HTTP request with its own HTTP response, including the HTML it just generated.

Web-Based SSO: OAuth

Web applications can work together in a number of collaborative ways. One increasingly valuable technology is web-based *Single Sign-On*, or SSO. The concept of SSO is simple. Instead of needing to remember a username and password to a hundred different systems, SSO allows a user to log in to one system and then provides a secure way for the logged-in system to prove to the others that the user is logged in. If the other systems trust the system the user logged in to, there is no need for the user to separately log in to others. On the Internet, this is becoming common; perhaps you have used it yourself. You may have seen websites that permit you to log in using Google, Facebook, or other websites as shown in Figure 9-20.

There are multiple technologies that can be used to provide SSO. One of the more common web SSO technologies is called *OAuth*. OAuth is technically broader than SSO. It is a full authorization framework that allows users to grant *third-party websites* some form of *limited access* to their resources on a *resource server*. Those are a lot of words; let’s walk through each of these key ideas.

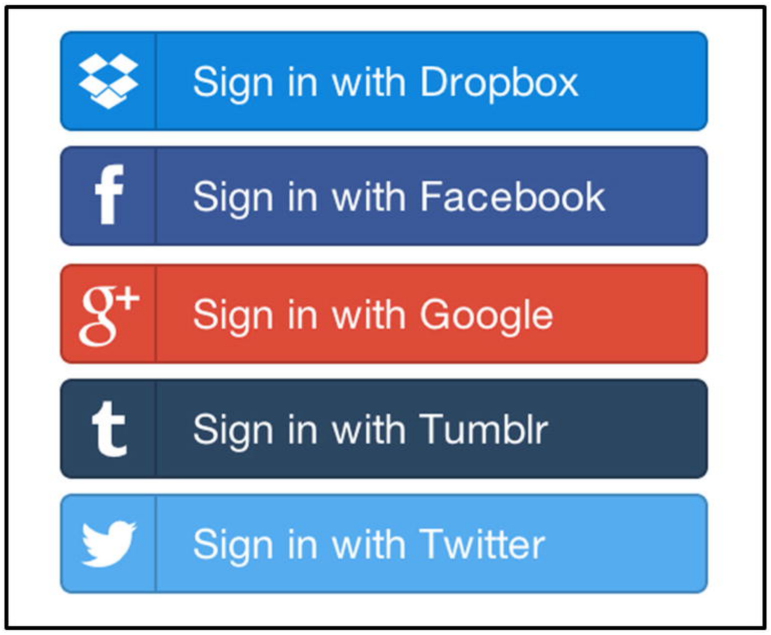


Figure 9-20 An example widget for web-based SSO. Some websites provide widgets such as these to permit signing in using an existing service so that no new password is needed

Starting with the last term first, a *resource server* is some system on which the user (e.g., you perhaps) has some kind of *resource*. For SSO, the resource might simply be an *identity* such as a username. That is a resource because it represents the identity “owned” by the user on the system. As I said, OAuth is broader than SSO and can enable the sharing of other resources. For example, suppose that you have an account on Google. Using OAuth, you might be able to share certain Google resources, such as your contact list or calendar, with another website. For the purposes of this section, however, I will focus on SSO and the sharing of identity. Also, now is probably a good place to introduce the term *resource owner*. This is the term used when talking about OAuth to describe the party with the resources on the server. So if it is your account on Google that you are sharing via OAuth, *you* are the resource owner (i.e., Google is not the resource owner).

The second term, *limited access*, refers to the ability of the resource owner to control how much information the resource server shares. It also means that the resource owner can *revoke* access to shared resources.

Finally, the *third-party website* is any website controlled by a party other than the one that controls the resource server. If, continuing the same example, you create an account at a Pizza website using your Google account via the OAuth protocol, the Pizza website is the third-party website, you are the resource owner, and Google is the resource server.

Because OAuth is a framework, it enables a number of authentication applications to be built using its capabilities. *OpenID Connect* (OIDC) is the authentication protocol that enables OAuth to provide Single Sign-On (SSO) functionality.

This system has a lot of moving parts that are required to make SSO work. To walk through these, I first need to flesh out a few components. I have already explained what a *resource owner* and *resource server* are, but I will repeat them just to have a complete list.

- **Resource**: For SSO, the resource is an authenticated identity (e.g., a username) to be shared.
- **Resource Owner**: The user that owns the resource to be shared.
- **Resource Server**: The website or system or service that will share the resource.
- **Authorization Server**: The system that will *authenticate* the user (e.g., via username and password). This can be the same system as the resource server, but it is a separate function and can be separated out.
- **Client**: This may be the most confusing one because *client* is the *third-party server*. It is called the client, however, because even though it acts as a server to the resource owner, it acts as a client of the resource server *from the perspective of OAuth*.

For OAuth and OpenID Connect, the first step is for the *client* and the *authorization server* to trust each other and exchange some initial information. The *client* (i.e., the third-party website) has to request, and the authorization server has to grant, a *client ID* and a *secret*. The implementation can vary for this step, but there must be a secure mechanism to exchange this data. The data must be secured both *in motion* from the authorization server to the client and *at rest* after receipt. Typically, the data in motion is secured by HTTPS. Secure storage of the data at rest depends on storage security of the client and is outside of the protocol.

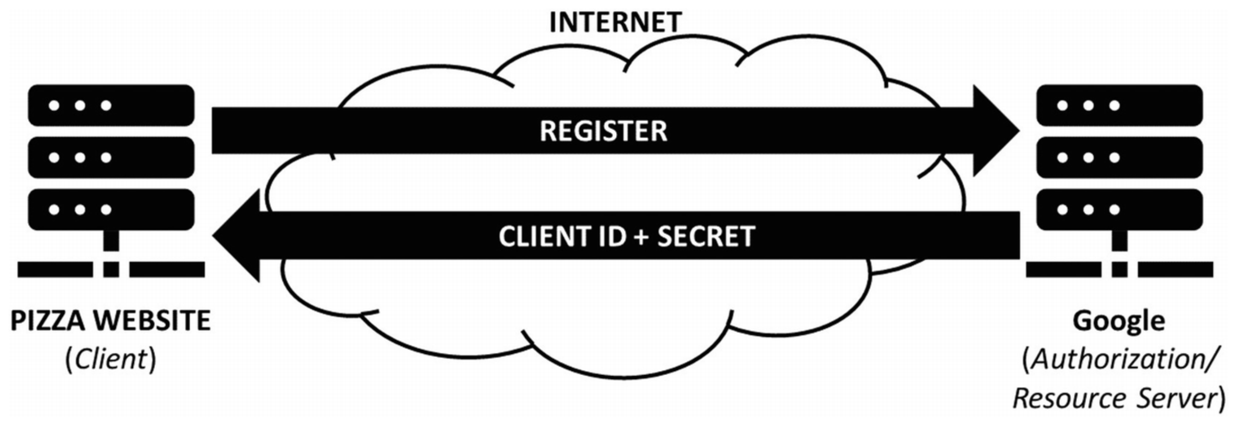


Figure 9-21 In this example, the Pizza website is a client of the authorization server. The authorization server will issue an ID and a secret to the client

One way or another, however, the *secret* must be, as the name implies, *kept secret*. It becomes a kind of machine password used by the client to authenticate itself to the authorization server. The *secret* is typically a long random sequence. Because it is stored and used by a machine, there is no need to worry about if it can be *remembered* by a human. In Figure 9-21, an example is depicted between a Pizza website and SSO using a Google account.

Once the client has a *client ID* and *secret*, the client can provide the SSO functionality. Remember that the client is the *third-party website*, like the Pizza website in the example. As shown in the figure, typically the third-party website will include some kind of widget on the web page that looks like a button or something similar. When the user clicks one of these links (e.g., “Sign in with Google”), the user is indicating to the third-party website that the user is a resource owner of an identity stored on Google servers and would like to authenticate using that identity via OAuth and OpenID Connect. Now that an OAuth protocol will start, the parties take on their roles within that protocol. The user becomes the *resource owner*, the third-party website becomes the *client*, and Google becomes the authorization server and resource server. For the purposes of this example, the authorization server and resource server are combined.

One of the challenges of the authorization sequence is that the authorization server must confirm the identities of *both* the client and the resource owner. Moreover, both authorizations must be tied together. Although OAuth defines a number of means for the authentication sequence, the approach known as *Authorization Code Flow* requires the client to send the request to the authorization server *via the resource owner*. Instead of sending the data directly from the client to the authorization server, the client (third-party website) will send information to the user's browser, then direct the browser to automatically connect to the authorization server with this information. That way, the authorization server will be able to tie the user's authentication and the client request together. Communications sent through the resource owner (i.e., user) are known as *frontend communications*.

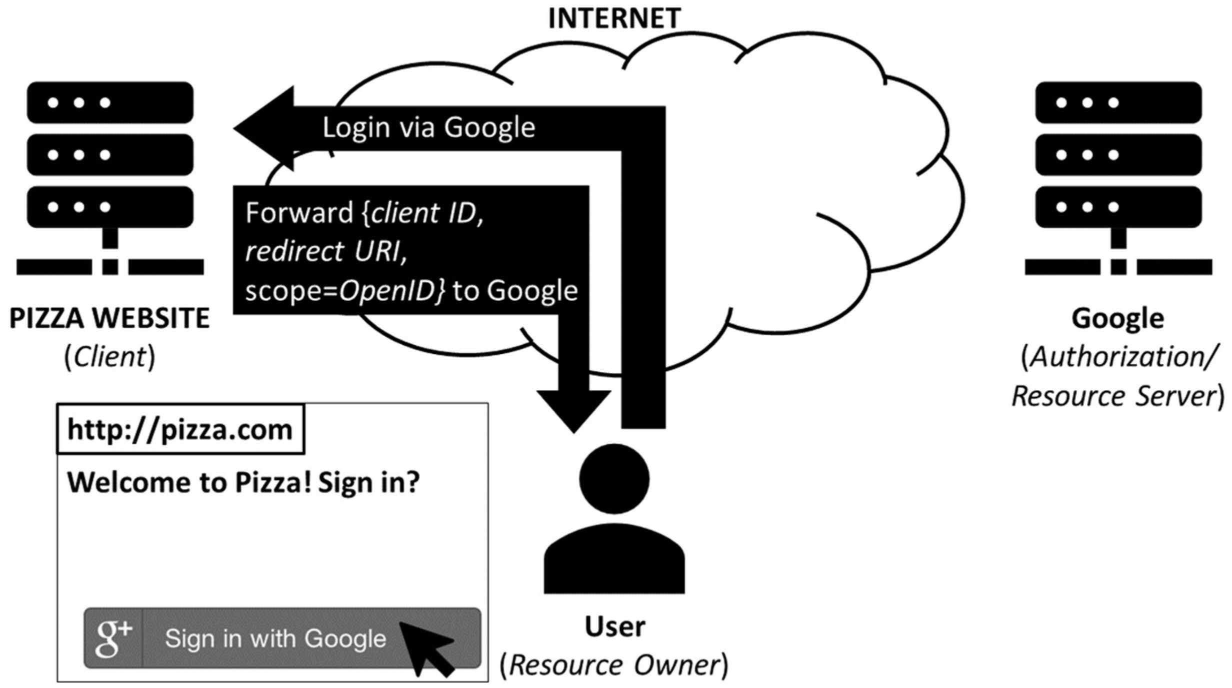

INTERNET
Login via Google
Forward {client ID, redirect URI, scope=OpenID} to Google
PIZZA WEBSITE
(Client)
Google
(Authorization/ Resource Server)
http://pizza.com
Welcome to Pizza! Sign in?
Sign in with Google
User
(Resource Owner)

Figure 9-22 The Pizza website receives a request from the user (the resource owner) to sign in via Google. It transmits its ID and secret along with an

OpenID scope to Google *through* the user. The user receives the Pizza website’s client ID, secret, and scope

The data that is forwarded from the client to the authorization server via the resource owner includes the client’s *ID*, the *redirect URI*, and a *scope* that tells OAuth what kind of request this is. The client ID was received in the registration step previously. The redirect URI is another URI of the client. It is necessary because the user’s browser is being redirected to the authorization server. The redirect URI tells the authorization server how to send the user back to the third-party website when it is done authorizing them. For SSO, the scope will be OpenID. This entire process is illustrated in Figure 9-22.

Once the resource owner has been redirected to the authorization server and forwarded the client’s data, the authorization server will sign in the user if they are not signed in already. The forward and sign-in processes are illustrated in Figure 9-23.

The authorization server will also ask the user to confirm the requested resource access. Typically, the server will tell the user what the client is asking for and ask for confirmation that this is correct. This is illustrated for the Pizza website example in Figure 9-24.

Once all authorization is confirmed, the authorization server will send the resource owner back to the client by redirecting the browser to the redirect URI. The authorization server will also send back an authorization code to the resource owner that is forwarded to the client at the redirect URI. Figure 9-25 shows this process for the Pizza website.

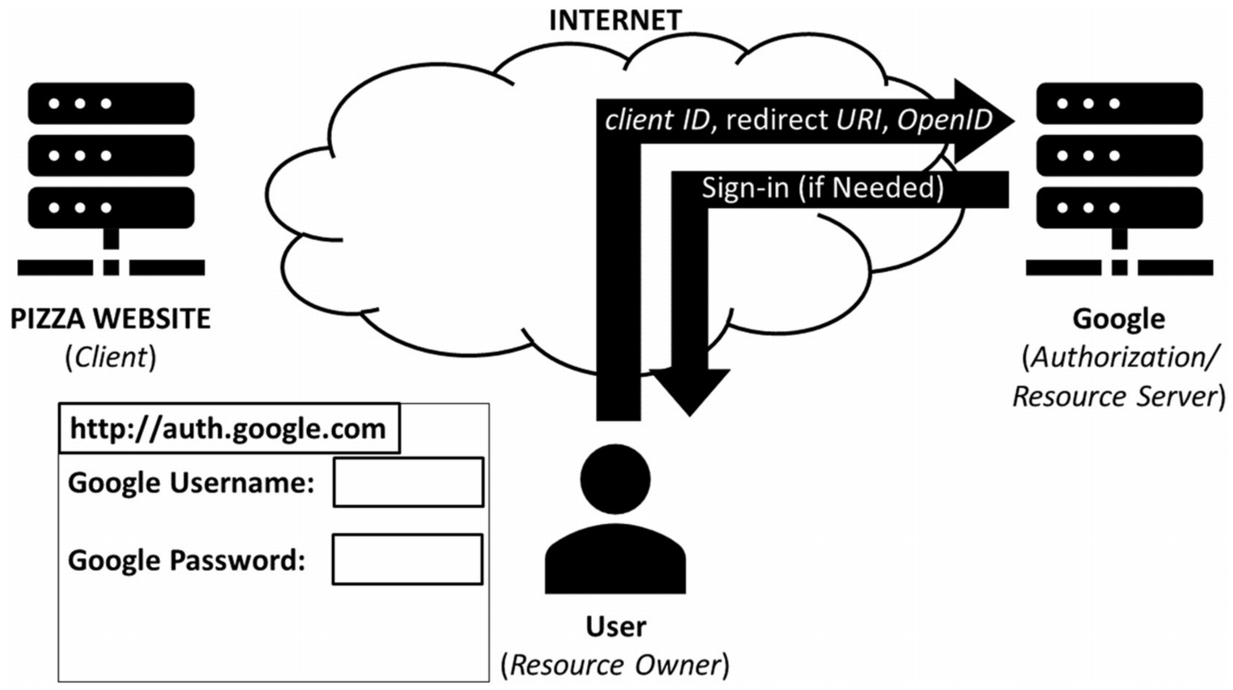


Figure 9-23 The user sends the data received from the Pizza website to Google. Google will provide a sign-in for the user if they are not already signed in

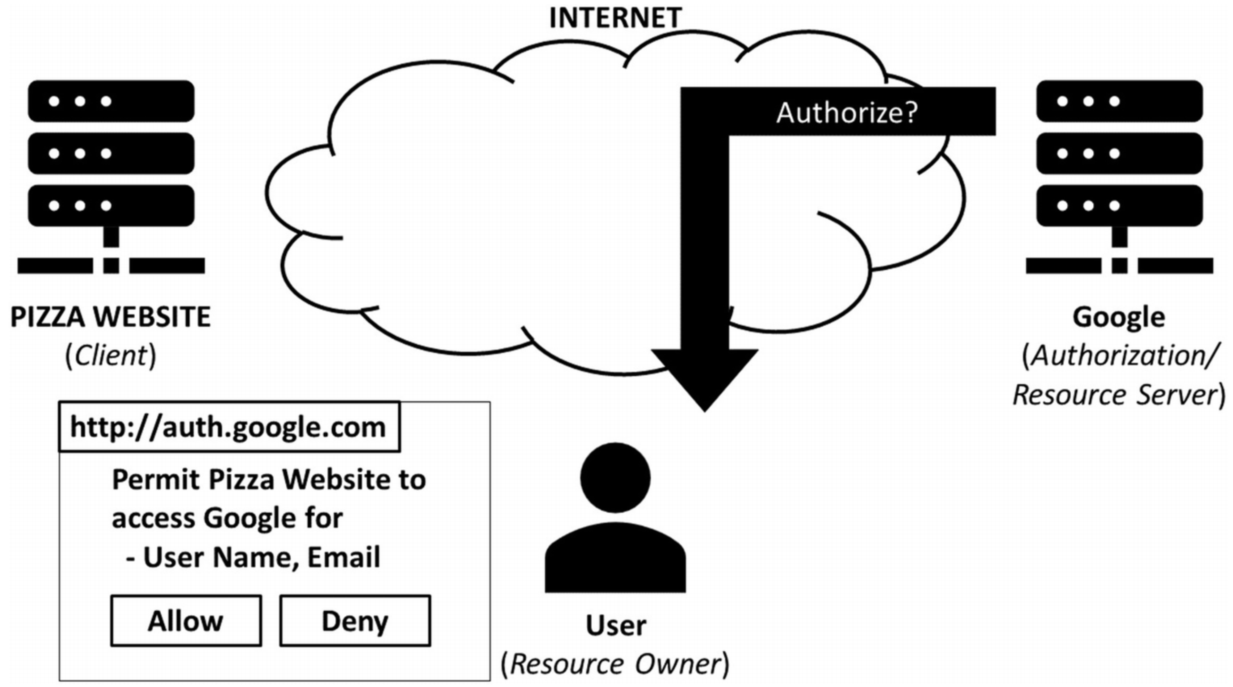


Figure 9-24 Google verifies that the resource request from the Pizza website is what the user expected

At this point, the client has an authorization code that will enable it to talk directly with the authorization server. This direct communication is referred to as *backend communication*. The client transmits its client ID, secret, and the authentication code directly to the authorization server (without going through the resource owner). After verification, the authorization server sends directly back to the client an *access token* and an *identity token* from the authorization server. For SSO, the identity token often has everything that is needed, as it represents Google's "proof" of the user's identity. But the client can use the access token to request additional data from the resource server, such as a contact list or calendar in the case of Google. These final steps are illustrated in Figures 9-26 and 9-27.

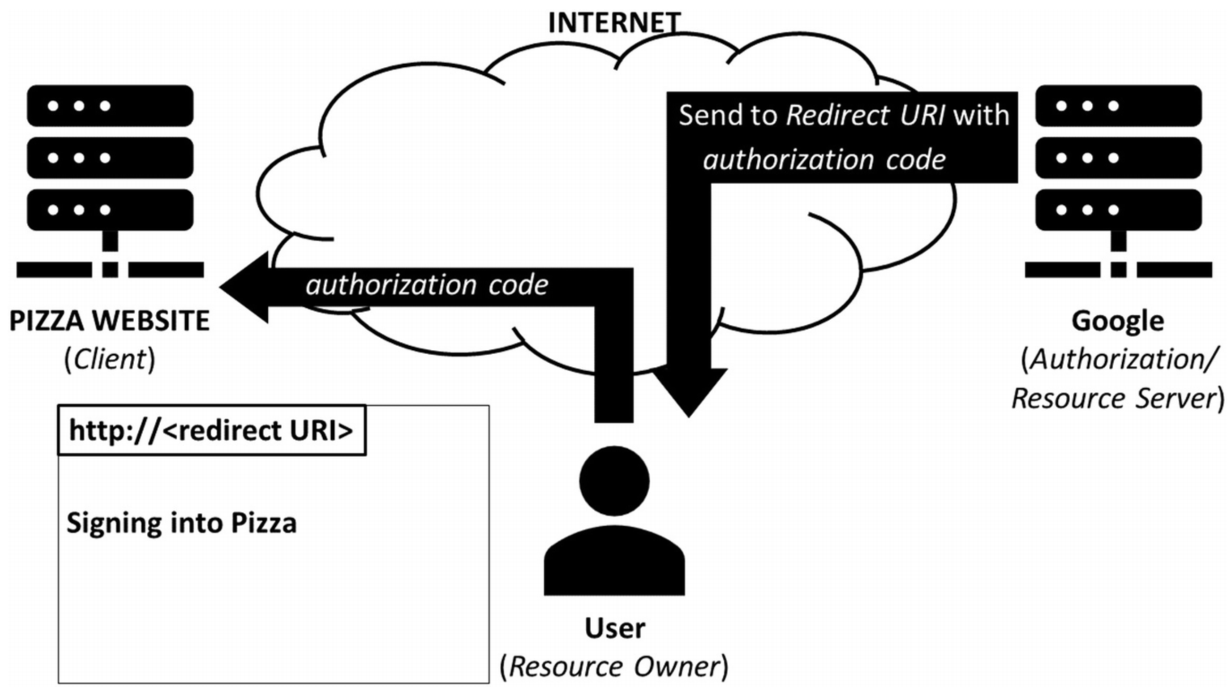


Figure 9-25 Google sends the user back to the Pizza website using the *redirect URI* that was sent by the Pizza website to Google via the user. Google also sends an *authorization code*

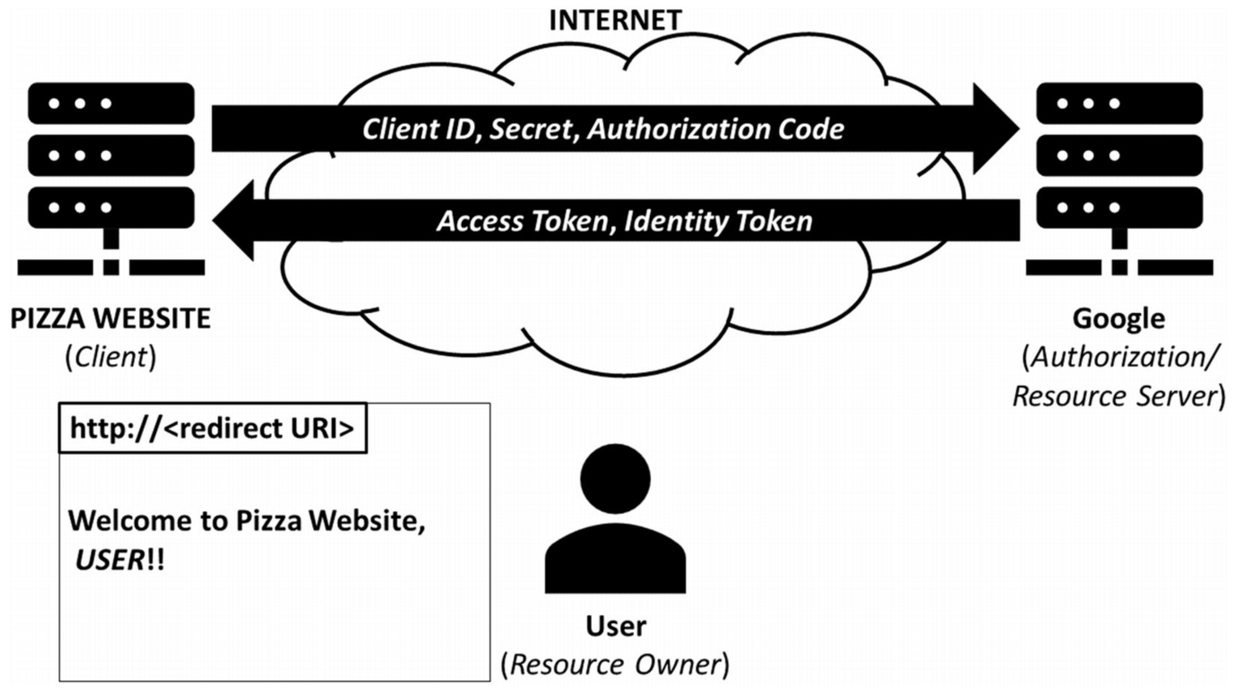

INTERNET
Client ID, Secret, Authorization Code
Access Token, Identity Token
PIZZA WEBSITE
(*Client*)
Google
(*Authorization/ Resource Server*)
http://<redirect URI>
Welcome to Pizza Website, *USER*!!
User
(*Resource Owner*)

Figure 9-26 The Pizza website now has an authentication code from Google. Using this code, it obtains an access token and identity token directly from Google without going through the user

At the conclusion of this process, the client (the third-party website) is assured of the resource owner's identity by the authorization/resource server.

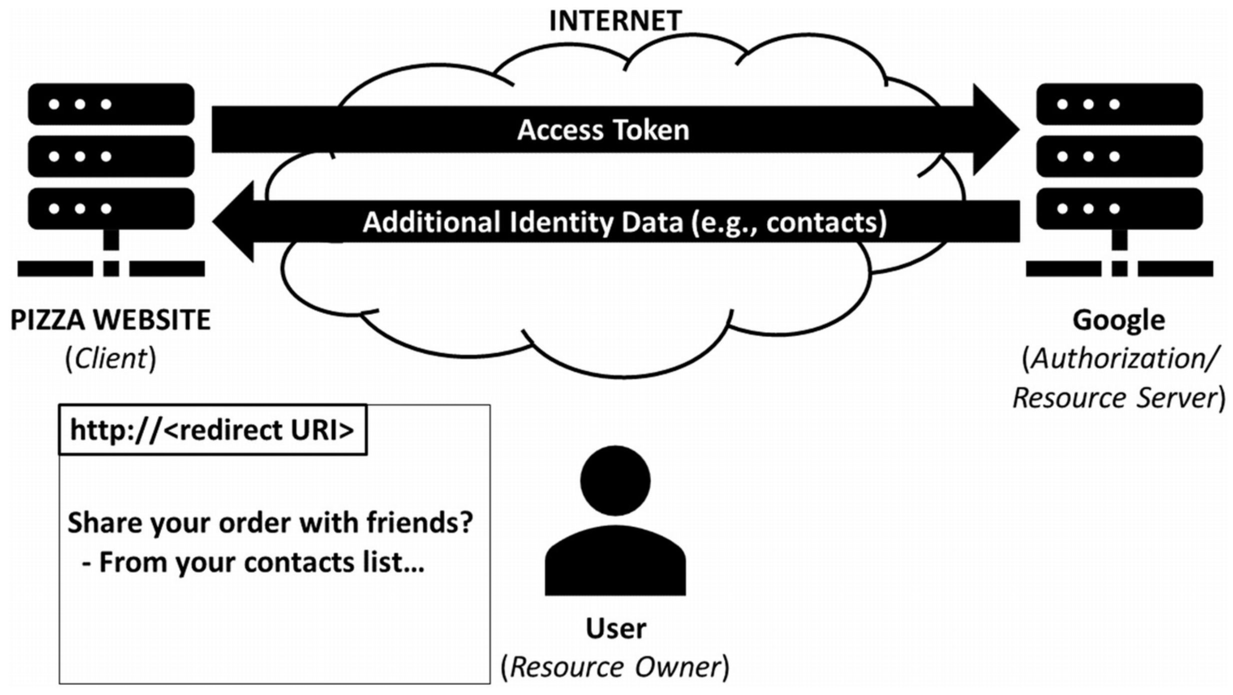

INTERNET
Access Token
Additional Identity Data (e.g., contacts)
PIZZA WEBSITE
(*Client*)
Google
(*Authorization/ Resource Server*)
http://<redirect URI>
Share your order with friends?
- From your contacts list...
User
(*Resource Owner*)

Figure 9-27 Using the access token, the client can request additional data about the user such as a contact list

It is worth noting that the terms I have used in this section are the terms used by OAuth. OIDC by itself uses slightly different terms, calling the client the *relying party* and the authorization server the *OIDC provider*. The resource owner is also renamed by OIDC as the *end user*. You should be aware of these terms, but the OAuth versions seem to be more prevalent.

The OAuth and OIDC walk-through in this section is, like many systems in this book, simplified. However, there are important security lessons to be learned from this overview. All of the communications between the three parties are typically secured with HTTPS. However, there are some other elements of the protocol that provide security properties between the parties.

Starting with the *client secret*, notice that it is *never* sent to the resource owner. The client secret is a shared secret between the client and the authorization server. If it were exposed to the resource owner, the resource owner could *impersonate* the client to the authorization server. Notably, there are some types of web applications in which this could be a problem, and OAuth must use a different type of flow in order to protect the client. While these issues are not covered in detail here, the important lesson is that the client secret must be secured between the client and authorization server only and no other parties (including the resource owner).

The purpose of the *authentication code* is similar. Did you notice that the *access token* was not sent to the resource owner? The client used the authentication code to request it directly from the authorization server. The reason for this is because the access token must also be secured and not shared with the resource owner (or other parties). The authentication code becomes a disposable value that can be passed through the resource owner to the client. Once the client has it, it can communicate directly with the authorization server to exchange this value (which the resource owner has had access to) for an access token which has not been shared or exposed.

Both the *access token* and the *identity token* are typically issued as a type of structured code known as a JSON Web Token, or JWT (pronounced "jot"), that is signed by the issuer. When the access token is received by the resource server, for example, it can always verify that it is a valid token and has not been modified or forged by the client. Similarly, when the client receives the identity token, it can always validate its origin by checking the signature.

Web Threats and Defenses

Now that you have learned about some of the core technologies behind putting together a web application, it is time to learn about how it all goes wrong and what can be done about it.

TLS "Visibility" and Other Attacks

There are a number of ways that TLS can be defeated in practice. The first one I will talk about is unusual because many people do not consider it an "attack." It is usually performed by authorized people and with good intentions. But I think of it as an attack nonetheless.

The technology is called by various names, but the one I'll use primarily is *TLS visibility*. Also known as TLS interception and other names, this system is designed to break TLS by a presumably authorized party in order to examine the data being sent over the encrypted channel. This technology is typically installed on something like a firewall or a device like a firewall. The firewall as part of inspecting TLS data *generates its own root CA certificate*. You may start to recognize from our earlier discussion on TLS tunnels that this would allow the firewall to act as a man-in-the-middle that, unlike the attacker in Figure 9-13, *has the keys to decrypt the traffic*.

Typically, companies that use this technology require their employees and other users to *add this root CA to their operating system and/or browsers*. As you have already learned, once you have control of a root CA, you can generate a certificate for *any website*. Using this root CA, the firewall does exactly that. For every HTTPS-protected website the user tries to visit, the firewall generates in real time a new certificate for the website, signed by the firewall's root CA. The firewall makes a connection to the real website, using its own TLS connection. At the same time, the firewall *pretends to be the real server* to the browser by using the fake certificate. This essentially creates *two* tunnels. A TLS tunnel from the browser to the firewall (which the browser believes to be the real server) and a TLS tunnel from the firewall to the real server. This enables the firewall to decrypt the data from the server at the firewall, examine it for threats, and send it back to the browser.

The companies also configure their firewalls to block all outbound TLS traffic that is not configured to accept the fake root CA. This basically prevents anyone connected to the company LAN from connecting to HTTPS secured websites unless they have compromised their machines. An illustration of TLS visibility is shown in Figure 9-28.

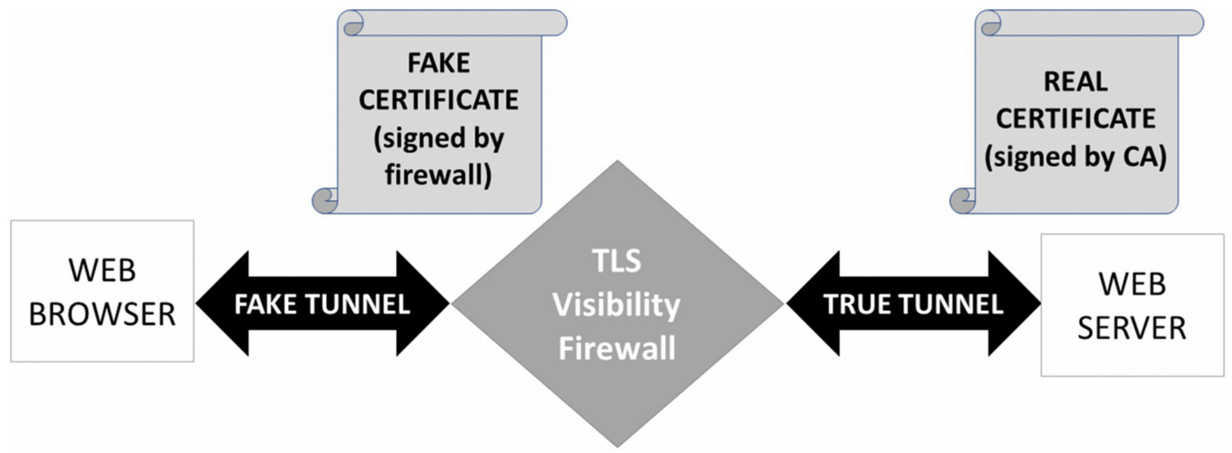

FAKE CERTIFICATE (signed by firewall)
REAL CERTIFICATE (signed by CA)
WEB BROWSER
FAKE TUNNEL
TLS Visibility Firewall
TRUE TUNNEL
WEB SERVER

Figure 9-28 A simplified view of a TLS visibility firewall. The firewall can generate fake certificates for any website and use those certificates to perform man-in-the-middle interception, inspection, and modification

Although this technology is common, popular, and accepted, I think it is bad policy, bad technology, and bad security. Just a few of the problems with it will illustrate the issues.

First, it creates a massive target for attackers. If an attacker can breach the firewall and steal the private key associated with the fake root CA, the attacker can generate a certificate for any website.

Second, it breaks the "end-to-end" contract that users have come to expect from TLS without any indications, warnings, or other alerts.

Third, it trains users to trust and rely on a system that is not working as it should.

For all of these reasons, I oppose the use of this kind of technology. For those people that argue introspection is required, I suggest that new technologies need to be developed. For example, users could be forced to connect to TLS websites through an application gateway. This explicit security technology would force them to know that they are not using end-to-end security and that their connection is monitored. Another alternative would be to create a user agent for corporate devices, which is already common. The corporate agent could decrypt data at the endpoint, transmit it to a centralized system for scanning, and then return it to the user.

TLS has also had a number of attacks against it by more malicious attackers. Many of these attacks have been against weaknesses in the TLS cryptography. As I have mentioned repeatedly, there are *many* configurations of TLS 1.2. Some of these configurations have been found to be vulnerable to different kinds of attacks. These weak or broken configurations should not be used. It is worth noting that TLS 1.3 was stripped down to a much smaller number of secure configurations precisely because of this problem with 1.2. Most of these attacks have "clever" names or acronyms including POODLE, FREAK, Logjam, Sweet32, ROBOT, CRIME, TIME, and BREACH. The technical details of these attacks are outside the scope of this book. But web server operators should make sure they are aware of the current vulnerabilities in TLS and mitigate accordingly.

Another attack of note is *heartbleed*. This attack only applied to a specific implementation of TLS that used nonstandard extensions. So, technically, this was not an attack on TLS per se, but on a commonly used implementation that had some added features. Heartbleed was so called because one of the extra features was something network designers sometimes call a "heartbeat" message. Because connections in TCP, and by extension TLS, are nothing more than messages, there is no way to know if the connection has "gone down." After all, maybe the other side just had nothing more to say right now but will later. A heartbeat message is a very small message sent simply to indicate that "I'm still here!"

In the OpenSSL version of TLS, heartbeats could be enabled. The problem was, there was an error in how they were implemented such that one side of the TLS connection could extract chunks of memory from the other side through a kind of buffer overflow attack. This is not like the buffer overflows from Chapter 7, wherein the attacker inserted code to execute.

Instead, heartbleed overflows permitted the requesting of data far beyond the intended buffer. This basically permitted an attacker to read unauthorized sections of memory of the other side of the connection. The XKCD comic strip has a great explanation of this in comic form at https://xkcd.com/1354/. I highly recommend that as a way of learning more about how it works.

But the reason for mentioning heartbleed here is to emphasize that sometimes things go wrong with specific implementations and not within the fundamentals of the protocol. So it is good for organizations to be aware of the implementations they are using and the vulnerabilities associated with them.

Cookies and Privacy

Because cookies can be used to track a user across web requests, there are very reasonable and difficult privacy concerns. For example, a user might be perfectly comfortable with Amazon knowing what the contents of their Amazon cart are, but might be far less comfortable with Amazon knowing the balance of their bank accounts, potentially embarrassing medical conditions, and socially unpopular political opinions.

Suppose that a user visits a web page about a medical issue and the web page includes advertising from Amazon. Now imagine that the user visits a message board where a protest is being organized and that the message board also includes advertising from Amazon. Many other web pages are visited including news sites, social media sites, and hobby websites. Many of these pages had advertising from Amazon.

After all of these page visits, what does Amazon have?

First of all, one thing they definitely *should not* have are any of the cookies meant for the other domains. That is, even if Amazon is advertising on the web page for the medical site, Amazon should not get the medical site's cookies. Ever. Remember, cookies are what enable sign-in. If an attacker steals a user's sign-in cookie while they are still signed in, the attacker could access their account. The cookie is what determines that the HTTP request is from a signed-in user. If the attacker has it, the web server may not be able to tell the difference.

To enforce this, browsers adopt what is known as the "same-origin policy" for cookies. The same-origin policy states that any information that comes from a particular Internet domain must only ever be sent back to the same domain. The way this works in practice is that when a cookie is set by the server (e.g., "Set-Cookie" header in a response), a domain is set for the cookie. This domain instructs the browser where the cookie should be sent to. For example, if the domain is "google.com," then every site that *ends* with ".google.com" should get the cookies. The same-origin policy prevents the server from setting the domain to anything other than the domain of the web page and any parent domain, excluding public suffixes. For example, a cookie set by "images.google.com" could be sent to the domain "google.com" host, because it is the parent domain of "images.google.com". It could not, however, set the cookie to the ".com" domain because ".com" is a public suffix. Not all public suffixes have just one dot. For example, ".co.uk" is another public suffix.

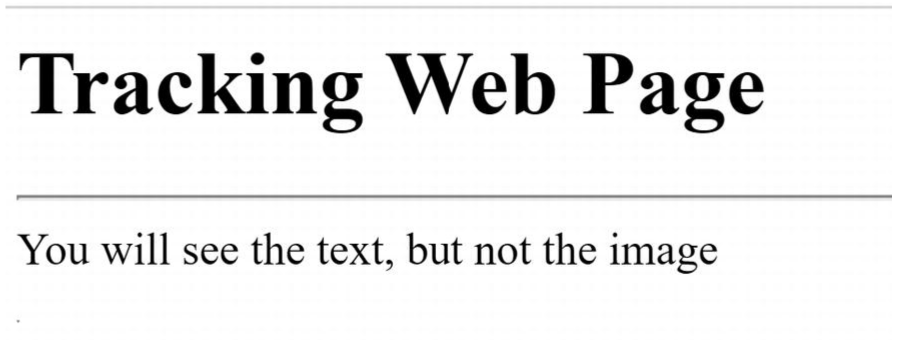
In any event, this policy is meant to provide a first level of protection for cookies. Advertisers are supposedly limited by this as well. So an advertiser on the page still only receives their own cookies and none of the cookies for the hosting page. At least not without some collusion with the hosting page, which unfortunately happens.

It should be noted that even if the advertiser does not get the cookies of the hosting page, it may, in fact, get the name of the website visited. Many of the methods for including the ads into the web page involve what is called the “referer” (note the missing “r”). What this means is that when the HTTP request is sent to the ads server for the ads URL, a “referer” header is often included that indicates the original host URL (or some part thereof).

This means that when the user visits the medical website, even though Amazon only gets its own cookies, the browser also tells Amazon that the request for the ad came from the medical website. Amazon can now associate that website with the user associated with the cookie. Next, when the user visits a message board, Amazon is again informed via the “referer” header of the website and adds this to the information known about the individual user.

One of the solutions that is just now beginning to be enforced in all browsers is the complete blocking (by default) of all third-party cookies. A third-party cookie is a cookie set by any domain other than the domain typed into the URL bar of the browser. In the example used in this section, when the user visits a medical site, the medical site sets first-party cookies. But if Amazon advertises on the page, any of its cookies would be third-party cookies. Most browsers have permitted the blocking of third-party cookies for some time, but now most browsers are moving to adopt this as the default policy.

Sadly for those concerned about being tracked, the advertising companies are already working around the limitation. One of the most common ways to do this is for the hosting page to voluntarily transmit unique information directly to the ads (or other third-party) server. One way this is done is to put the information as a parameter to the URL.


Tracking Web Page

You will see the text, but not the image

Figure 9-29 The rendered HTML code, with the “tracking pixel” present but hard to see

For example, suppose that the ads server has a normal URL of “www.tracking_ads_server.com/ads”. It is trivial to send information to this URL using the parameters discussed earlier in this chapter, for example, “www.tracking_ads.server.com/ads?user=john”. When websites contract with ads servers, the ads server will typically provide integration code for the website, so that the transmission of this information is automatic.

Tracking can happen without visible ads as well. One of the ways this is done is using a *tracking pixel*. This often takes the form of an image tag in the HTML, but either completely invisible or just a single dot on the screen, called a pixel (hence the name). Recall that when an image is inserted into HTML, there is a “SRC” parameter that tells the HTML where to go to get the image. A tracking pixel requests an “image” from the advertising or third-party server. But the purpose is not to get an image, because the image will not be displayed. The purpose is to send information to the advertiser. Here is an example:

```
<HTML>
<BODY>
<H1>Tracking Web Page</H1>
<HR>
You will see the text, but not the image
<P>
```

```
 <IMG SRC= "https://www.crimsonvista.com/img/logos/CV_icon.png?
user=john" width= "1" height= "1">
 </BODY>
 </HTML>
```

Figure 9-29 shows the rendering of the HTML code. Notice that the image is not easily seen. Figure 9-30 has it a little larger. If you look closely, you will see it underneath the text. Just to be clear, the image “CV_icon.png” at my company’s website does not do any tracking. But the Crimson Vista web server is receiving an HTTP request with that path, including “user=john”. It is transmitted, but because Crimson Vista is not tracking anything, the data is ignored.

Because websites get paid to host ads on their site, there is an incentive for them and the advertisers to work together. As long as this is the case, it is unlikely that information sharing will cease.

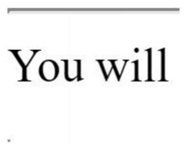


Figure 9-30 Close-up of the rendered HTML code. Notice the “tracking pixel” in the bottom-left corner

Story Time: Browser Fingerprinting and FingerprintJS

There are other ways of tracking people browsing the Web that are even more terrifying. Browser “fingerprinting” is a technique used to identify or track an individual by tracking their specific browser. That is, if the unique browser can be tracked and followed, the user can be tracked (assuming there is only one use of the browser).

This technology works by by collecting and analyzing information about the browser configuration and system settings, such as operating system, browser extensions, time zone, language, and screen resolution [129]. Some limited fingerprinting can be performed on the server, but most information is collected on the user's computer itself by running JavaScript code in the browser without the user's knowledge. Like all tracking, this data can be used for targeted advertising, behavioral profiling, and fraud detection.

One reason why browser fingerprinting is so insidious is because it can be used to identify users even if they clear their cookies or use private browsing modes. It is often used as an alternative to traditional tracking methods such as cookies, which can be blocked or deleted by users.

FingerprintJS is a browser fingerprinting library that collects information about a user's browser and device to create a unique identifier, or "fingerprint." It is often used by websites and online services to track user behavior and identify unique users, even if they try to mask their identity using privacy measures such as using a VPN or browser extensions [108].

JavaScript Protections

As discussed, JavaScript can do just about anything to a web page. It can read data from the web page, change the data in the web page, and even send data to arbitrary servers. Even if JavaScript is loaded into its own page (i.e., is first-party code instead of third-party code), it could still potentially trigger damaging things to web pages in other browser tabs. For these reasons, JavaScript must be carefully controlled.

To help confine JavaScript and limit the damage it might be able to do, browsers enforce the same-origin policy on JavaScript as well. The same-origin policy is quite a bit more complicated for JavaScript than for cookies, because JavaScript is a programming language and can do a lot more things. In fact, the same-origin policy has so many different applications to JavaScript that I will not cover them all here. Instead, I will focus on a few core examples.

First, the same-origin policy prevents JavaScript from, with very few exceptions, querying data from any part of the document (i.e., web page) that is from a different origin. This, of course, prevents JavaScript downloaded from an advertiser from reading data out of the hosting page.

Second, the same-origin policy prevents certain kinds of network communications. JavaScript has multiple mechanisms for sending data over the network. One type of request is an XmlHTTPRequest. This kind of request does not require the page to be reloaded. This makes communications with the server much faster. However, the same-origin policy prevents JavaScript from *reading* data from any request to a different origin (it does *send* the request, however). This general approach holds for many types of requests. For example, JavaScript can request JavaScript from another origin and can even execute its functions, but it cannot actually read the JavaScript code itself. The main value of this policy is to prevent JavaScript from reading any data from any domain other than where it originated.

One of the main reasons for this policy is to prevent third-party JavaScript from using the browser’s current cookies to get unauthorized data. Remember: The browser will send the cookies it has to the designated domain for *every request to that domain*! This includes requests sent by *third-party code*! If the third-party code triggers a request to another domain, the browser *will send with it all the cookies corresponding to that domain*! Although the same-origin policy allows JavaScript to make the requests, by blocking any reading of that data, the third-party JavaScript cannot learn any confidential information.

The third, and last, application I discuss here is how the same-origin policy applies (or does not apply) to form submissions. There are two parts of web forms. A browser must visit a website to have the form rendered on the browser (note, however, it can be hidden from the user). The second part is a mechanism whereby the data inserted into the form is transmitted to a server, generally transmitted using the POST HTTP method. It may seem strange, but forms, or parts of forms, can also be *hidden*. The hidden form cannot accept user values but can be preloaded with default values. This is commonly used to ensure that a form transmits some kind of prepackaged data.

The same-origin policy, with its focus on blocking reading but not writing, does *not* prevent JavaScript from transmitting a form, downloaded from one domain, from being submitted to another origin. This is a very serious problem because it enables an attack known as a *Cross-Site Request Forgery*, or CSRF. The name "cross site" refers to the downloading of the form from a different domain (or site) than where the data is submitted to.

A CSRF enables an attacker to *change the state* of a user's web application by triggering data transmission. Suppose that the user is logged in to their online banking account at the same time that they are tricked into visiting an attacker's website. If the attacker's website submits a form (usually hidden and automatically submitted) to the online banking website, the form will be sent to the banking website with all of the currently logged-in cookies. The form submitted is intended to trigger some kind of operation on the banking side, such as transfer of funds. This process is illustrated in Figure 9-31.

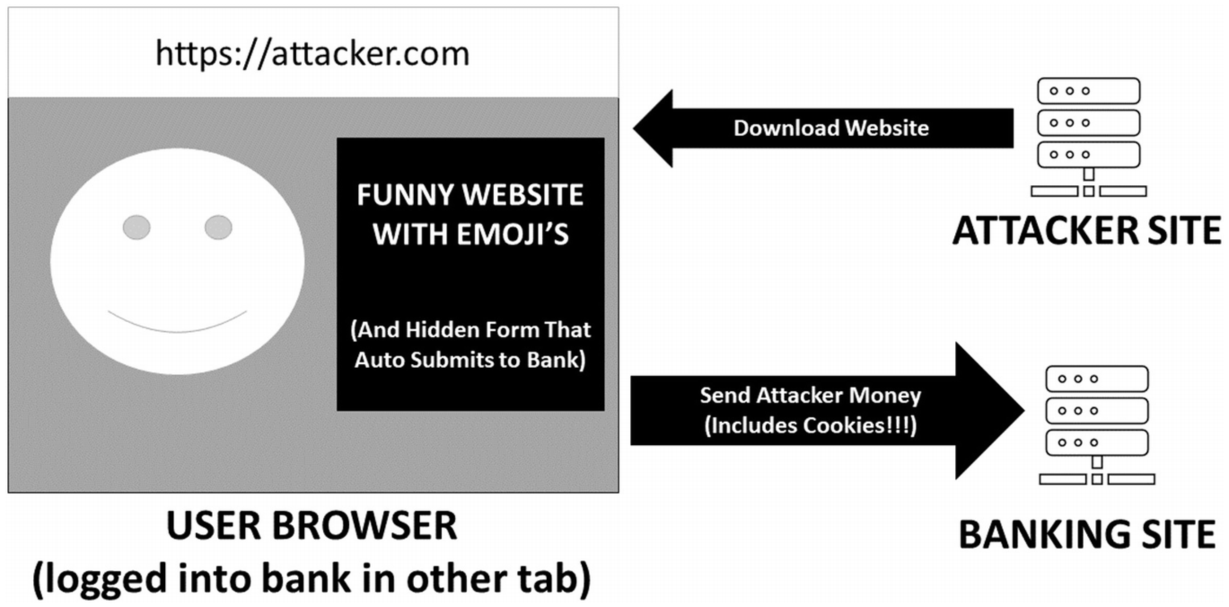

https://attacker.com
FUNNY WEBSITE WITH EMOJI'S
(And Hidden Form That Auto Submits to Bank)
Download Website
ATTACKER SITE
Send Attacker Money (Includes Cookies!!!)
BANKING SITE
USER BROWSER
(logged into bank in other tab)

Figure 9-31 A simple example of a CSRF attack. The attacker website appears to be benign. But it has a hidden form that auto-submits to the user's bank. If the user is logged in, the submission will be accepted

Notice that this example in the figure has nothing to do with a third-party code. The two websites are running independently. But the problem is, the browser has cookies for the online bank in another tab (not shown). So when the user visits the attacker's website, the hidden, and automatically submitting, form is transmitted to the online bank *with the appropriate cookies*!

For this attack to work, the user of the browser does have to be currently signed in to the online bank. And it does require the attacker to know what kinds of forms can be submitted to the bank in order to trigger a response. It is also worth noting that the same-origin policy does prevent the attacker from seeing the response, but in this case the attacker is less interested in reading the result. Sending the information is what the attacker requires.

As stated, the same-origin policy does not solve this problem. The most common solution is to use a CSRF *token* to prevent these kinds of attacks. The idea is that every form *in a legitimate* page includes a *hidden field with a random value*. This hidden value is transmitted with the form data, just like any other field in the form. But it is a value that the server validates is present before accepting the form data as valid. When the attacker tries to submit their fake form, they cannot predict what value to put into the hidden field. When the fake form data is submitted, the server will reject it for not being a valid token.

The same-origin policy is helpful with CSRF tokens because if the attacker could *read* a downloaded page, the attacker could just copy the CSRF token out of it. But the same-origin policy prevents them from being able to get this data.

SQL Injection Attacks

Another attack that can be used on a web application is to attack the *backend*. The backend is meant to be invisible to users, but that does not mean an attacker cannot figure out what kind of backend systems are in use. A clever attacker can sometimes figure out how to manipulate the frontend in order to insert “evil” inputs into the backend.

One of the most common forms of this type of attack is SQL injection (usually pronounced “sequel injection”). As a reminder, SQL is a popular type of database. For these kinds of databases, queries are created as SQL commands. The SQL commands for a given web operation are typically based, at least in part, on some kind of input from the user. An evil user may find ways to transmit an input *with SQL commands in it* that get inserted (or injected) into the authorized SQL.

Without getting into too many details about how SQL works, the following is a prototypical SQL command for looking up the password for a user with the name of John:

```
 SELECT password FROM user_auth WHERE username= 'John'
```

Of course, when a web server needs to query a database, it does not know ahead of time the user’s name. That is an input from the user, perhaps at a sign-in field for the web application. So the website might do something like this:

```
 SELECT password FROM user_auth WHERE username= '[USER INPUT]'
```

In this example, the user’s input replaces “[USER INPUT]”. So, if the user typed in “John” for their username, it would result in the first SQL example earlier.

But what if the user did not enter their name? What if, instead, they entered an input like this:

```
 X ' OR 1=1 OR username='X
```

When this is inserted into the SQL command, it now looks like this:

```
 SELECT password FROM user_auth WHERE username=
'X' OR 1=1 OR username= 'X'
```

This new command says “get the password for a user whose name is ‘X’ or if 1 equals 1.” Because 1 always equals 1, this would get the password for every user in the database. The reason for adding the duplicate “username=‘X”’ is simply to have a matching single quote. Remember, the template gets dropped in between quotes. So without the duplicate username (or something like it), we would get this:

```
 SELECT password FROM user_auth WHERE username= 'X' OR 1=1 '
```

SQL would treat this as an error, because it expects a matching single quote somewhere.

Assuming the attacker figures out a correct construction for the injection, the SQL database will execute the modified instructions. Exactly what would happen depends on how the output of the database is processed by the web server, but in some circumstances, it might dump all of the usernames and passwords into the web page.

Even if it does not display much that is helpful, similar tricks can be used to delete parts of the database, change values, or otherwise corrupt the system.

Defending against this kind of attack typically requires the two-step process known as *sanitize and filter*. Basically, “sanitize” is what a web server should do to any incoming data from a user. The server should *assume* that inputs from the user cannot be trusted. Sanitizing is a way of eliminating unacceptable inputs. A username, for example, can be prohibited from having single quotes.

Filtering, on the other hand, is applied to the data after sanitizing, but before going to the database. Essentially, the commands being sent to the database need to be scanned to make sure they look like sane commands. When reading from the database (e.g., to get a password), for example, there usually should not be any writing. Commands can be filtered to reject any such unexpected directives.

Most web servers created these days have built-in libraries to help ensure that data is properly prepared before being sent to the database. Most libraries have their own systems for inserting parameters into SQL statements. These systems enforce parameters to follow the rules no matter what input is sent to them. These kinds of attacks are completely preventable. They still tend to show up because some web developer forgets to use the library or otherwise customizes a system in an insecure way.

Story Time: SQL Injection on WordPress Sites

In 2022, Patchstack discovered multiple security vulnerabilities in the popular WordPress plug-in “LearnPress.” One of the vulnerabilities, CVE-2022-45808 [24], was an unauthenticated SQL injection vulnerability. A particular function that handled SQL queries did not properly sanitize or validate an input supplied by a user. This meant that an attacker could insert malicious code into this input, which potentially enabled them to extract sensitive information, modify data, or execute arbitrary code.

The solution to this vulnerability was to introduce additional input sanitization before running the user’s query on the database. Sanitizing the inputs ensured that only valid inputs can become part of the SQL query [264].

Cross-Site Scripting Attacks

One of the absolute favorite ways for an attacker to compromise a website is to insert their own JavaScript directly into the website. I do not mean through a third party, or some other collaborative mechanism. I mean literally into the HTML of the victim site itself. If the attacker can somehow insert their JavaScript *directly into the victim web page*, none of the same-origin policy applies, and the attacker can do almost anything they want to do.

This kind of attack is called a Cross-Site Scripting attack, oddly abbreviated XSS. Using XSS, an attacker can steal data out of the web page, change the behavior of the web page, and potentially use it as a foothold for further infiltration. An XSS attack is more powerful than a CSRF attack as anything that could be done by CSRF can be done by XSS.

Two common approaches used by attackers for getting their JavaScript into a victim web page are *reflection* and *storage*. In a reflection XSS attack, the attacker tricks a victim into submitting a compromised input into the server. The compromised input includes JavaScript somewhere in a way that will get written back out to the web page in the HTTP response. Suppose, for example, that the victim user is logged in to some kind of search engine that is not *sanitizing* its inputs. Whatever is typed into the search is put directly into the page without preprocessing. Imagine if the following URL jumped directly to the search engine with search results for “cat”:

```
http://unsafe_search.com/search?query=cat
```

An attacker uses this information to create a URL to this search engine, but replaces “cat” with JavaScript. Although it can look awkward, the JavaScript can all be put on one line and in a way that can be used with a URL query. Once created, the attacker sends this URL to the victim user via email or social media. If the victim clicks the link, it takes them directly to this page, and, instead of searching for a term like cat, it directly inserts JavaScript into the page and executes it. The attacker has now run their code on the victim’s browser. If the victim is signed in to the search engine, the XSS script can directly read and write to this application without running afoul of the same-origin policy, or CSRF tokens. The code, even though sent by an attacker, is coming from the search engine website from the browser’s perspective. So any queries to the search engine would be the same origin.

The other common approach to XSS is storage. In this version, XSS gets inserted into a database. The XSS will be pulled out and put into any web page that relies on the data. If successful, a storage-based XSS attack is often more flexible and powerful than a reflection attack. For one thing, the attacker can, for example, insert the bad JavaScript into the database from their account. Any account that uses this stored data will be subject to the attack. An easy example of this is social media. If someone can create a post from their account with the evil JavaScript inserted into the database, anyone on social media that reads the post will have their systems compromised when it inserts the “post” into their browser.

The best protections against XSS are the same protections against SQL injection. Inputs need to be sanitized. Anytime this important step is forgotten, attacks are almost sure to follow.

Although this is the best option, some systems insert additional defenses that attempt to mitigate the damage if an XSS attack is successful. For example, many security-sensitive cookies are marked “HTTPOnly.” Cookies marked in this way cannot be read by JavaScript *at all*, same origin or not. They are only included in the HTTP transmissions. By blocking them out of JavaScript operations that can read the web page, even a site compromised with XSS will hopefully not lose the security cookies.

Story Time: Tesla XSS Vulnerabilities

In 2019, a security researcher discovered a stored XSS vulnerability on the Tesla Model 3. While he was working on discovering hacks, he experimented with changing the name of the vehicle to reveal any format string vulnerabilities. He then attempted to place an XSS payload into the input. This did not initially work.

However, during a road trip later on, a rock cracked his windshield, and he used Tesla’s in-app support to set up an appointment to have it repaired.

When he received a message about the issue that someone was looking into it, he was notified that the XSS payload was executed. The XSS payload had fired on a dashboard used for managing Tesla vehicles. In his analysis, he concluded that an attacker could pull and modify information about other cars using this vulnerability.

For finding this vulnerability, he was awarded $10,000 as part of Tesla’s bug bounty program [87]. Many companies offer “bug bounties” to incentivize security researchers to find and ethically report vulnerabilities in their software and products. This often enables companies to fix such defects before they become publicly disclosed and thus widely exploited.

Web Application Firewalls

A Web Application Firewall (WAF) is a type of filtering system that is used to protect web servers from malicious inputs. A WAF can be set up as a reverse proxy, as discussed in Chapter 8. A reverse proxy, by way of reminder, is a proxy that acts in the place of the server. It receives the inputs, just like the server would, but then scans them for threats. If the input is considered safe, it is passed on to the real server. A WAF can also be configured as a bridge, at a router, or a plug-in for the web server [218].

WAFs are a special kind of firewall and typically run separately from a traditional firewall. A WAF is designed to scan web traffic for the kind of threats that might indicate some of the attacks described in this section, including XSS and SQL injection.

An effective WAF uses a suite of tools for the inspection and processing of data. For example, one tool is called *normalization*. Attackers will often transform data into a format that is hard to inspect. This is usually done in a way that will be reversed into usable information at some point before it hits the target component of the system. Or, it is sometimes a format that will be understood by a computer, but difficult to recognize as an attack signature. Normalization is the process for taking data in various encodings and converting them to a standard encoding. Once converted, normal filtering rules work much better and can be defined more easily.

Like a regular firewall, WAFs typically have to use both signatures and heuristics. Signatures look for known bad sequences, although the sequences can be expressed in forms like *regular expressions* that match patterns rather than exact sequences of bytes. Heuristics can combine various rules together to attempt to identify dangerous traffic.

Unlike a regular firewall, WAFs are often configured to *allow by default* and only block traffic that is identified as irregular. Some WAFs are set up to block by default and only permit known good traffic, but this is a challenge because of the extreme variety in the kind of traffic permitted.

WAF is often considered an important security feature for security-sensitive web applications. Even though many of the defenses could be inserted into the web application itself, by putting them in a firewall, the defenses can be specialized. The WAF can be updated with new signatures and rules regularly without the need to update each and every web application.

Secondly, the protections are centralized. This enables uniform security policies for all web applications within the organization. It also means that threats can be blocked before they reach the actual web servers.

And finally, the WAF provides defense in depth. A security failure at the web application may be mitigated or blocked by the security of the WAF.

Summary

The World Wide Web is built on top of the fundamental networking technologies of the Internet and began as a way to semantically link together documents, called web pages. Web pages can be identified with a Uniform Resource Identifier (URI) and can be retrieved from the web servers hosting them using HyperText Transfer Protocol (HTTP). The standard language for structuring web pages is HyperText Markup Language (HTML), which provides many semantic and formatting tags. Web browsers enable a user to view these web pages and easily traverse the links between them.

Although HTTP is a stateless protocol, meaning that it did not originally keep track of a user across their time using a website, other technologies like cookies have allowed for the creation of web applications where users can interact with databases, companies, and other users in ways that merely serving web pages did not allow. You are no doubt familiar with many web applications, from banking to travel booking to social networking.

A critical component of web applications is the ability to securely exchange data. HTTPS uses Transport Layer Security (TLS) to communicate over HTTP using encryption to keep data secret and trust the identities of the communicating parties.

Web applications also rely on interactivity. JavaScript is a programming language that runs in browsers to allow users to interact with web applications in rich, responsive ways. Scripting on the browser or even on the server allows creating dynamic applications that use data from databases and APIs to provide everything ranging from up-to-date bank balances to the most recent stock market prices to user-generated content like social media posts.

OAuth is an authorization framework that enables one application to trust a separate application to vouch for the identity of a user, also called Single Sign-On (SSO).

As always, these systems are subject to various forms of attack, and we discussed several of them in this chapter. TLS visibility allows an (ostensibly) authorized third party to decrypt a user's TLS traffic, a sort of man-in-the-middle attack. Cookies have various protections built in to prevent disclosure of sensitive site-specific information to other websites. But advertisers employ many ways of tracking users, from third-party cookies to tracking pixels to browser fingerprinting. JavaScript may also present an avenue for attack, through Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS). Sites that use SQL databases must protect against SQL injection attacks by properly sanitizing and validating any user inputs before passing them to the database. Finally, a Web Application Firewall (WAF) can be used to protect applications from many of these kinds of attacks.

Further Reading

There is a lot to know about the Web, and there are many directions you might explore. I mentioned Peterson and Davie's book *Computer Networks* in Chapter 8. It includes a chapter on applications including the Web and web applications [205, Chapter 9]. But the whole book, with its deep background into how networks work, is helpful to understanding how the Web works.

I always recommend OWASP for web security guidance. OWASP, or the Open Worldwide Application Security Project, is focused on all software security, but I believe that web security is one of their most important contributions. The OWASP Top Ten, for example, refers to the “Top 10 Web Application Security Risks.” This list, which is updated every year, identifies the top ten reasons web applications are compromised [5].

OWASP also provides a range of guides, assessments, and tools for improving web security. OWASP’s Software Assurance Maturity Model is used to measure and quantify the security readiness of an organization’s software [8]. They also provide a Security Knowledge Framework for helping to train engineers in secure software development practices [4]. Another important project is their testing guide for web security testing [10].

In terms of securing web servers (rather than just the software running on a web server), Apache, which develops an open source web server of the same name, provides a guide on hardening the TLS/HTTPS configuration [9]. More broadly, NIST provides SP800-44 “Guidelines on Securing Public Web Servers.” This document is fine as far as it goes, but it was written in 2007 and there have been no subsequent revisions [265].

Part of the problem is that web server security is now typically related to cloud security. Many organizations no longer run a web server on-prem, but instead run their web server in the cloud. For this reason, cloud security books may be a good place to look for more information on this topic. For example, I already mentioned *Practical Cloud Security* by Dotson in Chapter 6. This book also provides some very practical advice on understanding the security relationship between an organization’s systems (including web servers) and the cloud. I recommend reading the section entitled “The Cloud Shared Responsibility Model” as part of your reading [95]. The other cloud security books mentioned along with Dotson, specifically *Securing DevOps: Security in the Cloud* and *Container Security: Fundamental Technology Concepts that Protect Containerized Applications*, may be helpful to securing web servers in cloud environments [217, 272].

I mentioned OAuth and OIDC briefly in this chapter. There is a lot more to OAuth than what I could cover here, and development is ongoing. And there are more SSO technologies than just OAuth. A good book for learning more about OAuth and other SSO technologies (like SAML) is *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2* [283].

In terms of books that provide the offensive point of view, the *Web Application Hacker's Handbook* is considered a classic. Although more than ten years old now, it is still considered worth reading. The book is quite a tome at more than 800 pages, but it breaks down a web application into all the different components: frontend, backend, and even *people*. For each component, it goes into depth about how to attack them [255]. A more recent book is *Real-World Bug Hunting: A Field Guide to Web Hacking*. This book focuses more on specific bugs and how to look for them [287]. If you happen to enjoy the Python programming language, the book *Black Hat Python* is a lot of fun. This book is only partially directed toward web security issues, but it has a whole chapter on exploiting Internet Explorer [237].

I have already mentioned the books *Exploiting Software* and *24 Deadly Sins of Software Security* in other chapters. Both books have web components and are good reading for this chapter as well [135, 137].

References

3. Duo auth API.

4. Owasp security knowledge framework.

5. Owasp top ten.

8. Software assurance maturity model.

9. Ssl/tls strong encryption: How-to.

10. Wstg—stable.

24. CVE-2022-45808. 11 2022.

56. Berners-Lee, T. 1996. WWW: Past, present, and future. *Computer* 29(10): 69–77.
[Crossref]

87. Curry, S. 2019. Cracking my windshield and earning $10,000 on the tesla bug bounty program.

95. Dotzon, C. 2019. *Practical Cloud Security: A Guide for Secure Design and Deployment*. Sebastopol: O'Reilly Media.

108. FingerpringJS, Inc. Frequently asked questions.

129. Hauk, C. 2023. What is browser fingerprinting? How it works and how to stop it. *Pixel Privacy*.

135. Hoglund, G., and G. McGraw. 2004. *Exploiting Software*. Addison-Wesley Professional.

137. Howard, M., D. LeBlanc, and J. Viega. 2009. *24 Deadly Sins of Software Security*. McGraw-Hill.

205. Peterson, L.L., and B.S. Davie. 2021. *Computer Networks*, 6th ed. Morgan Kaufmann.
[zbMATH]

217. Rice, L. 2020. *Container Security: Fundamental Technology Concepts That Protect Containerized Applications*. Sebastopol: O'Reilly Media.

218. Ristic I., et al. 2006. Web application firewall evaluation criteria. Technical report, Web Application Security Consortium.

237. Seitz, J., and T. Arnold. 2021. *Black Hat Python: Python Programming for*

Hackers and Pentesters, 2nd ed. No Starch Press.

255. Stuttard, D., and M. Pinto. 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security*, 2nd ed. Wiley.

264. Toulas, B. 2023. 75k wordpress sites impacted by critical online course plugin flaws. *Bleeping Computer*. www.bleepingcomputer.com/news/security/75k-wordpress-sites-impacted-by-critical-online-course-plugin-flaws/.

265. Tracy, M., W. Jansen, K. Scarfone, and T. Winograd. 2007. Guidelines on securing public web servers. Special Publication (NIST SP) 800-44r2, National Institute of Standards and Technology, Gaithersburg.

272. Vehent, J. 2018. *Securing DevOps: Security in the Cloud*. Shelter Island/New York: Manning Publications Co.

283. Wilson, Y., and A. Hingnikar. 2022. *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2*, 2nd ed. Apress.

284. Wiseman, B. 2017. Page not found: A brief history of the 404 error.

287. Yaworski, P. 2019. *Real-World Bug Hunting: A Field Guide to Web Hacking*. No Starch Press.

Footnotes

1 A URI is technically a broader term than URL, and in some circumstances, the differences might be important. However, in most circumstances, they are used interchangeably.

2 There is also a brief overview of HTTP in Appendix C if you want to review how it interacts with other network protocols such as TCP and IP.

3 There are a couple of minor components that tie the two pieces together. For example, the URL in the HTTP request method must match the *Common Name* in the certificate sent back by the web server. But all of the HTTP request methods (e.g., GET and POST requests) and responses (e.g., 200 and 404 responses) work exactly the same.

4 In many cases, an eavesdropper could figure out the domain from the destination IP address that is unencrypted. However, sometimes more than one host name is associated with an IP address, and this extension leaks which of the host names the client is connecting to.

5 Web developers that work with both the frontend and backend are called *full-stack developers*.

S. J. Nielson, *Discovering Cybersecurity*
https://doi.org/10.1007/978-1-4842-9560-1_10

10. Overlay Security: Email and Social Media

Seth James Nielson[1]
(1) Austin, TX, USA

Chapter Quick Start Guide
The rich infrastructure of the Internet and the World Wide Web allows us to build semantic communications networks that operate at a more abstract level. In this chapter, we'll discuss email and social media, two kinds of overlay networks that allow communications between individuals and organizations. Similar security principles apply here, but because these networks also present unique security challenges.

Key Concepts

1. An overlay network allows communication between nodes and is built on top of a lower-level network.
2. Email and social media are two examples of such networks in which the nodes are people rather than machines.
3. These networks may be subject to threats such as spam, phishing, artificial amplification, disinformation, and reputation attacks.
Some defenses for these threats can be built into the

4. Some defenses for these threats can be built into the network technology, and some defenses rely on the humans on the network.

Common Pitfalls and Misunderstandings

1. Overlay networks often have their own protocols, but they are usually built on top of the widely available lower-level networks that already exist and have broad usage.
2. Social networking sites in particular invite a broad range of threats that are unique to the broad reach and discovery algorithms they employ.

Useful Vocabulary

- **SMTP**: Simple Mail Transfer Protocol. A protocol for sending email messages
- **POP**: Post Office Protocol. A protocol for retrieving email messages
- **IMAP**: Internet Mail Access Protocol. Another (newer) protocol for retrieving email messages
- **Spam**: Unsolicited bulk email
- **Phishing**: An attack that attempts to defraud a user with generic (bulk phishing) or highly targeted (spear phishing) messages
- **S/MIME**: Secure/Multipurpose Internet Mail Extensions. A feature for signing and verifying (and, optionally, encrypting) email messages

---

Computer science loves layers. You may have noticed this already. After all, we have layers of network protocols, layers of host isolation, and layers of security components (i.e., defense in depth). In fact, a famous expression in the field is that, “All problems in computer science can be solved by another level of indirection” [158]. So it will perhaps not be surprising to learn that networks can also be built on top of another network.

In fact, most of the commonly used Internet communications systems can be thought of as an *overlay network*, or a network that is built on top of an existing network. These communications systems include *email* and most *social media*. In this chapter, I will explain how these operate as overlays and how these introduce some interesting security challenges.

Overlay Networking Background

What is an *overlay network*? Quite simply, it is a network built on top of another network. Or, said another way, a new network is built using an old network as lower-level infrastructure. What constitutes a network? In Appendix C, I use the following definition:

> A *network*, as used in this book and in most network security contexts, refers to computing resources, generically identified as “nodes,” that are connected together directly or indirectly across one or more communication media and can engage in intentional data exchange across the media.

When you think about the Internet, for example, you could think of your laptop as a node and the web server for your favorite website as another node. They are connected together via the Internet infrastructure and intentionally engage in data exchange.

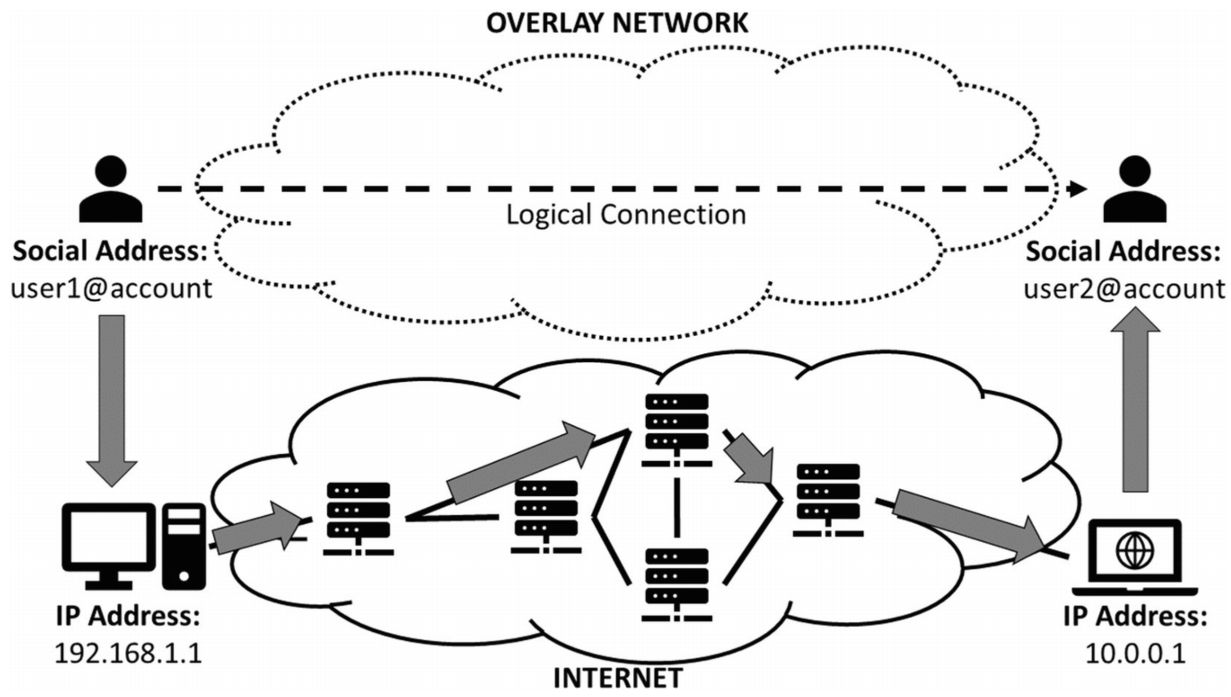


Figure 10-1 In an overlay network, there are logical connections between the higher-level addresses. Data is sent using a lower-level network

No matter how a network is put together, it must usually deal with the following issues:

1. Connecting the nodes to one another
2. A meaningful address scheme that identifies each node
3. A protocol or set of protocols that enable communication from one address to another over the connection mechanism

An overlay network exists when a network is created using another network as the mechanism for the connection between the nodes. It is typically characterized by having its own addressing scheme and means of getting messages delivered between addresses. In addition to the three requirements for a network, it usually also requires some kind of *bootstrap* process in which a new node can be inserted into the overlay network, get an address, and obtain some initial connections to other overlay nodes. Figure 10-1 illustrates these ideas.

An overlay network is not the same thing as the network protocol stack that separates out the functionality and implementation of a network into separate components. For example, even though your laptop has both an IP address and a MAC address, IP communications to a distant web server cannot be carried over just your Local Area Network. Referring again to your laptop and your favorite website, the web server is not on your LAN and cannot communicate with your laptop using the MAC address. Only the IP protocol, with its routing mechanism, can enable the worldwide communications.

On the other hand, the networks we will talk about today, such as email and social media, have a fully working and fully operational network as a starting point. From that starting point, they create a new addressing scheme and a new means of exchanging messages between those addresses.

The reason for creating overlay networks is twofold. The overlay networks typically want to connect at a different conceptual level than the existing network. This is manifest in how the addressing scheme identifies a different type of node in a way that more meaningfully represents a node within the new networking system. But the second reason for an overlay network is to take advantage of the old network's infrastructure. After all, nobody wants to lay down new infrastructure when effective infrastructure already exists. It is much easier to build the new concepts on top of the old ones. Any improvements or advancements in the old network are automatically propagated to the overlay network as well.

Social Networks As Overlay Networks

What is a social network? There are a number of different definitions. The concept of a *social network* predates social networking websites. The nature of human interaction is very much a network. Humans can be modeled as a node, and data, especially ideas, are transmitted from one human to another.

> The most classical definition of a social network is one which is based purely on human interactions. This is the classical study of social networks in the field of sociology. These studies have traditionally been conducted with painstaking and laborious methods for measuring interactions between entities by collecting the actual data about human interactions manually. An example is the six-degrees-of-separation experiment by Milgram... [32, Chapter 1]

Social media sites, such as Facebook and Twitter, “have arisen explicitly in order to model the interactions between different actors” [32, Chapter 1]. In other words, these media sites were designed from the beginning to model the social networks present in human society. One set of authors defined these kinds of sites as having the following characteristics [63]:

1. Web-based services
2. Allow individuals to construct a public or semi-public profile within a bounded system
3. Articulate a list of other users with whom they share a connection
4. View and traverse their list of connections and those made by others within the system

Interestingly, this last definition is explicitly web based, but the term being defined is social media *site*, rather than social network. I find these definitions to be too limited because social media *sites* are just a more specific form of social media networking. I will provide my own definition. A social network is *a network in which the nodes of the network, the addressing scheme, and communications protocols model or enable social, rather than purely computer, interactions*.

Email Operations

Using the definition of social networks I identified in the previous section, email is such a network. Consider that there is an email network that exists above and apart from the Internet. This is the network that connects people sending messages to each other based on email addresses. An email address is not associated with a computer on the Internet but rather with a *person*.[1]

In terms of routing messages to these human-identifying addresses, notice that there are two different levels that could be considered. On the one hand, email is based on protocols that define how email messages are sent to email servers for both inbound and outbound directions. The other level is the social level of message routing. Each warrants consideration.

At a technical level, traditional email uses a standard protocol called the Simple Mail Transfer Protocol, or SMTP. SMTP is an application protocol used to route email messages from a sender to a destination. SMTP provides the overlay routing for getting a message to a particular email address destination.

An email transmission typically starts with a Mail User Agent (MUA) transmitting an email message to a Mail Transfer Agent (MTA). The MUA might be a program running on a laptop, such as Microsoft Outlook, or it could be web mail such as Google’s Gmail. Either way, the MUA uses the SMTP protocol to connect to the MTA.[2]

The MTA’s job is to figure out where on the Internet the email needs to be sent to. This involves resolving the email domain. For example, if an email was being sent to `nobody@example.com`,[3] the MTA would need to figure out the mail server for `example.com`. This process typically involves using DNS to look up what are called Mail Exchanger, or *MX*, records. These records, which are again retrieved via DNS, identify which Internet servers are responsible for email communications for the domain in question.

Once the MTA has the MX records, it routes the email to a destination MTA using SMTP. The destination MTA then stores the email message for subsequent retrieval. SMTP does not prescribe how a message is received by the actual destination MUA, and, instead, other protocols provide this service. For example, Post Office Protocol (POP) is a protocol that permits the message to be downloaded (and commonly deleted from the server), while the Internet Mail Access Protocol (IMAP) is a protocol that reads the messages directly from the server. This flow is depicted in Figure 10-2.

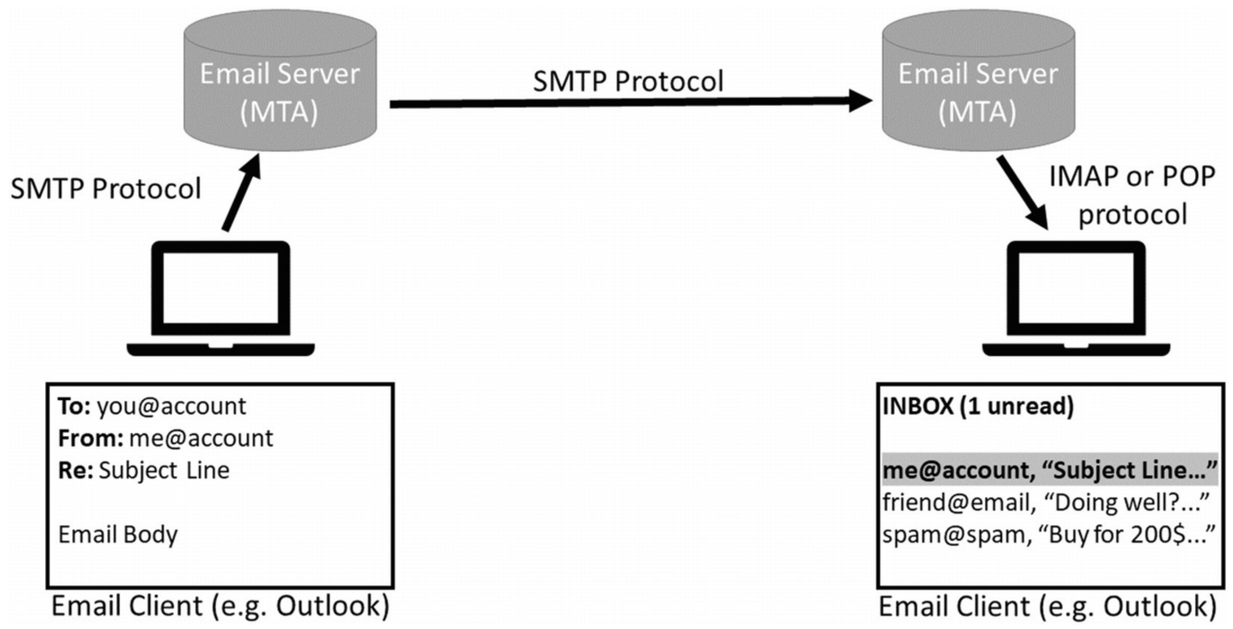


Figure 10-2 When an email is transmitted from one person to another, the message is first sent to the sender's MTA and then to the recipient's MTA using the SMTP protocol. Once at the receiver's MTA, the email is retrieved using a protocol like IMAP or POP

From the perspective of the overlay network, things are straightforward. The Internet provides all the connectivity, and there isn't much in the way of routing. The sending MTA looks up the destination MTA and that is it. Within the context of the overlay network, there is always just one "hop." One of the interesting issues from a security perspective is that the default email protocol has no verification that the sender of the email address is authentic.

When an email is transmitted, there is both the email content and an email envelope. What can be somewhat confusing is that information can be duplicated because the email content includes *headers* that may mirror information in the envelope. By way of analogy, if you write a paper letter, you might put the recipient's name on both the envelope and on the first line of the actual letter (e.g., "Dear John,"). The first line of the letter is roughly analogous to the email headers that are in the content.

Note that the email envelope is only by the SMTP servers. When the email message is downloaded via IMAP or POP, it does not include the envelope. I have depicted the envelope, content, headers, and body in Figure 10-3.

Just like a real letter, however, an email could be addressed to someone completely different from the recipient identified in the envelope. That is, the information in the envelope and the information in the headers do not have to match. In other words, the email address that is shown as the sender in your inbox, which comes from the headers, could be completely different from the sender claimed in the envelope during transmission.

Worse, the email address used in the envelope is also unreliable. So the fact that the sender address in the headers can be different from the envelope is almost irrelevant. *Both* are unreliable and unverified (by default).

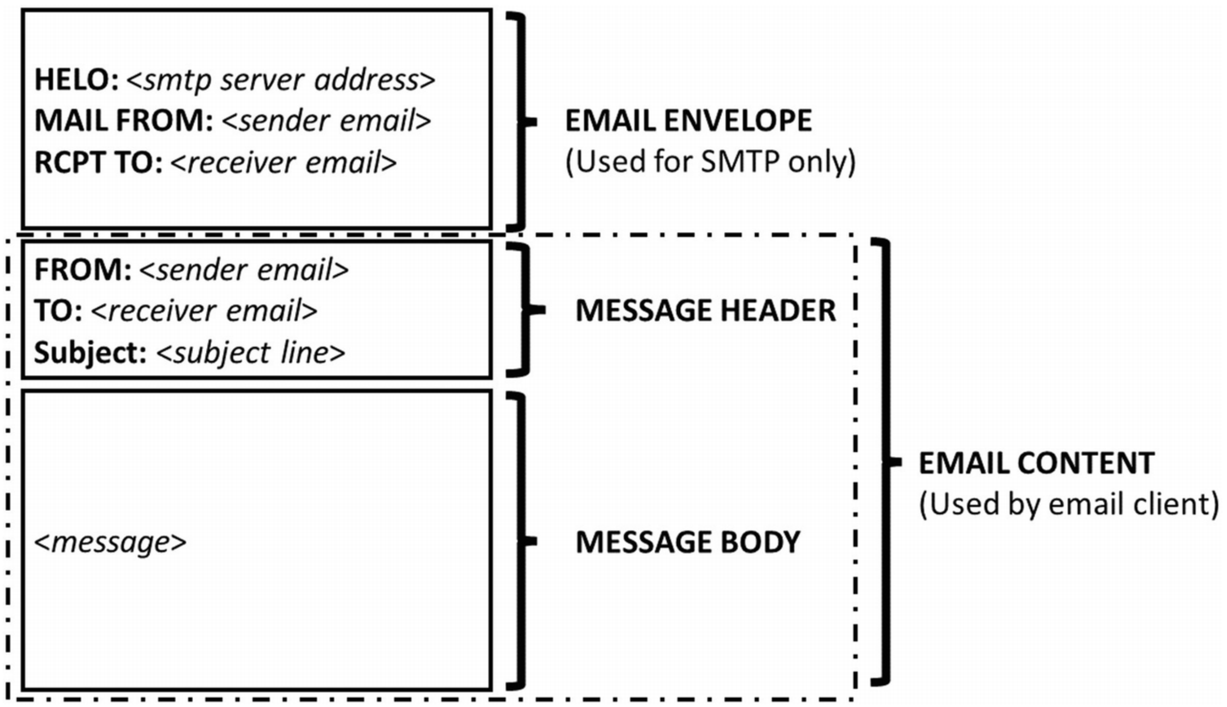


Figure 10-3 Email messages have an envelope, which is used for the SMTP communications, and a content portion. The content portion is what is processed by an email client such as Outlook. The content is further subdivided into headers and body

Perhaps even worse than all of this is that the sender’s address in the headers also includes free-form text that *cannot* be automatically validated. An email address in the headers typically takes the form of `Display Name <email@address>`. Even if the email address itself can be authenticated, the display name cannot. This is because, while there can be indications that a display name is bad, there is no affirmative mechanism for confirming that it is authentic. What would authentic even mean? If someone puts their nickname into a display name, what would be the process for confirming and authenticating that the nickname is real? All of these issues contribute to problems discussed in the forthcoming discussion about security threats.

Fundamentally, SMTP, like other Internet protocols designed before the World Wide Web, was designed for safe networks that had limited access to a relatively small number of users. The design of these protocols simply did not account for the prevalence of bad actors, such as cyber criminals, scammers, hackers, and spammers. Moreover, these protocols were never intended to be used in vast ecommerce enabling systems with tempting targets for cyber criminals, like email addresses of corporate executives and high-value information in corporate networks.

But there is another way of looking at the overlay network. Separately from the technical protocols that enable the communications, there is the overlay at the *social* level. From this perspective, each email address is connected to another email address based on the social ties that connect them. There is a strong correspondence between a person’s social ties and their email address book or contact list. For example, suppose a student starts their freshman year at a university. Starting with some kind of orientation, they are typically provided with an email address at the university. While they may import their old email contacts, such as friends and family, within the context of the new organization, they have very few connections. They probably know very few people at the university. But as they get involved in their classes, social organizations, and professional organizations, their email contact list will grow. Their email social network grows more or less proportionally with their social interactions.

Moreover, unlike the technical email network that is one hop, the social email network is multihop. For example, the student might send a message to a friend inviting them to a party but also encouraging them to bring their own friends. The originator might even ask the recipient to forward the message. And thus, data “routes” through the social network via forwarded emails along social lines.

Because the email social network is associated with *meaning* and not just raw transmissions of unrelated data, I refer to it as the *semantic* network. Both the semantic view of the network and the more technical view of the network have a significant impact on security concerns as discussed later in this chapter.

The semantic nature of social routing is very interesting. Typically, data gets forwarded based on the relationship between the data and the social connection. For example, work data is typically forwarded by work connections. A person might send a colleague a professional document and ask explicitly, or expect implicitly, the document to be forwarded to additional colleagues that the original sender is not directly connected to. An engineer might send a finished design to their manager expecting that the manager might transmit that data to a higher-level manager or review team. Someone in sales might forward a question from a client to a colleague. If the colleague cannot answer the question, the salesperson might request that they forward it along to someone that can.

On the other hand, it might be inappropriate, or even illegal, for an employee to transmit information related to work to a personal contact that is not authorized to receive it. Even if it is not illegal or inappropriate, it may not be socially expected to do so. In Figure 10-4, I have illustrated data being forwarded along a social network. In this figure, a full email need not be forwarded. Perhaps some portion of the text, or even just ideas from it, is spread along the social connections of the initial sender.

Nevertheless, exceptions can and do occur. And, of course, there are emails such as automated emails and notifications. These, however, are associated with a person’s social connections in one form or another. One of the very few types of completely socially unrelated emails a person receives is spam email and other forms of unsolicited communications. One reason why these might bother us so much, besides the wasted time, is the fact that they *are* unrelated to our social connections. Just like a person may appreciate being physically close to a well-known friend or associate but may feel extremely uncomfortable with physical proximity of an unknown person, our negative reactions to spam may be at least partially because of how “alien” they are to our social expectations.

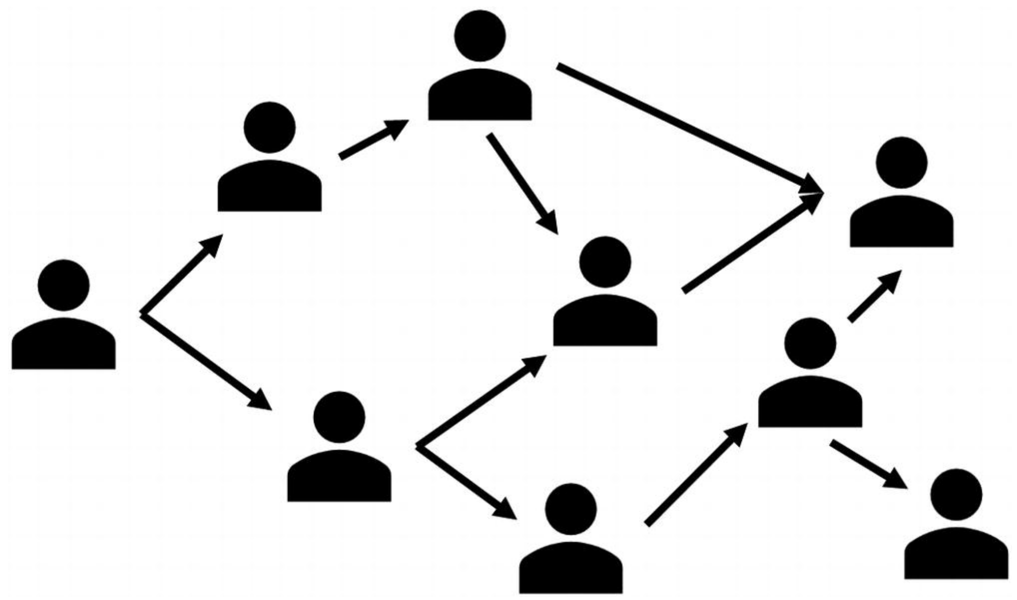


Figure 10-4 Data being forwarded in a social network, such as email

Social Media Sites

Although I have described email as social networking, the term is far more commonly applied to web services that explicitly support creating connections between individuals and social groups. As I noted in the introduction to social networks, it has been observed that all of these various web services tend to support some amount of profile creation, an ability to create and maintain connections to others within the service, and an ability to find new connections.

The concept of *profile* is not limited to the static data that is often referred to as a user’s profile. This information, such as relationship status, name, location, age, occupation, or any other such data, is usually of relatively little importance to the social network. Instead, the user’s profile is reflected in their ongoing generation of *content*, such as posts, images, videos, or other personalized data.

Although many social networking services provide for a direct message capacity that permits one-on-one communications, most social networking interaction is driven by publication of content either to the entire world or to one or more sets of social connections. Service providers support various forms of posting content that permit users of the service to share and manage their social content.

Although I have categorized email as a type of social networking technology, there is an extremely significant difference between most social media sites and email: each social media site is the *property* of some organization, and the operations of the site *are directed to the organization's goals*. This stands in stark contrast to email, which is an open standard. Email users can take their email business to one of a vast number of email service providers or even, if so motivated, operate their own email server on computers under their own control. Users can maintain an independent email address that is independent of operator, and any email provider is interoperable with all other providers (meaning that a user can send an email to *anyone* with an email address, not just those on the same provider).

At least in part because of this difference, there is a fundamental, driving force behind most social media services: *user engagement*.

This driving force appears to be necessary for a social network provider's survival. As noted already, every social media site has more or less the same capability to make and maintain social links or connections to others, and every social media site could enable the same kinds of content to be posted. But it is only through continuous and ongoing engagements with their users that social networking providers maintain their relevance. Even if Alice is connected to Bob using one social network provider, there's nothing to stop her from also having a social connection with Bob through another provider. These providers will need to compete for Alice's attentions and for her interactions with Bob.

This force, the need to engage users, means the social network provider is motivated to ensure that their users are engaged by this provider's flavor of representation and processing of a person's social network and social activities. One method that social media providers have adopted to maintain and grow their users' engagement is the curation of social media content each user consumes. This shifts a user's interactions with their social network from being driven by other people and organizations within the network to being driven by the network provider. Instead of a person finding out about the ideas, events, and interests of those in their society through interactions and discovery, the provider shapes and puts together a stream that is calculated to be of interest.

That is obviously not to say, of course, that the actions of the provider's users have no influence on the provider. One mechanism that is used in determining the relevance of any given content to *someone* is measuring the relevance of that content to *everyone*. Social networking providers enable their users to interact with the content published by others through reinforcement (e.g., likes and shares) or publishing derivative content (e.g., comments or references within other posts). The popularity of a given content posting influences how likely that content is to be shown to others.

Moreover, studies have shown that users are influenced by the popularity of content and by how that content is perceived by others. Thus, users reinforcing and extending content *amplify* the power and reach of a message. Amplification is achieved in social networks by creating *the perception* of widespread popularity and acceptance. As will be discussed later in this chapter, there are a number of ways to manipulate this perception.

Threats

Because of the nature of social networks, including email, they are a constant source of cybersecurity risk. What makes them so ripe for fraud and malicious activity is the direct connection that they have to people. It is hard enough to secure the traffic that is bound for a computer using a technology like a firewall. But it is harder still to protect a living person, and their mind, from messages that go directly to them. Although I have spoken about computer systems as "endpoints," human users are the true endpoints of communications and operations on the Internet.

Using a semantic, social network, an attacker can induce humans to act, and to act in undesirable ways. Firewalls and other scanning technologies can do some filtering of messages, but more than enough malicious content gets through to human users on a daily basis. Once the human is reading the message, there is no effective firewall. That information is going into the brain, and if the brain is tricked, the human will be exploited.

In the following subsections, I discuss some common attacks that make use of these systems. I will sometimes use the term *fraudster* to refer to the malicious actor instead of the term *attacker*. This is because in most of the malicious activity, fraud is involved in one form or another.

Spam

Although unsolicited bulk email, also known as *spam*, is the most benign of all the threats discussed in this chapter, it is still a significant problem in its own right. Economic researchers in 2012 estimated that spam is costing businesses in the United States $20 billion per year in lost productivity, remediation, and scams [214]. Spam continues to increase; according to the 2019 Symantec

Internet Security Threat Report, approximately 55% of emails received were categorized as spam [16].

You might be surprised that spam makes anyone any money. After all, when you have looked at a spam message for some shady pharmaceuticals, an unrealistic sounding business opportunity, or some very strange online dating suggestions, you may wonder why anyone would ever respond to such an email. But the fraudsters behind spam have numbers on their side. Because sending email is cheap, it can be sent to millions, and maybe even tens of millions, of people. Suppose a spam message reached one million people overall. If even just a tenth of a percent respond to the message, that would get the fraudster 1000 potential victims at almost no cost.

Phishing

One of the most common and effective attacks against email and social networks is *phishing*. A phishing email or social media message is one that attempts to fraudulently induce the recipient into destructive operations such as revealing passwords, transmitting money, or otherwise enabling the malicious sender. Phishing tends to split between two major subcategories: *bulk phishing* and *spear phishing*. As with spam, these attacks work for both email and social media communications, but for convenience I will describe them using email only.

Phishing first began to be a popular form of malicious email during the early 2000s and has been growing ever more prevalent since that time. For many years, the raw rate of phishing email was increasing, and in 2014 the phishing rate had reached 1 in every 965 emails received [12]. However, that overall rate has been decreasing since that time and in 2018 was “just” 1 in 3207 emails [12, 16]. However, the drop in overall numbers should not be seen necessarily as a good sign. As bulk phishing decreases and spear phishing increases, the phishing rate goes down, but the risks go up.

Phishing attacks, especially spear phishing, are such a sore spot for businesses that it is part of the category known as *Business Email Compromise*. By 2017, Symantec reported that “Business email compromise (BEC) scams, which rely on little more than carefully composed spear-phishing emails, continue to cause major losses; more than $3 billion has been stolen in the past three years” [12].

In the following subsections, I will walk through the two different categories of phishing to identify how these attacks operate and how they are enabled.

Bulk Phishing

A bulk phishing attack is typically a mass email (i.e., an email that is sent to many recipients, largely without customization to the intended recipient/victim) that tends to impersonate well-known organizations such as banks, online shopping services, and package delivery companies. The email also will usually have some kind of *call to action* that is some specific behaviors that the fraudster attempts to induce on the part of the victim. For example, if the attacker wants the victim’s online banking login, they may tell the victim that their account is compromised and that they need to log in and change their password. A link is provided that goes to a fraudulent login screen. The call to action in this case is the request for the login change combined with a fraudulent link. Through various calls to action, a victim may be tricked into sending passwords, social security numbers, or bank account numbers under the guise of an authorized information request.

Bulk phishing emails are crafted to appeal to the widest audience possible and then distributed just as widely. Of course, the email contents might only be relevant to a small portion of the recipients. For example, the scammers might send an email purporting to be from one of the large national banks, but only a small percentage of recipients might actually have an account with the bank identified in the email. Regardless, the cost of sending phishing emails is low, and even if an extremely small percentage of recipients send money or sensitive data in response to the scam email, it is still worthwhile for the scammers.

Bulk phishing is enabled by the following components and characteristics:

1. Deceptive sender identification
2. Deceptive website or phone number
3. Deceptive visual appearance of the email
4. Psychologically effective call to action narrative, context, and prompt
5. Large, centralized services with populous user bases

The first of these, the deceptive sender identification, stems from the weakness of identifying the true identity of senders in email or social media communications. In almost all of these technologies, there is a distinct lack of authentication on the part of the sender’s identity. As stated in the background on email, the original protocol has no authentication of the sender whatsoever. There are some approaches for validating a sender’s email, but attackers get around this using an email address that looks very similar. For example, if the address needs to be at “amazon.com”, the attacker may use something like “amazn.com” or “ama.zon.com”. Or, as I also explained previously, the display name can always be whatever the attacker wants. This means there is always a mechanism for a deceptive sender identification in email.

Most social media accounts are just as problematic. Some systems do not validate the name of the account at all, and it is quite easy to create an account that impersonates someone else or simply creates a fake individual out of whole cloth. Even for social media systems that have a verification option (e.g., “Twitter Blue” for Twitter), that does not get rid of *unverified* accounts. Much like the display name of an email, even an unverified account with a convincing display name will trick some users under the right circumstances.

The second issue that enables bulk phishing is a deceptive contact point such as a website or a phone number. While some phishing scams work with email responses, many include a link that is the focus of the call to action. When clicked, the link will typically load a web page that *looks* authentic. It will visually have all of the correct images, fonts, and color schemes for the web page being impersonated. However, for most websites these days, the authentic version will have a URL that cannot be forged. Recall from Chapter 9 that if the website is using HTTPS, only the holder of the private key that matches the website’s certificate can claim the URL. Unless the website has been hacked in a way that the attacker has stolen their private key, or if the Certificate Authority has been hacked in a way that permits fake certificates, the fraudster cannot impersonate the real URL.

Attackers get around this in clever ways. One way of doing this is to use a URL that looks almost the same. Just like the fake email addresses discussed earlier, an attacker can use a fake website name that looks very close to the original. One notorious example is the compromise of John Podesta’s Gmail account prior to the 2016 US election. The phishing email redirected him to this URL: http://myaccount.google.com-securitysettingpage.tk. To the untrained eye, this might look like the domain name for this address is google.com. In fact, the real domain name is com-securitysettingpage.tk.

The domain of a host is, for all intents and purposes, calculated as the last two components of the URL, where each component is separated by a period. The last component is called the top-level domain or TLD. The second to last part is a second-level domain or SLD. Generally speaking, these are the two parts that matter in identifying the owner or operator of a website. When a browser shows a lock icon, meaning that the page verified for HTTPS, the certificate must match at least these two parts. All of the other parts that precede the SLD identify *subdomains*. In terms of identification, the subdomains are almost always irrelevant.

But to the average user, what they see in this fraudulent email is what comes first (myaccount.google.com) instead of what comes second. Moreover, most such users will have no idea that the dash is not separating anything in the URL. This and other tricks are used to convince users that they have reached the authentic website.

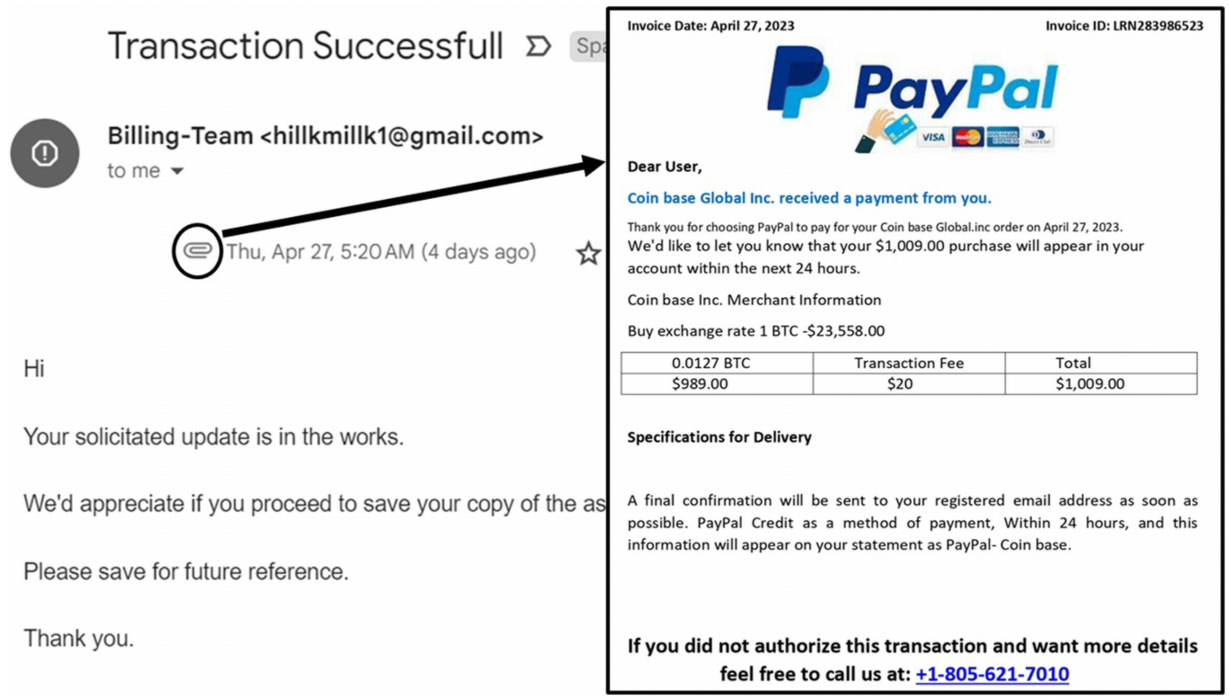
The third trick to a phishing email was alluded to already with the fake web page: convincing visuals. As discussed in Chapter 1, humans tend to respond to visual images. They may even include emotional pathways through our brains that have a more convincing impact than most other forms of expression. When a user sees an email that has all the right visual stimuli, the *normal* response for the brain is to accept it as real. Security training forces us to *reject* those feelings and impressions from what we see; but it does not get rid of them. This is especially sad because it is *easy* for fraudsters to fake the visual elements.

Fourth, the attackers use psychologically effective calls to action. This is usually done by creating a sense of urgency, which feeds into *emotional fallback* and *bias* *toward action* discussed in Chapter 1. There are an almost limitless number of approaches for creating urgency in the human mind. I find some of the approaches *that make no explicit call to action* the most interesting. In these phishing attacks, instead of asking the user to do something, they *let the user create the urgency for themselves*. This is typically done with telling the user, almost as just an update, that some transaction (usually involving some amount of money) has just taken place. For example, it might say something like, “Your transaction at Amazon for $2,000 has just been approved.” Sometimes, the emails include a link that offers to let them view the transaction, or report if the transaction is fraudulent, but some include a link to (fraudulent) account access without saying anything at all. The user freaks out over a transaction they do not recognize and clicks on the fraudulent link to “fix” the problem.[4] In Figure 10-5, you can see one such email that I received recently.

Finally, the last requirement for bulk phishing to work has nothing to do with the attackers at all. It is simply the nature of modern society. Like spam, the vast majority of recipients *will not* fall for a phishing email. In order to be successful, the email has to be sent to a large number of potential victims. The problem is the phishing email also has to be applicable to a large number of potential victims. If the vast majority of people banked at small, community banks, banking would be less desirable as a phishing target and narrative. It would be unlikely that sending out a phishing email would reach a population big enough to support it. But with large national (and international) banks that have massive customer bases, the economics of a bulk phishing attack against banking becomes more reasonable. Similarly, institutions like Amazon, Netflix, and PayPal provide similar opportunities.

Spear Phishing

Another form of phishing is known as *spear phishing*. In contrast to bulk phishing, spear phishing attacks are specialized and targeted to the intended recipient/victim. Spear phishing is typically directed at just one person, using their name and any other personal or identifying information the scammer was able to access in order to craft a more legitimate-appearing email. These emails are particularly nasty because the more information the scammer uses about the individual, the more legitimate the email will appear to be. For example, a scammer with information about where an individual works and what their role is (e.g., obtained from the individual's social media accounts) might be able to craft an email that appears to come from that individual's boss, asking for sensitive corporate information, such as bank accounts or other private data.



Transaction Successfull

Billing-Team <hillkmillk1@gmail.com>
to me

Thu, Apr 27, 5:20 AM (4 days ago)

Hi

Your solicitated update is in the works.

We'd appreciate if you proceed to save your copy of the as

Please save for future reference.

Thank you.

Invoice Date: April 27, 2023 Invoice ID: LRN283986523

PayPal

Dear User,

Coin base Global Inc. received a payment from you.

Thank you for choosing PayPal to pay for your Coin base Global.inc order on April 27, 2023.
We'd like to let you know that your $1,009.00 purchase will appear in your account within the next 24 hours.

Coin base Inc. Merchant Information

Buy exchange rate 1 BTC -$23,558.00

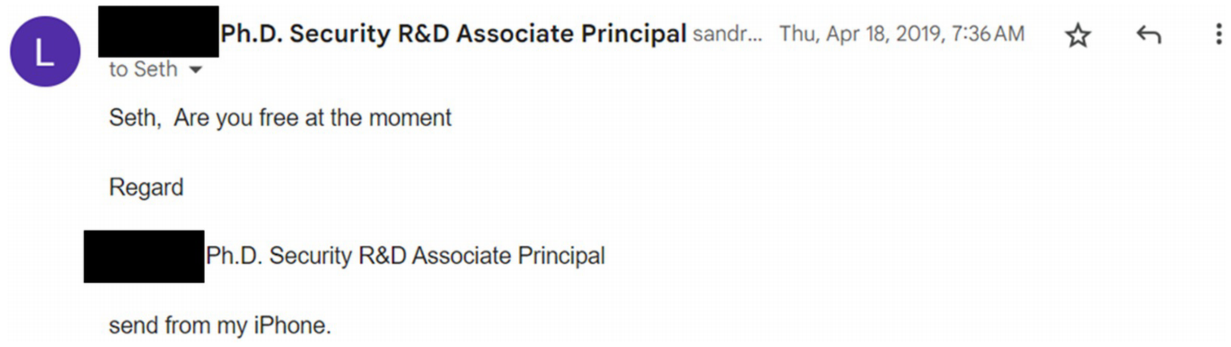
0.0127 BTC	Transaction Fee	Total
$989.00	$20	$1,009.00

Specifications for Delivery

A final confirmation will be sent to your registered email address as soon as possible. PayPal Credit as a method of payment, Within 24 hours, and this information will appear on your statement as PayPal- Coin base.

If you did not authorize this transaction and want more details feel free to call us at: +1-805-621-7010

Figure 10-5 A phishing email that does not specifically tell the victim to contact them


L [redacted] Ph.D. Security R&D Associate Principal sandr... Thu, Apr 18, 2019, 7:36 AM

to Seth

Seth, Are you free at the moment

Regard

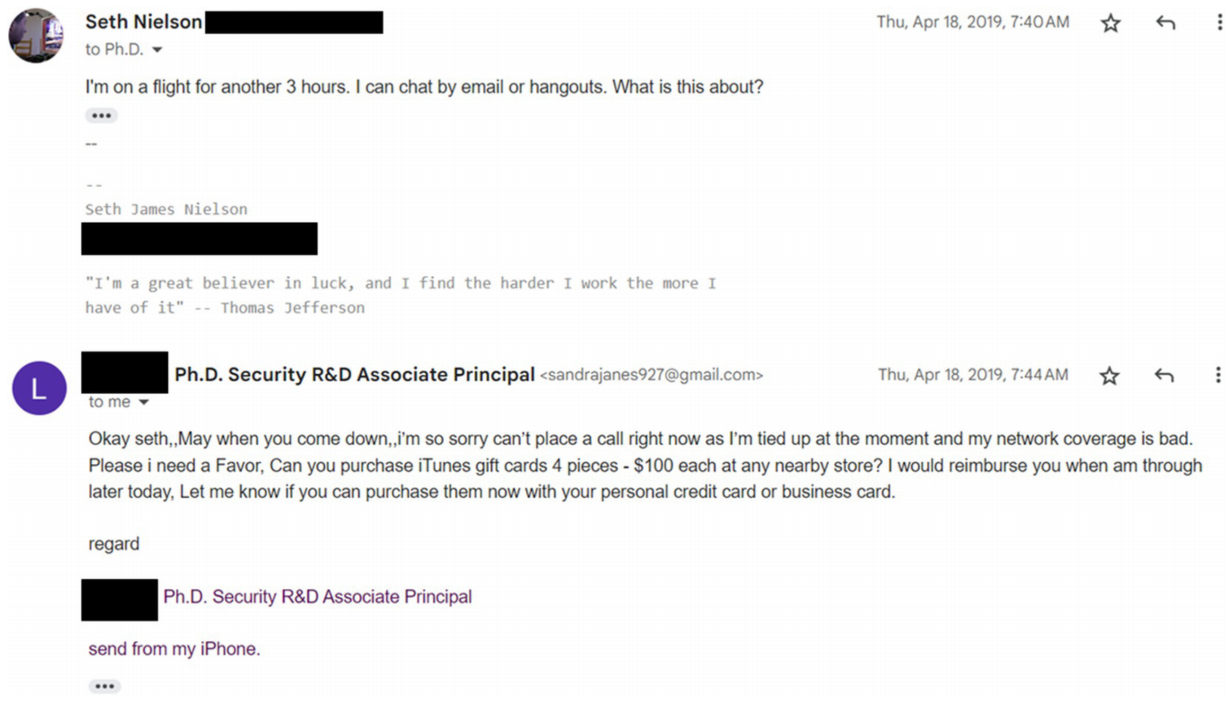
[redacted] Ph.D. Security R&D Associate Principal

send from my iPhone.

Figure 10-6 A short and somewhat cryptic request I received

By way of example, Figure 10-6 is a sample spear phishing email that I received in April of 2019.

As you can see, the email is suspicious, but it can be difficult to positively identify this email as malicious. Although I didn't personally know Lei Ding, Dr. Ding is another adjunct faculty member at Johns Hopkins University, where I taught at the time. It was at least possible that Lei needed to speak with me. Figure 10-7 is the follow-up exchange.


Seth Nielson [redacted] Thu, Apr 18, 2019, 7:40 AM

to Ph.D.

I'm on a flight for another 3 hours. I can chat by email or hangouts. What is this about?

--

--

Seth James Nielson

[redacted]

"I'm a great believer in luck, and I find the harder I work the more I have of it" -- Thomas Jefferson

L [redacted] Ph.D. Security R&D Associate Principal <sandrajanes927@gmail.com> Thu, Apr 18, 2019, 7:44 AM

to me

Okay seth,,May when you come down,,i'm so sorry can't place a call right now as I'm tied up at the moment and my network coverage is bad. Please i need a Favor, Can you purchase iTunes gift cards 4 pieces - $100 each at any nearby store? I would reimburse you when am through later today, Let me know if you can purchase them now with your personal credit card or business card.

regard

[redacted] Ph.D. Security R&D Associate Principal

send from my iPhone.

Figure 10-7 The next exchange was a clear and obvious phish attempt

Now from this follow-up email, it was easier to immediately recognize that the emails purportedly from Dr. Ding were spear phishing attempts. Nobody at Johns Hopkins, or in almost any professional environment, would ever ask for iTunes gift cards under even the most extraordinary of circumstances. By reason of this very blatant money-grab, this deception was not particularly sophisticated. In fact, the entire spear phishing email was not particularly sophisticated. All the scammer knew was that I was at Johns Hopkins and that Lei Ding was at Johns Hopkins. It included no information about my recent work at the University and was not from a person I actually worked or interacted with. However, better targeted spear phishing emails could (and often do) result in the victims providing the requested information in response to what they believe is a legitimate request.

There are a couple of different types or flavors of spear phishing attacks that are worth mentioning briefly.

The first of these are spear phishing that attempt to redirect an existing, usually large, financial transaction. The way these typically work is the fraudster becomes aware that one organization is going to transmit money to another organization. These transactions are usually invoices in one form or another, but sometimes these attacks take the form of a payment *reversal*. In the case of invoices, the attacker waits until just before a payment and then sends an email (as the vendor) telling the payer (victim) that the vendor’s bank accounts have changed and they need to update their information. In the case of a reversal, the victim is told that a payment made needs to be reverted for some reason or another. They might claim it was sent from the wrong account, for example. Payment reversals have been especially problematic in real estate. Money is often wired, for example, to a title company. The title company acts as a kind of clearing house and neutral third party. It is their job to make sure that money goes where it is supposed to (e.g., to paying off a loan) and that the assets are transferred as per the buying and selling agreement. But because money is usually wired in advance of the real estate closing, attackers have been known to contact the real estate company impersonating the buyer and asking for the money to be sent back. There have been many people that have lost large sums of money to this kind of fraud.

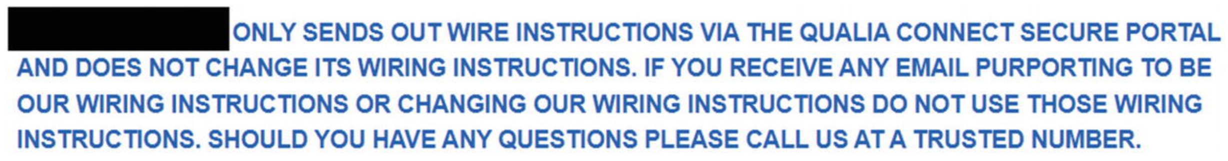

[redacted] ONLY SENDS OUT WIRE INSTRUCTIONS VIA THE QUALIA CONNECT SECURE PORTAL AND DOES NOT CHANGE ITS WIRING INSTRUCTIONS. IF YOU RECEIVE ANY EMAIL PURPORTING TO BE OUR WIRING INSTRUCTIONS OR CHANGING OUR WIRING INSTRUCTIONS DO NOT USE THOSE WIRING INSTRUCTIONS. SHOULD YOU HAVE ANY QUESTIONS PLEASE CALL US AT A TRUSTED NUMBER.

Figure 10-8 This is an example of reducing the risk of financial loss from email fraud. The wiring instructions are only sent via secure communications, and all insecure communications carry a warning not to trust them

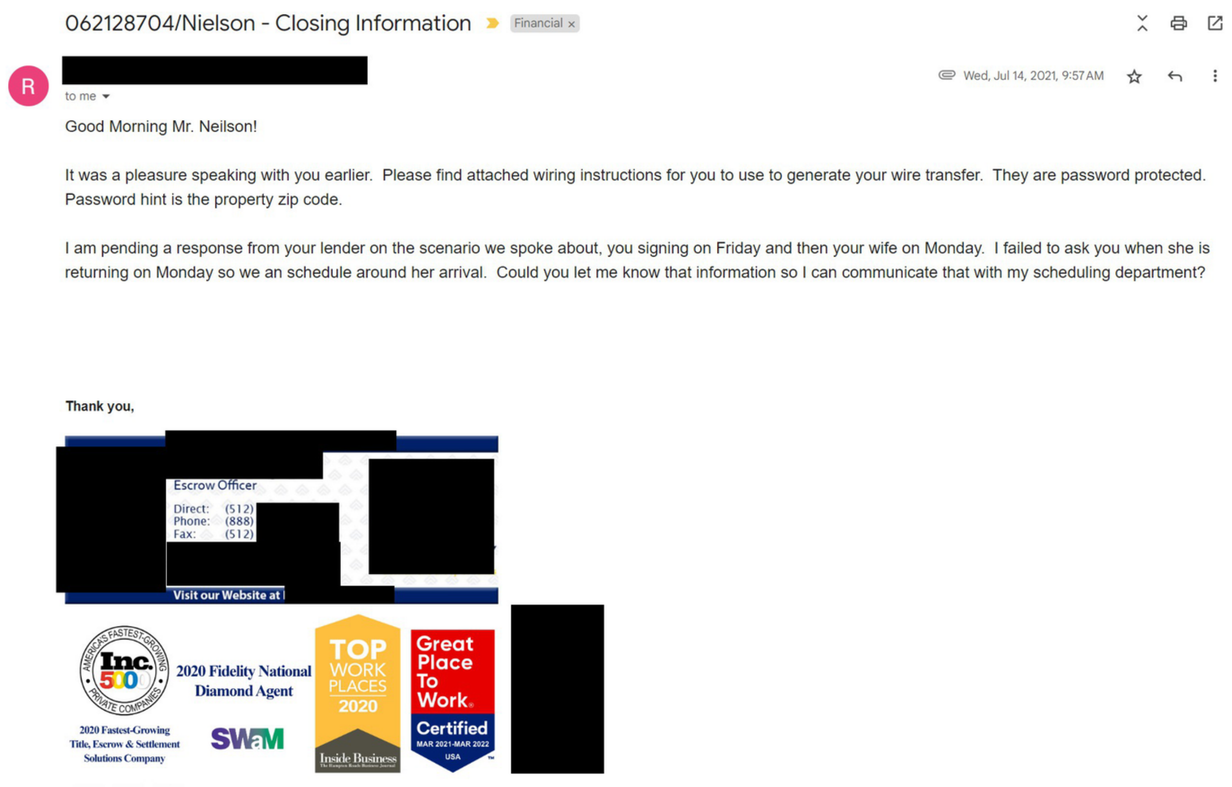
On the flip side, fraudsters have also pretended to be the title company and tricked buyers into sending their money to a fraudulent account. These days, it is best practice to only send wiring instructions via secure communications and have email include warnings about getting wiring instructions via email. For example, as part of a transaction in 2019, I received the warning message from a title company shown in Figure 10-8.

Story Time: Dark Humor Is Like Security. Not Everyone Gets It

Unfortunately, not every title company is following the best practices illustrated in Figure 10-8. In 2021, two years later, I was involved in another transaction wherein I received wire instructions via email. I did not even know who to call to verify, because the only phone number I had was *the phone number in the email*. I finally called that number anyway and told the agent that I was going to have to find a way to verify her. She seemed confused and asked why talking to her on the phone was not verification. I patiently explained that because I got the phone number from an email I could not trust, I could not trust the phone number either. I told her I would accept the phone number as valid if I could find it listed on the title company’s web page provided that the web page was protected by *https*. It took a while to verify, but I was eventually able to find and validate. The scariest part of the entire conversation was when she told me this had never come up before. That is fraud waiting to happen, but they had no awareness of it.

You can see the email sent to me in Figure 10-9. They got my name wrong too (“Neilson” instead of “Nielson”), which made me even more suspicious.

Whether a redirected payment or a refunded deposit, the fraudulent message is usually some form of spear phishing, directed to accounts payable, an executive, or an administrator. These kinds of attacks are particularly worrisome because they often leverage an intrusion that gets them access to some generally nonpublic information and communication channels. While some of these financial transactions can be predicted or guessed by attackers, some occur because attackers break into the email systems of a related party, like the vendor, and have accurate information about the transactions, the current bank accounts, and the date of payment. Moreover, they can send emails that are, in fact, from the correct email account so that the source is authentic even though it is fraudulent.


062128704/Nielson - Closing Information

Financial ×

R [redacted]

to me

Wed, Jul 14, 2021, 9:57 AM

Good Morning Mr. Neilson!

It was a pleasure speaking with you earlier. Please find attached wiring instructions for you to use to generate your wire transfer. They are password protected. Password hint is the property zip code.

I am pending a response from your lender on the scenario we spoke about, you signing on Friday and then your wife on Monday. I failed to ask you when she is returning on Monday so we an schedule around her arrival. Could you let me know that information so I can communicate that with my scheduling department?

Thank you,

[redacted]

Escrow Officer

Direct: (512) [redacted]
Phone: (888)
Fax: (512)

[redacted]

Visit our Website at [redacted]

2020 Fastest-Growing Title, Escrow & Settlement Solutions Company

2020 Fidelity National Diamond Agent

TOP WORK PLACES 2020

Great Place To Work. Certified MAR 2021-MAR 2022 USA

[redacted]

Figure 10-9 An email with wiring instructions. This is not safe and cannot be trusted, nor can the phone numbers in the virtual business card

A slightly different flavor of spear phishing is one that simply seeks to submit an unauthorized invoice for payment. These kinds of attacks take advantage of how often and in how many ways invoices get introduced at even relatively small firms. Invoices can come in from all over an organization. Each department usually has its own purchasing authority for the relevant vendors. Some attackers can simply try to send an invoice directly to an accounts payable department (which often have “helpful” email addresses like `ap@company.name`) with hopes that the department is too busy to validate and verify it. But more often, phishing emails will be sent to people within the company with messages like, “We got this invoice, do you know who should handle it?” or “This one is important so can it get expedited today?”

There is a variant of this kind of scam that involves trying to harvest personal information instead of money or passwords. The basic idea is to steal something like a social security numbers and payroll data in order to file fraudulent tax returns. Or, the attackers could simply steal data that could be used for a more convincing and realistic phishing attack later. These attacks can seem very convincing and, because they are just asking for personnel records, some victims have had their guards down. An official warning from an Ohio government tax agency reported the following [7]:

> The W-2 scam first appeared last year. Cybercriminals tricked payroll and human resource officials into disclosing employee names, SSNs and income information. The thieves then attempted to file fraudulent tax returns for tax refunds.
>
> This phishing variation is known as a "spoofing" e-mail. It will contain, for example, the actual name of the company chief executive officer. In this variation, the "CEO" sends an email to a company payroll office or human resource employee and requests a list of employees and information including SSNs. The following are some of the details that may be contained in the emails:
>
> - Kindly send me the individual 2016 W-2 (PDF) and earnings summary of all W-2 of our company staff for a quick review.
> - Can you send me the updated list of employees with full details (Name, Social Security Number, Date of Birth, Home Address, Salary).
> - I want you to send me the list of W-2 copy of employees wage and tax statement for 2016, I need them in PDF file type, you can send it as an

> attachment. Kindly prepare the lists and email them to me asap.

The final flavor of spear phishing for this chapter is sometimes called *whaling*. This kind of spear phishing is for specifically going after the “biggest fish” (i.e., the “whales”). Generally, these go after the C-suite, and the CEO in particular. There are a number of reasons these individuals make good targets. First of all, they do have a lot of authority to disperse funds or have credentials to important systems. But more importantly, they are amenable to these kinds of frauds if for no other reason than how busy they are. A CEO’s schedule is typically intense and crowded. An email about getting something like a bill paid is an annoyance and is often shifted to someone else. But if the CEO forwards it, the recipient may think that the CEO has confirmed that the email is authentic. Worse, many CEOs and other top executives have assistants with access to their email and calendar and who respond on behalf of the CEO for certain kinds of tasks. Attackers can exploit this relationship by crafting emails that speak to the assistant’s desire to keep from putting more on the CEO’s plate. Even if there is some suspicion, some assistants have not wanted to disturb the CEO by asking about it.

You should be aware that when fraud is perpetrated on a company over email, it is often called a *Business Email Compromise* or BEC. Many security products are marketed as defenses against BEC, and these typically take the form of phishing defenses.

Story Time: Paranoia Is Good

Attackers combine various techniques into a coordinated operation. A friend of mine is the CEO of a small medical technologies company. He told me a story of some spear phishing that, unfortunately, cost his company tens of thousands of dollars.

According to my friend, one of his people received an email from a vendor shortly before a big payment was due. The email asked to change the ACH details for the payment. My friend was immediately suspicious. The email appeared to be authentic, at least in terms of the email address. It certainly came from the correct email servers. Still, suspecting something was up, my friend asked his employee to *call* the CFO and get a *verbal* confirmation that the new ACH information was correct. Some time later, the employee reported back that the CFO had, in fact, confirmed the new data.

Surprised, but trusting his employee, my friend changed the payment data. The wire went through, but a few days later, the vendor called them and asked why they had not been paid. My friend immediately knew what was up. He called the vendor's bank and demanded a freeze on the transaction. That is an interesting story all by itself but for another time. He immediately asked his employee if he had, in fact, called and talked to the CFO directly.

It turned out he *had not*. He had called, but only gotten to the voicemail. After leaving the voicemail, he got an email assuring him that the information was correct. He never spoke to the CFO directly nor did he get a verbal confirmation from anyone.

With some phone calls to the vendor's IT office, my friend managed to piece together what happened. They figured out that the attackers had compromised the vendor's machines. The attackers had access to the vendor's email, which is why the spear phishing came from "real" email addresses. Moreover, the vendor has voicemail transcription. They saw a transcript of the voicemail left for the CFO and responded to it by email.

After the vendor’s IT became aware of the attack, they started remediation and intrusion response. But the attackers knew they were about to get kicked off the systems. Immediately they sent out a flurry of emails to everyone they could, including my friend’s company, with malware. Amazingly, the same employee that had been fooled by the phishing email and voicemail interception *also* fell for the malware-laden email. My friend had to run down to his desk and stop him from getting their networks infected.

My friend also asked this employee why he did not follow instructions and do as he was told. Why had he not gotten a *verbal* confirmation of the ACH changes? His reply was, “I thought you were being paranoid.” Based on his lack of appreciation for the risks of phishing and malware in emails, the employee was given the opportunity to resign. It was just too many strikes and too much damage.

Let me draw your attention, however, to the bigger point of this story. The attackers combined all kinds of approaches to these attacks. They compromised email servers, engaged in targeted spear phishing, and tried to launch malware upon discovery. The really dangerous attackers have more than one trick up their sleeves and are skilled at coordinating offensive operations using many tools and approaches.

Artificial Amplification and Disinformation

Email communication is most commonly one-to-one or one-to-few. It usually occurs between parties that already know of each other’s existence and identity, as they may have a personal or business relationship. Social media, on the other hand, often focuses on one-to-many communication styles, in which users publish content not just for the people they are directly connected to but also the friends of their friends and even the public at large. Because of the volume of information being created in this way, and because of the business incentives of the platforms on which the content is created, unique problems arise from the way the information gets discovered, shared, and amplified.

As discussed previously, the social networking provider's ability to permit users to follow one another, retweet and like tweets, and other meta-information services is used by individuals to indicate interest, preferred content, and even agreement. In aggregate over millions of users, this can lead to mass activity that influences advertising, social ideas, and government elections. Although each individual person using Twitter can express their own individual influence, there appears to be powerful influence of communities within social networks to influence the individual. Accordingly, parties that want influence are motivated to harness social network influence over others in order to achieve their objectives [81].

One of the means of influence on digital social networks, considered by myself and others in the security community to be malicious, is the use of "fake" participants to influence the thinking and opinions of "real" participants. By "real" and "fake," I am respectively referring to social network accounts that either more or less represent the parties that are using them and accounts that are intentionally dishonest about the party operating the account, the beliefs and motivations of said party, or any other characteristic that may be perceived as having an impact on social influence.

One of the most problematic challenges for social networking accounts is recognizing *sock puppets*. The concept of a sock puppet is basically wherein a single party represents to others that they are, in fact, multiple parties.

Although the concept predates modern technology, the nature of digital communications over the Internet makes it a much more notorious problem. In terms of social media influence, a single party can appear to be a much larger group. Thus, the “beliefs, judgments, and actions of others” can be disproportionally influenced. It is widely believed that “[e]stablishing the identity of online personas would assist in adding meaning and credibility to social media discourse” [81].

Story Time: Sock Puppets in Financial Markets
This “sock puppet” problem manifests in other realms, such as financial markets, where the investing public needs to know whether large blocks of financial investments are controlled by related entities. In these situations, laws can be devised to protect public trust in the integrity of the market and to compel disclosure of centralized control over multiple entities.

For example, investment firms controlling at least $100 million of publicly traded securities are required to disclose their holdings quarterly to the Securities and Exchange Commission (SEC) on Form 13F. These forms are made public so that investors can understand large market players and which investments they hold, since large players with concentrated investments may influence companies in ways that large numbers of smaller, unrelated investors may not.

One particular investment firm owned $32 billion of publicly traded securities. It divided those investments up among 13 (ostensibly unrelated) shell companies, each of which filed the mandated reports and indicated that the company had investment discretion over the funds in its individual portfolio.

However, an SEC investigation uncovered that the shell companies were not in fact operating independently, and investment decisions were made centrally by the parent company. The SEC imposed a fine on the company of $5 million for these misstated filings [236].

It is, however, difficult to solve the sock puppet problem. This problem has appeared in other research. For example, during the late 1990s and 2000s, there was research and development invested into completely decentralized systems. This research was at least one of various influences behind blockchain and cryptocurrencies. One of the major research challenges investigated during this period was how to deal with participants that claimed multiple identities and was known as the *Sybil Attack*. It was noted at the time (approximately 2002) that “it is practically impossible, in a distributed computing environment, for initially unknown remote computing elements to present convincingly distinct identities” [96].

To indicate how challenging the problem can be, the general solution used in many cryptocurrencies is to use *proof of work*. This basically requires solving exceptionally challenging mathematical proofs. The theory is that it does no good to “pretend” to be more than one party because the computational resources would also increase. The problem with proof-of-work solutions, however, is that it requires exorbitant amounts of energy that cost money, damage the environment, and provide no value. Moreover, it leads parties with sufficient resources to purchase large amounts of computing hardware, leading to disparity in the social community wherein those with resources have louder “voices” than those with modest resources. From a certain point of view, this reduces to the same problem Sybil was causing in the first place (i.e., that voices were artificially louder than others).

In digital social networks, there are a number of ways to achieve sock puppetry. One approach is to use *botnets*. In a botnet, social media accounts are not directly operated by a person. Instead, one or more automated agents, called bots, interact with social media accounts to direct the account activities. The easiest tasks for automated agents are simple tasks that require minimal intelligence. These kinds of tasks include connecting to another account and reinforcement actions, such as “likes” or reshares. However, bots can also be used to create content. This artificially amplified content can be used for various purposes. The next section applies these approaches to attacking a reputation [61].

However, sock puppetry can also be achieved through so-called *meat puppets*. Meat puppets are accounts run by humans, but the humans are paid-for groups that are social media mercenaries, posting content, resharing content, and connecting to accounts for hire. In a fascinating study:

> Kevin Ashton’s imaginary motivational speaker “Santiago Swallow” legendarily raised up in prominence through the purchase of 90,000 fake followers for the sum of US$50. Ashton created the fake account in less 2 hours, searched the website `fiverr.com` for [people] selling Twitter followers, and generated Santiago Swallow on the 13th of April 2013. He was then able to acquire reports from dependable social media analysts such PeopleBrowsr, who announced that Santiago Swallow had an @Kred (2013) influence score of 754 out of 1000. [81]

A meat puppet is potentially deceptive in two ways. First, the account’s behaviors are deceptive in that they do not represent the real beliefs of the operator.[5] Second, and more problematically, a single voice is still able to exert disproportionate influence. That is, the meat puppet operator can operate many accounts.

One of the uses of amplification is disinformation. Disinformation is a politically charged topic and an incredibly sensitive one. Politically minded individuals of all backgrounds and beliefs are arguing over what is and is not disinformation and whether or not disinformation is impacting public policy, and *whether accusations of disinformation are stifling public debate*. While I have my own opinions on this topic, for the purposes of this book I am going to try to sidestep any political or public policy debates. Instead, I will try to focus on a few technical issues that I hope will be helpful to anyone, regardless of their opinions on any specific example.

The first major point already discussed is the question of artificial amplification. Parties that are motivated to sway public opinion, including politicians, advertisers, and social organizations, will always be incentivized to have the loudest megaphone they can for spreading their message. Whatever ethical constraints they may have personally or professionally are unlikely to inhibit them from using artificial amplification if it is available. To the extent any part of their message is false, exaggerated, or misleading, those parts are disinformation. And the disinformation is enhanced by the artificial amplification.

There is a flip side to artificial amplification: artificial diminishment. Just like voices can be perceived to be more reasonable and persuasive through increased popularity and engagement, voices can also be perceived to be less reasonable and less persuasive if they seem unpopular, less supported by others, and less prevalent. Accusations that social media organizations themselves reduce the reach of certain messages and content raise questions about this kind of activity. The stated goal of such diminishment is to reduce disinformation, but if the diminishment reduces legitimate content, it has, itself, become a form of disinformation.

The second point worth thinking about is the nature of social media and what it enables in terms of manipulation. For the first time in history, people that want to shape the thinking of other people have near instantaneous feedback as to how well their message is working. When the promoters of an idea or message publish it through a social network, they can receive instant feedback in terms of reinforcement actions, such as likes and reshares, or debates such as comments. Based on the feedback, the message can be adapted and adjusted.

The most concerning part of this kind of feedback cycle is that persuasive messages are often meant to influence through psychology manipulations, such as those described in Chapter 1. That kind of manipulation has been happening throughout history for as long as mankind has existed. However, with this kind of modern technology, those attempting psychologically manipulative messages know nearly instantly how effective their approach is on a sample population. It is almost never the case that calm and reasonable thinking or civil and respectful debate will sell a product, elect a supported candidate, or otherwise get the desired result. In other words, the incentives are for the most psychologically manipulative messages as possible with nearly instant feedback on how effective the manipulation was.

Reputation Attacks

One of the greatest risks of social networking is that it opens the digital doors to the worst of human prejudices and mobocracy. In the background on social networks, I noted that amplification is part of the nature of social networks, both the social networks that existed before the computers and the social networks that have been created and expanded using the Internet. Unfortunately, amplification can be exploited as a means of destroying another person or organization’s reputation.

Attacks on a person or organization’s reputation are almost always conducted over social media or other social websites, such as forums and message boards. There are many ways to attack someone’s reputation. One approach simply involves posting negative sentiment on social media and message boards.

One of the easiest ways to attack reputation is with false reviews and other false messages. This can be done with artificial amplification as discussed in the previous section. But it can also be done with strategic content that will snowball and amplify on its own. For example, companies involved in mining or other mineral processing can be attacked with allegations of environmental impropriety regardless of how environmentally friendly they are. An agent placed by a competitor or an unfriendly government on an environmental message board can post fake photos and false reports to get a growing mob of outraged activists reposting and amplifying the false content. These kinds of reputation attacks have led to live protests at the physical sites. With the advent of generative artificial intelligence capable of creating believable fake text and images on behalf of these adversaries, this risk increases even more.

Another way of damaging a reputation is hacking the account of the victim and using it to publish false and slanderous materials. Or, alternatively, the fraudster can impersonate a person or someone that purports to belong to the victim organization. This can be especially bad for a company’s reputation (i.e., brand) if the fraudster uses the impersonation to launch attacks against the victim’s customers.

Defenses

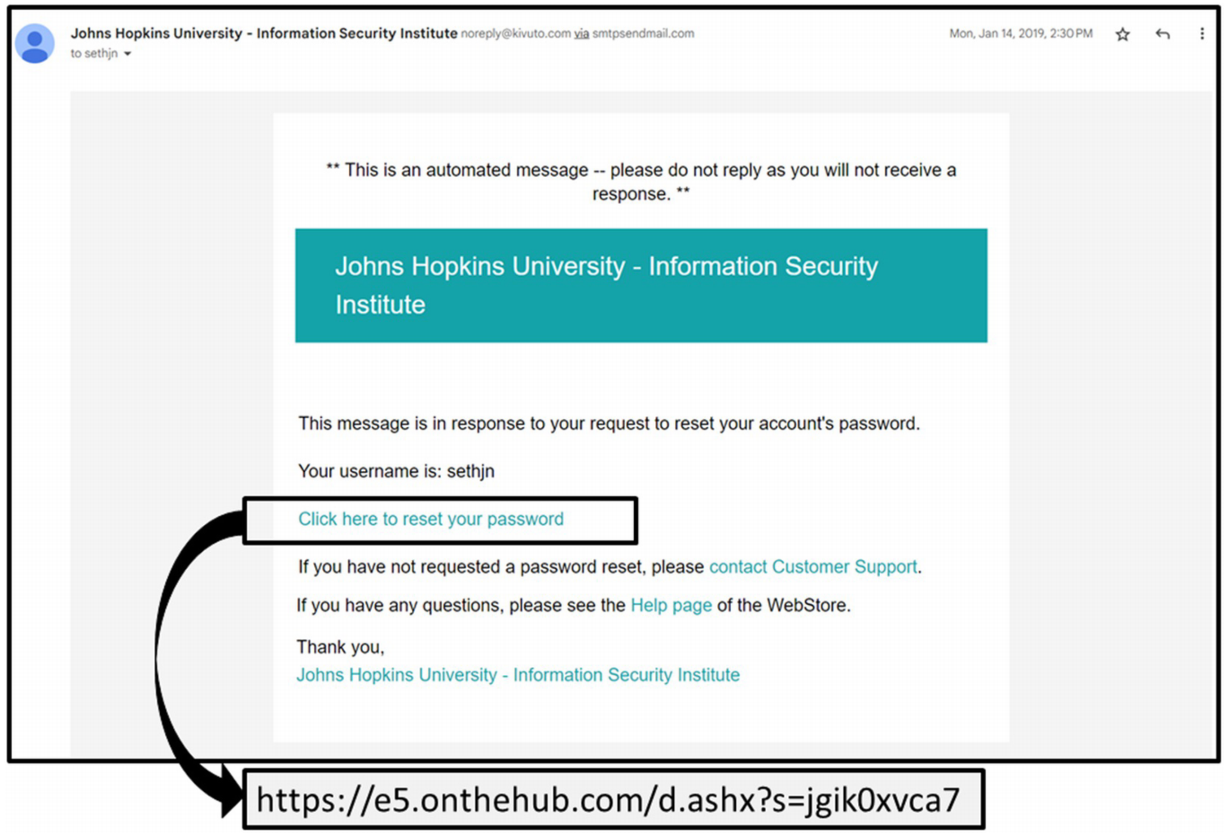
Unfortunately, defending against the threats identified in this chapter is challenging for a number of reasons. First, as you have seen repeatedly in this chapter, the ability to distinguish between a “good” email or social media message and a “bad” one can be very challenging for a computer program. The fraudsters get very good at learning how to make their messages look benign to a defensive system. The short version is: a lot of bad messages are going to get through.

A related problem is that *a lot of good messages get miscategorized*. In the efforts of defensive systems to identify and filter out fraudulent messages, legitimate messages sometimes get miscategorized as malicious or suspicious. As you have seen in other places in this book, identifying a good message as a bad one is called a *false positive*.

This happens with phishing messages. I have seen multiple examples where legitimate messages just look bad. For example, I had an interesting exchange that same year depicted in Figures 10-10 and 10-11.

The legitimate password reset looked very much like a phishing message complete with a vague, but urgent, call to action. Perhaps an even better example was a message that was sent from the FBI to a business executive. I cannot reprint the email, but the message looked like classic phishing. It suggested an urgent need to meet along with vague references to some unknown action overseas. I was asked to evaluate the message on behalf of the executive and was only able to validate that it was, in fact, a legitimate message from the FBI by calling the FBI field office, getting the phone number for the sender, and calling them to verify that they sent it.

The defenses that are available include tools for filtration, tools for message controls, and services for identifying social network misuse.


Johns Hopkins University - Information Security Institute noreply@kivuto.com via smtpsendmail.com

to sethjn

Mon, Jan 14, 2019, 2:30 PM

** This is an automated message -- please do not reply as you will not receive a response. **

Johns Hopkins University - Information Security Institute

This message is in response to your request to reset your account's password.

Your username is: sethjn

Click here to reset your password

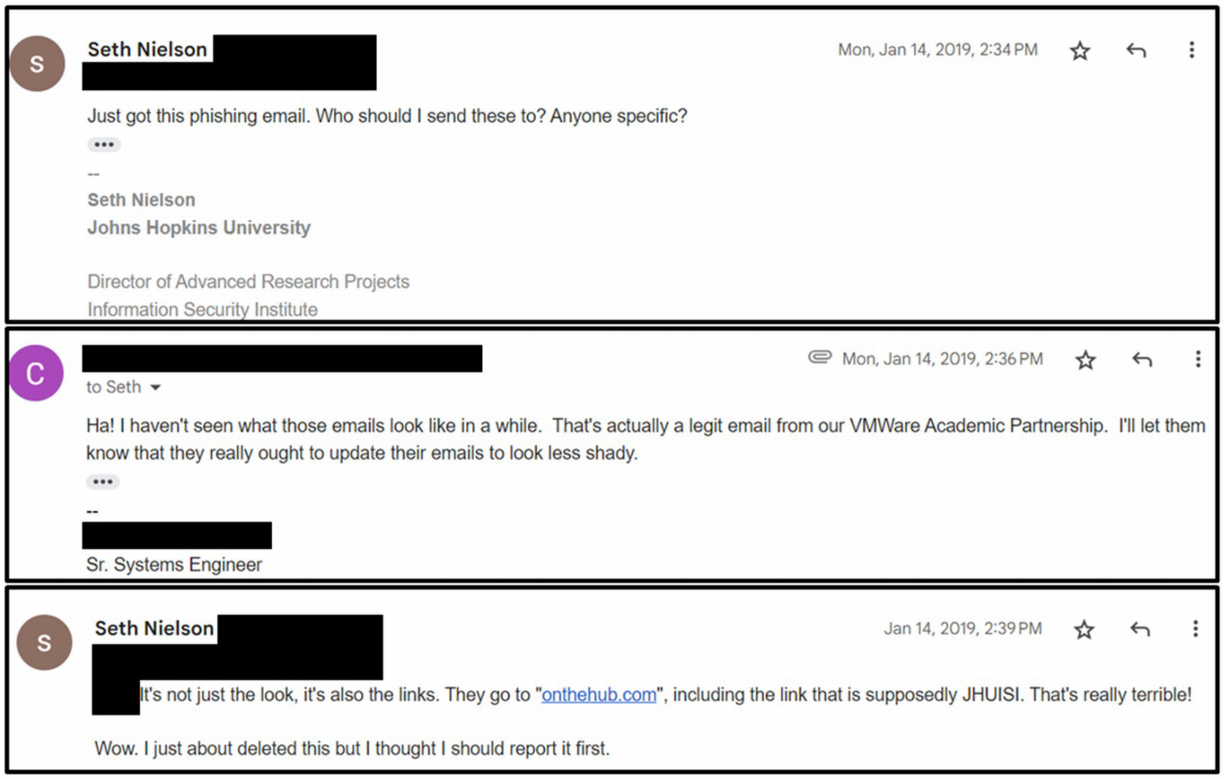
If you have not requested a password reset, please contact Customer Support.

If you have any questions, please see the Help page of the WebStore.

Thank you,
Johns Hopkins University - Information Security Institute

https://e5.onthehub.com/d.ashx?s=jgik0xvca7

Figure 10-10 I assumed this email was phishing. Notice the link does not go to a JHU URL



Seth Nielson [redacted]

Mon, Jan 14, 2019, 2:34 PM

Just got this phishing email. Who should I send these to? Anyone specific?

--
Seth Nielson
Johns Hopkins University

Director of Advanced Research Projects
Information Security Institute

[redacted]

to Seth

Mon, Jan 14, 2019, 2:36 PM

Ha! I haven't seen what those emails look like in a while. That's actually a legit email from our VMWare Academic Partnership. I'll let them know that they really ought to update their emails to look less shady.

--
[redacted]
Sr. Systems Engineer

Seth Nielson [redacted]

Jan 14, 2019, 2:39 PM

It's not just the look, it's also the links. They go to "onthehub.com", including the link that is supposedly JHUISI. That's really terrible!

Wow. I just about deleted this but I thought I should report it first.

Figure 10-11 This time, it turns out the message was legitimate. But if it is suspicious, it never hurts to check

Filtering Fraudulent Messages

In protecting email accounts from spam and phishing, email systems generally include filtration capabilities. The goal is to filter the messages before they reach the inbox. Filtered messages are often put into a separate folder, such as a spam folder. A wide range of filters have been developed to identify malicious emails.

The approaches used for the different types of fraud vary. Spam detection and bulk phishing detection can have elements in common. Both tend to have general messages that are recognizably different from more personal email. Moreover, because the same email is blasted out to so many people, the identification systems can quickly learn to recognize the bad email and ones that are similar to it.

On the other hand, filtering spear phishing attacks generally has to be done differently. As explained earlier, spear phishing emails are custom crafted and highly targeted and, because this is different from generic bulk phishing emails, require different detection. Typically, a spear phishing filter will look to see if the sender's email address is deceptive or if there is a detectable call to action in the email body. The sender's email address can be an important clue. For example, spear phishing often appears to come from someone that is known to the victim. Accordingly, the potential victim's contact list can be scanned for display names that look the same but have different email addresses. While not a perfect detector (after all, there are a lot of "John Smiths" in the United States), it is an example of how to check for spear phishing.

Another approach for filtering out both phishing and spear phishing emails is to identify suspicious links in the body of the email. Many domain names associated with phishing are known and tracked by security organizations. These organizations publish a list of bad domains and bad IP addresses, sometimes as part of a subscription. A defensive filter can look at URLs within the body, determine the domain and the IP address, and check if the domain and/or IP address is flagged as potentially malicious. If so, the email can be discarded.

The final filtration approach I will discuss in this chapter is detecting spoofed email addresses, which claim to be from an address that it is not. As discussed earlier, the default email protocol has no ability to verify that the sender's email address is authentic. A pair of additional technologies called *DomainKeys Identified Mail*, or DKIM, and *Sender Policy Framework*, or SPF, are technologies used for providing some partial verification of the sender. Neither technology attempts to prove that the sender controls the claimed address. Instead, they are used during the actual reception of the email to see if the message came from a computer authorized to send email for the claimed domain. That is, if the sender claimed to have a `gmail.com` address (e.g., `bob@gmail.com`), the message must have been sent by a Google-authorized server or it is not authentic. DKIM works by providing an authorized signature, while SPF works by identifying authorized IP addresses for senders. Either technology provides similar security. Additional mechanisms can be used to publish which of these two methods a domain is using so that recipients can determine how to verify the sender addresses.

Controlling Messages

The protections in this section focus on different security concerns than those discussed elsewhere in the chapter. However, the conclusion ties it back to issues such as phishing. The protections I am introducing here provide more security email communications through the use of cryptography and/or access controls.

Email transmissions by default are neither encrypted nor authenticated. Without encryption, the message cannot be kept confidential if, for example, it were accidentally forwarded to an unintended recipient. Without authentication, it is impossible to know, for sure, who sent the message. Various systems have been proposed that would add these kinds of security features into email.

A relatively old system is known as *S/MIME*, which stands for *Secure/Multipurpose Internet Mail Extensions*. S/MIME uses public key cryptography, similar to TLS, to provide authenticating signatures over unencrypted email or to provide both encryption and signatures for email.

The basic operation of S/MIME starts with an email sender obtaining a private key and a certificate (which contains the corresponding public key as discussed in Chapter 6). When an email message is sent, the private key is used to generate a signature over the message. The signature is sent with the message. To be verified at the other side, the recipient must have a copy of the sender’s certificate. Using the public key in the certificate, the contents of the message can be verified as authentic.

It is also possible to encrypt the email message provided that the sender has a certificate *of the recipient* (containing an RSA public key). Remember that when performing RSA public key encryption, it is the public key (which is in the certificate) that encrypts and the private key that decrypts. If a sender has the recipient’s certificate, the RSA public key is extracted and used to encrypt the message. The sender will still use their private key to sign. To repeat, the message will be signed with the sender’s private key and encrypted with the recipient’s public key. When the recipient receives the message, the recipient’s private key is used to decrypt the message and obtain the original message. The message is verified by using the public key from the sender’s certificate.

Two challenges keep S/MIME from being widely adopted. The first is that S/MIME performs end-to-end encryption. It cannot be decrypted by systems in the middle. This means it cannot be used by webmail that does not have explicit and built-in support. The second issue is that exchanging certificates is challenging. If the certificate is signed by a CA, it is relatively easy to distribute. It can be sent via any channel, even an untrusted one, because the recipient can validate the certificate using the CA that issued it. However, if the certificate is created self-signed, which is quite common, the only way the certificate can be trusted is if it is exchanged through a secure channel (e.g., in person or over a secure connection). In any event, the need to keep track of a certificate for everyone that you want to send a message to is a bit awkward and does not scale well.

A more modern alternative is Microsoft’s Message Encryption technology. This technology not only enables encryption and authentication, it also enables various access controls such as whether or not a received message can be forwarded, printed, or even kept permanently. Microsoft’s system works by enabling encryption and access controls under certain triggers. For example, a user can set a sensitivity label (e.g., “confidential”) in Outlook.[6] The label may be associated with an encryption operation and access controls, all of which are applied to the message when it is sent.

When the message is delivered to another user of Microsoft (with the appropriate license), Microsoft’s software (e.g., Outlook) will decrypt the message for the recipient to view. It will also enforce the access controls. For users that do not use Microsoft or otherwise do not have the built-in capabilities, a link is sent instead of the original email. Once received, the user is required to create a Microsoft login if they do not already have one. The link permits them to log in and view the message in a controlled application that performs the decryption and the enforcement of access controls.

These systems are not directly related to preventing attacks discussed in this chapter. However, they can provide some protections against phishing. For S/MIME, for example, if a recipient is used to receiving sensitive messages signed by the sender, they may notice the spear phishing email that claims to be from a sender, but lacks the signature to prove it. It is also possible to introduce policies that require messages to be signed for certain types of messages.

Microsoft’s Message Encryption provides the same benefit. If messages about financial transactions are expected to arrive with a specific sensitivity label, then users may recognize the fraudulent emails that do not have the appropriate label. Moreover, the access controls may also be useful at preventing exploitation. I mentioned that a sensitivity label can be applied manually, but Microsoft Office 365 can be automatically configured to apply sensitivity labels upon certain stimuli, including keywords detected in the text. So, suppose that a spear phishing email arrived about changing an account for a payment. An automatic policy could be implemented that would flag any message discussing financial data that did not arrive encrypted.

The other big benefit to Microsoft's Message Encryption is simply the downstream control. The access controls that can be put on a message include a do-not-forward control and a mechanism for automatically deleting in the future. This limits the amount of sensitive data on other machines and accounts, reducing the risk that if those machines or accounts are compromised, the sender's data will also be compromised.

Investigating Social Media Misuse

The previous two sections both focused on fraudulent messages, such as phishing and spam. But what about artificial amplification and reputation attacks?

Unfortunately, these types of problems are significantly less solvable with defenses. Defeating a reputation attack commonly requires the use of expert services that have the capabilities to monitor social media for references to an organization's brand. ZeroFox, for example, is an American company that offers brand monitoring and remediation. They have capabilities for scanning a wide range of sources to detect when there is impersonation related to an organization or when false information is being published about an organization. Once the false info is detected, the victim can request that the social media providers remove the false content, which they typically have been willing to do.

In terms of disinformation and artificial amplification, the best thing a person can do for themselves is simply stop using social media. Social networking sites do not have great track records with privacy, introduce additional channels for malware and fraud, and may not even be healthy for mental health. At least one 2018 study of college students, for example, found that college students that strictly limited using social media reported improved mental health outcomes [140].

Clearly, exiting social media may not be viable for an organization that reaches a wide range of customers via those channels. And even if they exited, that would not stop brand exploitation by others. In fact, it might make it worse. Nevertheless, it is my professional advice that private individuals, at least, should limit social media use both in time and in scope. If enough people reduced their interactions with social media, there would be a natural reduction in amplification and attacks on reputation.

Summary

Email operations and social media sites are two common communication channels that are extensively used for personal and professional purposes. However, these channels are also prone to various threats such as spam, phishing, bulk phishing, and spear phishing, which are aimed at stealing sensitive information or money from users.

Spam refers to unsolicited messages sent to a large number of users, whereas phishing involves tricking users into providing personal information by masquerading as a trustworthy entity. Bulk phishing targets a large group of users, whereas spear phishing is a more targeted approach to tricking a specific user into revealing sensitive information.

Artificial amplification and disinformation are major threats that have emerged in recent times. These threats involve the spread of false information and the amplification of opinions using artificial means such as bots and fake accounts. Reputation attacks are another common threat that targets individuals or organizations by damaging their reputation through negative comments or reviews. These attacks can have serious consequences for the target's personal and professional life.

To defend against these threats, various strategies can be implemented. Filtering fraudulent messages is an effective way to prevent spam and phishing attacks. Controlling messages by setting up security protocols, monitoring user activity, and implementing strong password policies can also help in reducing the likelihood of attacks. Investigating social media misuse involves identifying and removing fake accounts, monitoring suspicious activity, and reporting potential security breaches. Reducing the total amount of social media usage would naturally result in a reduction of social media misuse and impact.

Further Reading

Stallings' book *Cryptography and Network Security* includes an entire chapter entitled "Electronic Mail Security." It provides a more detailed background about the architecture of email systems, formats of email messages, and other background information. The chapter also explains in detail the various threats against email and the defenses such as DKIM, SPF, DMARC, and others [250]. Peterson and Davie's book *Computer Networks* also covers email specifically and overlay networks generally. This book does not discuss email and social media as examples of overlay networks and instead focuses on more classic peer-to-peer networking examples but the concepts and principles apply [205, Chapter 9].

In terms of analyzing the different kinds of evil messages in email and social media, each class has its own area of study. Here are a few recommendations for each one.

Spam detection and elimination has been going on for a long time, with paper from the early 2000s to the present [106, 113, 231, 262]. Since the beginning, the profit model has been of fascination to researchers [144]. More modern papers analyze spam in social media as well as in email [93, 146, 289].

There are many articles and books written about phishing. In previous chapters, I recommended *Introduction to Cyberdeception* for both the overall psychology and defensive deception. It also discusses phishing and the concepts and principles behind it [225]. Anderson’s chapter on psychology is also useful for thinking about how phishing works [40, Chapter 3]. Another interesting read is *Social Engineering* [125]. In some ways, though, the best reading about phishing is simply to see as many examples as possible. I find many examples to show my classes with simple Google searches. For example, UC Berkeley’s Information Security Office provides an archive of such emails [6]. You may also find it useful to review trade papers such as Cofense’s “Annual State of Phishing Report” [22].

A helpful analysis of trends with respect to ransomware is the reports produced by Sophos. The most recent report is “The State of Ransomware 2022,” and, of course, there are reports for previous years. These reports cover information about how often ransomware occurs, how often the ransom is paid, and the average cost, among others [26]. There are many articles that analyze how ransomware and ransomware defenses are evolving and the impact of ransomware on businesses [68, 196, 200, 285, 290]. In terms of defense, CISA’s MS-ISAC (Multi-State Information Sharing and Analysis Center) offers a reasonably good and understandable guide [18]. For a more comprehensive treatment, you might try *The Art of Cyberwarfare: An Investigator’s Guide to Espionage, Ransomware, and Organized Cybercrime* [94].

References

6. Phishing examples archive.

7. Security summit alert: Renewed alert about phishing e-mail scam targeting payroll or human resource departments.

12. Istr: Internet security threat report, Phrack vol. 22, Apr 2017. http://phrack.org/issues/49/14.html.

16. Istr: Internet security threat report, vol. 24, Feb 2019.

18. Ransomware guide. Technical report, 09 2020.

22. 2022 annual state of phishing report. Technical report, Cofense, 2022.

26. The state of ransomware 2022. Technical report, Sophos, 04 2022.

32. Aggarwal, C.C. 2011. *Social Network Data Analytics*. Springer. [Crossref][zbMATH]

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing. [Crossref]

61. Boshmaf, Y., I. Muslukhov, K. Beznosov, and M. Ripeanu. 2013. Design and analysis of a social botnet. *Computer Networks* 57(2): 556–578. [Crossref]

63. Boyd, D.M., and N.B. Ellison. 2007. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication* 13(1): 210–230. [Crossref]

68. Cartwright, A., E. Cartwright, J. MacColl, G. Mott, S. Turner, J. Sullivan, and J.R. Nurse. 2023. How cyber insurance influences the ransomware

payment decision: theory and evidence. *The Geneva Papers on Risk and Insurance-Issues and Practice*, 1–32 (48).

81. Cook, D.M., B. Waugh, M. Abdipanah, O. Hashemi, and S.A. Rahman. 2014. Twitter deception and influence: Issues of identity, slacktivism, and puppetry. *Journal of Information Warfare* 13(1): 58–71.

93. Dhaka, D., and M. Mehrotra. 2019. Cross-domain spam detection in social media: A survey. In *Emerging Technologies in Computer Engineering: Microservices in Big Data Analytics: Second International Conference, ICETCE 2019*, Jaipur, 98–112. Springer.

94. DiMaggio, J. 2022. *The Art of Cyberwarfare: An Investigator's Guide to Espionage, Ransomware, and Organized Cybercrime*. No Starch Press.

96. Douceur, J.R. 2002. The sybil attack. In *Peer-to-Peer Systems: First International Workshop (IPTPS 2002)*, Cambridge, MA, 251–260. Springer.

106. Ferrara, E. 2019. The history of digital spam. *Communications of the ACM* 62(8): 82–91.
[Crossref]

113. Geerthik, S. 2013. Survey on internet spam: Classification and analysis. *International Journal of Computer Technology and Applications* 4(3): 384.

125. Hadnagy, C. 2018. *Social Engineering: The Science of Human Hacking*, 2nd ed. Wiley.
[Crossref]

140. Hunt, M.G., R. Marx, C. Lipson, and J. Young. 2018. No more FOMO: Limiting social media decreases loneliness and depression. *Journal of Social and Clinical Psychology* 37(10): 751–768.
[Crossref]

144. Judge, P., D. Alperovitch, and W. Yang. 2005. Understanding and reversing the profit model of spam (position paper). In *Proceedings of the 4th Workshop on the Economics of Information Security*.

146. Kabakus, A.T., and R. Kara. 2017. A survey of spam detection methods on Twitter. *International Journal of Advanced Computer Science and Applications* 8(3): 29–38.

158. Lampson, B. 1993. *Principles for Computer System Design*. New York: Association for Computing Machinery.
[zbMATH]

196. O'Kane, P., S. Sezer, and D. Carlin. 2018a. Evolution of ransomware. *IET*

Networks 7(5): 321–327.
[Crossref]

200. Oz, H., A. Aris, A. Levi, and A.S. Uluagac. 2022. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Computing Surveys* 54(11s): 1–37.
[Crossref]

205. Peterson, L.L., and B.S. Davie. 2021. *Computer Networks*, 6th ed. Morgan Kaufmann.
[zbMATH]

214. Rao, J.M., and D.H. Reiley. 2012. The economics of spam. *Journal of Economic Perspectives* 26(3): 87–110.
[Crossref]

225. Rowe, N.C., and J. Rrushi. 2016. *Introduction to Cyberdeception*, 1 ed. Springer International Publishing Switzerland.
[Crossref]

231. Sanz, E.P., J.M. Gómez Hidalgo, and J.C. Cortizo Pérez. 2008. Email spam filtering. In *Software Development*, Advances in Computers, vol. 74, 45–114. Elsevier.

236. Securities and Exchange Commission, U. 2023. Administrative proceeding file no. 3-21306.

250. Stallings, W. 2013. *Cryptography and Network Security: Principles and Practice*, 6th ed. Prentice Hall Press.

262. Thorkildssen, H.W. 2004. Spam-different approaches to fighting unsolicited commercial email a survey of spam and spam countermeasures. *Network and System Administration Research Surveys* 1: 45–55.

285. Woods, D.W. 2023. A turning point for cyber insurance. *Communications of the ACM* 66(3): 41–44.
[Crossref]

289. Yurtseven, İ, S. Bagriyanik, and S. Ayvaz. 2021. A review of spam detection in social media. In *2021 6th International Conference on Computer Science and Engineering (UBMK)*, 383–388. IEEE.

290. Yuryna Connolly, L., D.S. Wall, M. Lang, and B. Oddson. 2020. An empirical study of ransomware attacks on organizations: An assessment of severity and salient factors affecting vulnerability. *Journal of Cybersecurity* 6(1): tyaa023.

Footnotes

1 Email addresses can be associated with computers for certain automated tasks, but this is an unusual adaptation of a network that is primarily human.

2 Technically, the MUA may transmit to a Mail Submission Agent (MSA) first, and the MSA gives the message to the MTA. However, these are often on the same machine and work as a single agent. For simplicity, I have removed the MSA from the explanation.

3 This is not a real address; please do not send email to this address.

4 In the movie *Inception*, the protagonists enter the dreams of a target and plant a thought into his mind. They need the thought to seem self-generated, so they place some ideas into his subconscious that later are repeated back to him *from himself*.

5 It is possible, of course, for people to legitimately sell their behaviors and voice to another. This could be a kind of spokesman or promoter engagement. However, it is usually desirable that this purchase be disclosed so that observers can factor that information into their opinions.

6 This functionality is not available in the default Office 365 license, either for personal or business. As of the time of this writing, it requires Microsoft Business Premium.

A: Binary and Hexadecimal Numbers

Computers are built on the concept of binary numbers: ones and zeros. A full explanation of why is beyond the scope of this book, but the basic idea is that computers store value as either “something is there” or “nothing is there.” While the computer is running, for example, a computer can recognize a raised power level as a “1” and a lower (or off) power level as a “0”. When the computer is off, data stored to a hard disk can be kept permanently by magnetizing parts of a disk. The computer again recognizes “magnetized” and “not magnetized” as on/off values.

To understand binary numbers, however, it actually requires understanding how our “normal” numbers work. By normal, I mean “base-10” numbers. If you can understand base-10, you can also understand base-2, which are binary numbers.

Base-10 Numbers: Decimal

When you see the number 111, you see three separate 1s. And yet, you know that this is not three 1s added together. Even though it is the same three symbols, you know that the *value* of the symbol depends on where it is placed.

In everyday society, the number 111 has a value of one hundred and eleven. Let’s break down how that works. Suppose we start with the right-most digit. How much is that 1 worth? Well, it is worth 1, actually. If you replace it with a 0, the value will decrease by 1.

But how much is the next 1 worth? If you replace it with 0, how much will the value of the number decrease by? The answer is *10*. You may remember in grade school that we actually name the columns of our numbers. The right-most digit is in the 1s column, and the next digit to the left is in the 10s column. This means a 1 in the 1s column is worth 1, while a 1 in the 10s column is worth ten.

An easy way to think about this is using coins. In the United States, a penny is worth one cent (arguably, it is not worth anything at all), and a dime is worth ten cents. So even if you have one penny and one dime, you do not have *two* cents. The dime is worth more.

Moving one more column to the left, we have the 100s column and, as you might have already suspected, a 1 in this column is worth 100.

Why, however, is it a 1s column, a 10s column, and a 100s column? It is not arbitrary! First, notice that in this number system, there are *10 symbols* for representing numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. With just 10 symbols, *any* number can be represented. If we count upward from 0, we can just advance to the next number until we hit 9. Once we hit 9 there is no new symbol to use. At that point, it is necessary to move over to a new column. In effect, we *reset* the counter in the 1s column and *increase* the counter in the 10s column *because we just reached the number 10*!

In other words, when counting upward and we run out of symbols in one column, we need the next column to the left to "count" or keep track of that so we can reset and start over again. So the next column over must have a value equal to the counter when the reset happens. If the next number after 9 is 10, then the value of the column to the left must be 10.

But what about the 100s column? Well, remember that counting in the 10s column isn't counting by 1s. It is counting by 10s. So the 100s column has to keep track of how many times the 10s column has maxed out and reset.

There is, however, another way of looking at the value of the column. It has to do with how base-10 numbers are constructed. In base-10, the value of each column is ten raised to a power (don’t forget that anything raised to the 0 power is 1!!!).

Place 3	Place 2	Place 1	Place 0
10^3	10^2	10^1	10^0
$10 \times 10 \times 10$	10×10	10	1
1000	100	10	1

Once these columns are established, the value of any base-10 number is caclulated by multiplying the value of the column times the symbol in the column. Again, using 111 as an example:

$(1 \times 10^2) + (1 \times 10^1) + (1 \times 10^0)$

Make sure you understand this before moving on. Try writing out some other numbers and expanding them in this way. If you can do that, everything else in this appendix will be easier!

Base-2 Numbers: Binary

As stated above, the number system you are most familiar with is base-10. Binary numbers, however, are not that different. But instead of being base-10, they are *base-2*. This means that there are only two symbols instead of ten, and the value of each column will be a power of 2 instead of a power of 10.

As a starting point, here are the first ten binary numbers written along side the base-10 equivalent that you are most familiar with.

```
 0      0
 1      1
 2     10
 3     11
 4    100
 5    101
 6    110
 7    111
 8   1000
 9   1001
```

Why does binary work this way? Suppose that we want to just count from 0 to 10. We start with 0 (which is one of our two symbols) and when adding 1 we get 1! So far, so good! But we are now out of symbols because, again, we only have two: “0” and “1”! There is no symbol beyond 1. Just like there is no symbol beyond 9 in base 10, there is no symbol beyond 1 in base-2.

Remember that in base-10, when we ran out of symbols we reset the column to 0 and added 1 to the next column. So, to add 1 and 1 in binary, the right-most column resets back to 0 and the next left column is increased.

More importantly, each column’s value can be easily calculated using the “base raised to a power” approach discussed previously for base 10. Recall that each column in base 10 was 10 raised to a power. For base-2, it is 2 raised to a power as shown below:

Place 3	**Place 2**	**Place 1**	**Place 0**
2^3	2^2	2^1	2^0
$2 \times 2 \times 2$	2×2	2	1
8	4	2	1

This means that it is relatively easy to decode small binary values. Take the binary number 1100. How much is this worth?

$(1 \times 8) + (1 \times 4) + (0 \times 2) + (0 \times 1)$

So, working out the math, 1100 in binary is 12 in base-10.

Of course, if I ask you what is 10, there is a real problem. Am I asking what is 10 in base-2, or am I asking what is 10 in base-10? When displaying numbers for humans to read, most computers use “prefixes” for numbers not in base-10. The common prefix for binary is “0b”. So if you see “10”, you can usually assume base-10 unless told otherwise. But `0b10` is the binary number 10, which is 2 in decimal.

Base-16 Numbers: Hexadecimal

It is also common to see certain kinds of computer numbers written in base-16, which is known as hexadecimal. If you are beginning to see the pattern, you should have guessed that this means there are 16 symbols and the value of each column is a power of 16.

You might be wondering how you can have 16 symbols. After all, you are only familiar with 10 (0 through 9). Hexadecimal adds in the letters “A” through “F” for the values of 10–15 respectively. Thus, counting in “hex” is as follows:

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E

15	F

The following tables show the value of hexadecimal columns as compared with binary and decimal columns.

	Place 3	**Place 2**	**Place 1**	**Place 0**
Binary	2^3	2^2	2^1	2^0
Decimal	10^3	10^2	10^1	10^0
Hexadecimal	16^3	16^2	16^1	16^0

Or, multiplied out:

	Place 3	**Place 2**	**Place 1**	**Place 0**
Binary	8	4	2	1
Decimal	1000	100	10	1
Hexadecimal	4096	256	16	1

All of these number systems work in the same way: place value is determined by adding one to an exponent on the base.

Hexadecimal also has a common prefix in computer usage: “0x”. So, for example `0x10` is not decimal 10. Rather, it is 16. Remember, the column to the left of the 1s column in hex is worth 16.

So, as an exercise, what is the value of `0x3A0F`? Using the tables above, you can see that the value of the column the 3 is in is 4096, the A is in the 256ths column, the zero is ignored, and the F is in the 1s column.

$(3 \times 4096) + (10 \times 256) + (0 \times 16) + (15 \times 1) = 14863$

Why do we care about hexadecimal in the first place? Consider the following table with hex on the left and binary on the right:

0	0
1	1

2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

We ran out of digits in hex at exactly the same time that we needed to go from 4 columns to 5 in binary! That's really helpful, because it means we can trivially convert back and forth between a computer's native and sprawling binary numbers to the much more human-friendly and compact hex numbers. People even get good enough at this that they can just translate them on sight. Here's an example with binary on top and hex underneath:

```
101 1100 1010 0011 0111
5   c    a    3    7
```

No matter how big a binary number gets, you can take every four bits and write them as a single hexadecimal digit. For this reason many of the examples in this book, such as the cryptography examples, put the output into hex.

B: Computers, Data, and Programs

Computer Hardware

For purposes of this book, there are three major pieces of computer hardware that are necessary to understand how a computer operates.

The first component is the computer’s main processor: the Central Processing Unit (CPU). The CPU is sometimes described as the “brain” of a computer. As will be discussed later in this appendix, processors have “instructions” that are commands that trigger different processor functions. Some of the processor’s major functions are:

- Mathematical computations
- Memory operations (load/store in RAM)
- Control operations
- Security operations

CPUs largely work with data in system memory, typically RAM (Random Access Memory). Computer memory can be represented by a big long list of addresses that start at 0 and go up to the memory’s size. The CPU can store data in RAM at any one of these addresses and can retrieve it again in the future. RAM, however, will not store or retain data when the power is off.

To store data long term, permanent or “persistent” storage is needed. Such storage typically comes in the form of hard disks, USB sticks or drives, optical media such as CDs, DVDs, or Blu-Ray disks, and so forth. Although the optical media typically can only be written once, hard drives and USB sticks can store, retrieve, and erase data. Moreover, the data is not lost when the power is shut down to the system.

So why do computers use RAM at all? RAM is many, many times faster than a hard drive or USB stick. If you tried running your programs from hard drives, it would be incredibly slow and very likely would not function well.

Data stored, whether in RAM or on disk, is measured in terms of the number of bytes. A “byte” is 8 bits, where a bit is a single 1 or 0 (binary). Approximately 1,000 bytes (1,024 to be exact) is a kilobyte (KB). Approximately one million bytes is a megabyte (MB). Gigabytes and Terabytes are approximately one billion and one trillion bytes respectively.

Data Formats

As mentioned several times in this book, all computer data is a number. So how does that work? How can a word document be a number? How can a photograph?

To store data, an “encoding” format must be chosen and agreed upon. An encoding format is a way of translating data to numbers and back again. This encoding is *arbitrary* but must be agreed upon by the programs that use the data. This may seem strange, but we humans do it all the time. Think about the number pad on a phone. You can press a number on a phone to represent a number (e.g., for dialing), for letters (e.g., when you need to input words), and for commands (e.g., “press one for English.”). The number is *interpreted* differently based on *context*.

An early encoding format for letters still used today is called ASCII (pronounced “ask-ee”). This format assigns a number that could fit into a single byte to each letter of the English alphabet, plus punctuation, and a few other symbols. For example, in ASCII, the number for the letter ‘A’ is 65 and the number for the letter ‘a’ is 97. Full charts of this encoding can be found online with a simple Google search.

Encoding formats are sometimes evaluated in terms of how much space it takes to encode data. For example, using ASCII, each character for western languages takes up one byte. So text containing 200 characters (don’t forget punctuation and spaces!) will require 200 bytes of storage. Of course, word documents are larger than this because they contain control information such as fonts, formatting, and so forth. This data must *also* be encoded in some way.

It is not uncommon to have multiple layers of encoding. Data for a webpage can be created using HyperText Markup Language or HTML. But HTML is, itself, text and must be encoded for actual storage on a hard drive (e.g., using ASCII). Here is a simple HTML document:

```
<HTML>
<BODY>
<B>Hello</B> <I>World</I>!
</BODY>
</HTML>
```

HTML enables a document to store formatting information along with the text. The “ $<$ B $>$ ” tags, for example, encode that the data in between should be in bold and “ $<$ I $>$ ” indicates that the text within the tags should be italicized. Again, all of this HTML data is, itself, text that must be encoded into numbers for storage on the file system.

Visual data like photographs are also stored as numbers. A picture on a computer is made up of many tiny dots called “pixels” that, when displayed together, are understood by the human brain to be an image. Suppose that the image is 800 pixels by 600 pixels. The total area of the image is 480,000 pixels. If the image is black and white, then one way to represent it is a single bit for a pixel. If the bit is a 1, the pixel is white. If it is 0, the pixel is black. It would take 480,000 bits (60,000 bytes) of uncompressed data store this 800 by 600 black-and-white picture. To illustrate, look at Figures B-1 and B-2.

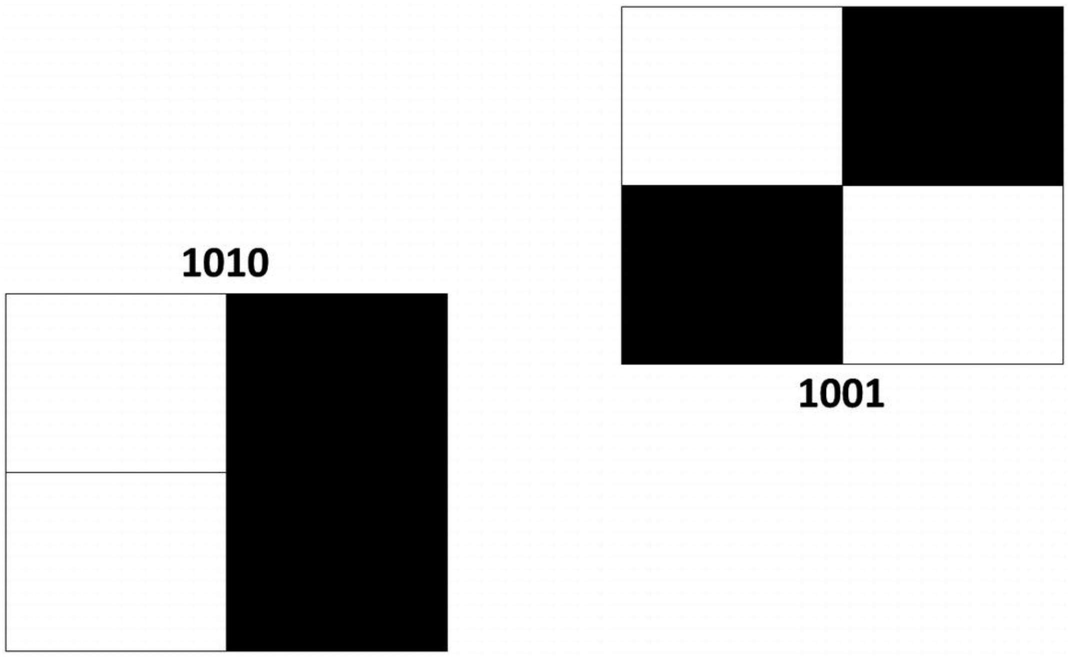


Figure B-1 In this very basic example, a 4-bit number can be encoded as a 2×2 black-and-white image. Clearly, the pixels in this image are much bigger than a display

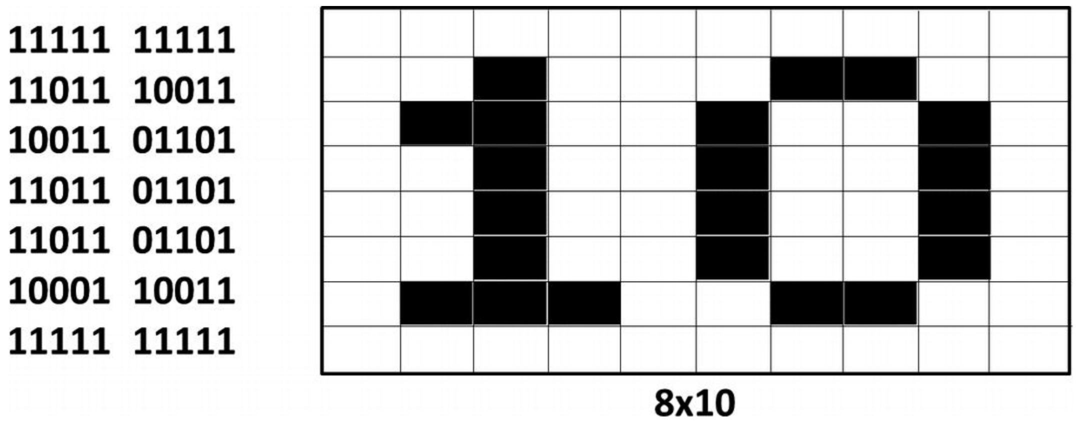


Figure B-2 In a slightly more interesting example, a 80-bit number can be encoded as an 8×10 black-and-white image. Although the pixels are enormously sized, it begins to illustrate how an image can be created from a binary number

If the picture needs to be color, then more bits are required for each pixel to represent the color information that goes along with the pixel. To represent 16 colors, for example, it would take four bits to represent each color individually. So an uncompressed file using this hypothetical format would take 240,000 bytes (60,000 bytes times 4).

Most pictures are generally not stored this way because it takes up too much space. Compression formats enable picture data to be stored in less space using various techniques. For example, large blocks of the picture with the same color can be represented together instead of each pixel individually. Videos have even more compression techniques to keep sizes down.

The key point, however, is that all data is converted, through an encoding means, into some kind of numeric format that can be stored on disk and in memory.

One last point: even computer instructions are *numbers*. Computer programs, which are instructions to the processor, must be numbers as well.

Program Execution

Given all of the amazing things that computer programs can do, it may be surprising to learn that, in the end, all computer programs are converted one way or another into operations that are relatively simple. A program is a set of instructions for a computer processor. The instructions are either ones and zeros that can be understood directly (or *natively*) by the CPU, or they are instructions that are *translated* by another program into the CPU's language.

Different types or families of CPUs have different languages. These languages are called *instruction sets*. An instruction set is just the list of instructions that the processor can understand. Despite the fact that different processor families use different instruction sets, the *types* of instructions are pretty consistent. Three of the most crucial types of instructions almost all CPUs have are:

- Arithmetic operations
- Memory operations

- Branching operations

Arithmetic operations are, unsurprisingly, instructions related to adding, subtracting, multiplying, and dividing.

Memory operations, on the other hand, have to do with loading and storing information into Random Access Memory (RAM). CPUs typically can only store a very small amount of data in the processor itself. This data is stored in *registers*, memory storage slots that can hold a unit of data. The registers of many CPUs each hold a 64-bit binary number. For example, an x86-64 processor (the most common CPU architecture for Windows computers these days) has at least 40 registers. One register holds the “instruction pointer,” containing a memory address of the next instruction to be executed. Another, the “status register,” holds data about the status of the processor and about the calculation most recently performed.

A large number of the registers, of course, store data primarily for immediate computations. For example, most arithmetic operations require some or all of the data to be located in registers. The results of operations are often stored in registers as well. But there needs to be a way to load data into registers and a way to save data out of registers (before, for example, it gets overwritten by the next operation). Memory operations allow the CPU to load the data from RAM and store results to RAM as needed.

The last set of operations, branching operations, are, in many ways, what makes a program a program. As I stated above, a program is a set of instructions. Running a program involves passing those instructions to the CPU to execute. If there was no ability to *branch* (take different paths through the set of instructions), the CPU could only execute the instructions one right after the other until they were finished. The program would do the same thing every single time it was run, no matter what input it received. A branching operation changes this by changing which instruction runs next based on some *condition*.

Branching instructions are often structured like this: if the value in a register is 0, jump ahead 100 instructions. There are also instructions like *jump* that are not conditional but instead mandate a jump to some specific instruction.

As discussed earlier, a number stored in a computer could be used in one place to be a number, or a letter in another place, or the number could be an instruction to a processor. So, running a program is basically sending a bunch of numbers to the processor that the processor interprets as instructions.

Where did the numbers come from? Running a program typically involves copying some sequence of instructions and other data out of storage into RAM. Storage technologies, including hard drives, are typically too slow for program execution.

The RAM memory is basically one big long sequence of memory locations. Commonly the addresses are based on bytes so that every one byte (eight bits) is a single address. For various reasons, the first memory location is identified as 0 instead of 1 and it is usually writen out with leading zeros: `0x00000000`. The actual number of leading zeros will depend on the hardware and software of the system. The 8 digits I am using are meant to be illustrative.

Also, notice the “0x” prefix. Recall from the discussion of binary in Appendix A that “0x” is a prefix that indicates hexadecimal notation and is often used as a shorthand for binary numbers. Because the memory addresses are used internally by the computer they are, themselves, stored in binary. So it is common to write them in this form.

The memory addressing is sequential. The next byte of memory has the address of 1 or, written in hexadecimal and with leading zeros, `0x00000001`. The memory addresses can be used flexibly depending on the system. Many systems use multiple bytes together as a single unit. These units are called *words* and are commonly 32-bits (4 bytes) and 64-bits (8 bytes). But these systems do not need to change the addressing of memory. They can simply have the first word at 0 (`0x00000000`) and the second word at 4 (`0x00000004`). This is the case with instructions, which are almost never one byte.

When a process starts up, the program's instructions are copied into memory, there is a loading process wherein the address of the first instruction is sent to the CPU, which stores it as the current instruction address. The CPU loads the instruction out of memory at the current instruction address and executes the loaded instruction. If the instruction is not a branching instruction, the CPU will increase the current instruction address by the appropriate amount so that the CPU can then load and execute the next instruction in memory.

On the other hand, if it is a branching instruction, the instruction will tell the CPU the next address to use as the current instruction address. This is similar to a "Choose-Your-Own-Adventure" book you may have read as a kid. If you were reading the book, you would go on from one page to the next unless it had instructions at the bottom to turn to a specific page, often dependent on a choice you made.

Although an example from a real CPU would be helpful, instructions in modern CPUs can be a little bit complicated and require more explanation than is appropriate for this background section. As an alternative, I will walk through an example using a made-up set of assembly instructions.

CPU Simulation Example

In order to help you understand how a processor works with its registers and with memory (RAM), I will walk you through a very simple and simplified paper-and-pencil example. This example will illustrate how instructions to a computer processor are just numbers and how those numbers are interpreted. It will also illustrate how these commands can be used to compute values. I will walk through two very simple programs. The first will basically just count to 3. The other will average 10 numbers stored in memory. You should try to do these steps on your own with pencil and paper before reading the solution.

Before walking through the exercise, I need to provide some setup instructions. For this exercise to work, we are going to have to *create* (in our imaginations) our very own computer processor. Why? Because *real* processors are much too complicated. It would be very difficult to do an exercise like this with real instructions. I will help you imagine up a much simpler processor.

As discussed above, a CPU makes use of some basic building blocks. Computer program instructions are usually stored in RAM along with data that it needs. It may also require some space for intermediate results. As discussed in Chapter 7, it also needs to hold the stack. In this example, we will *not* be simulating functions or subroutines and will not need a stack.

The CPU also has registers. The registers are special places where small amounts of data can be stored for the CPU to have (very) quick access to while it is executing instructions. These registers store only a small amount data. In a 32-bit computer, they generally store only 32 bits each; while in a 64-bit system, they generally store 64 bit values. But for purposes of this simulation, it would be a pain to work with even 32 bits. So our computer will only have *8-bit* registers!

Related to our small registers is a small memory. In a real computer, memory can be many gigabytes. We are not going to try and simulate gigabytes of memory! That would take too much paper! Instead, our memory will only have

256 addresses (0 through 255) and each address holds exactly one byte (or 8 bits). This introduces many limitations. Because our memory can only hold 8 bits in an address, each memory storage slot's maximum value is 255! For a real computer, this would be a serious limitation indeed. But for our simulation, we only want to work with relatively small numbers that make sense for paper-and-pencil computations.

I have already explained earlier in this appendix the concepts for how a processor (CPU) works. The CPU reads an instruction out of RAM, performs the instruction, and then moves on to a new instruction. We are going to do that here in our simulation! So, to get started, we need to have a list of instructions the CPU can understand.

In our simulation, the instructions are going to be 8 bits each (yes, the same size as our registers and our memory locations). When a CPU receives an instruction, the bits are broken up into pieces, each of which means a different thing. Typically, some number of bits at the beginning are called the *Op Code* and are the basic type of command. Our little simple and imaginary CPU will only understand 8 op codes. The first 3 bits (from the left-hand side) of every instruction will be the op code for that instruction. The following 5 bits will be used to specify the parameters which define the data on which the instruction is going to be performed. The set of all instructions for a CPU is called its *instruction set*.

Our 8-bit computer has the following instruction set:

- Load: 000
- Store: 001
- Add: 010
- Subtract: 011
- Multiply: 100
- Divide: 101
- Jump: 110

- Jump if not zero: 111

There are 3 types of instruction in this instruction set. The first is data movement: *STORE* and *LOAD* instructions. The first three bits (bits 1–3) define the instruction. The next 2 bits (bits 4–5) specify the register to store from or load to. The next 2 bits (bits 6–7) specify the register holding the memory address to store to or load from. The last bit is ignored.

In order to better understand how these instructions work, consider the following sequence of 8 bits: ‘00001110’. This can be parsed as such:

- Bits 1–3: 000 means Load.
- Bits 4–5: 01 specifies register r1 which is used to hold the data loaded from memory.
- Bits 6–7: 11 specifies register r3 for loading the address from memory.

So, those 8 bits actually mean “Load r1 r3”. When the CPU encounters these 8 bits, it will look in the r3 register, get a memory address, load the value stored in memory at that address, and store the result in r1. You might remember that hexadecimal is a shorthand way to representing binary values. Therefore, these 8 bits can be represented using hexadecimal as `0x0E` and it means the same thing. This instruction is illustrated in Figure B-3.

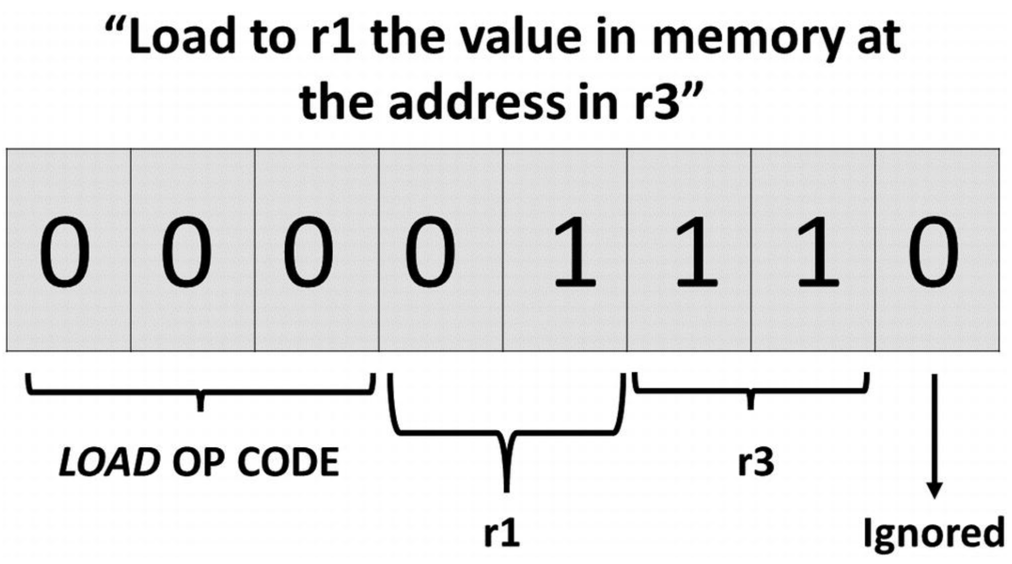


Figure B-3 An example *LOAD* instruction

The second is type of instructions are the arithmetic instructions: *ADD*, *SUBTRACT*, *MULTIPLY*, and *DIVIDE*. For these instructions, there are two values that need to be provided to perform the arithmetic operation on and then a location where the result will be saved. The first 3 bits (bits 1–3) are the instruction. The next bit (bit 4) specifies whether the second value for the operation is taken from a register (1) or is given as a literal number (0). The next 2 bits (bits 5–6) specify the register holding the first value for the operation. It is also the register where the operation result will be saved to. The final 2 bits (bits 7–8) specify either the register holding the second value or the literal value. Note that the DIVIDE instruction will halt the processor if there is a divide by zero error and it only returns the whole integer number of the division (no fractions or decimals).

It should also be noted that if doing these operations results in an answer that is bigger than 255 (the biggest number a register or memory address can hold), the value just wraps around. That is, $255 + 1 = 0$, $255 + 2 = 1$ and so forth. Some real processor instructions will save an overflow value in another register is overflow occurs. Our simple system ignores any complexity.

Repeating our exercise from above, here is an example ADD instruction of ‘01000010’

- Bits 1–3: 010 means Add.
- Bit 4: 0 means the last parameter is a number (not a register).
- Bits 5–6: 00 specifies register r0, from which a value will be loaded, and in which the final result will also be stored.
- Bits 6–7: 10 specifies the number (not register) 2, which will be added into r0.

So, those 8 bits actually mean “Add r0 2”. When the CPU encounters these 8 bits, it will take the value stored in r0, add 2 to it and then place it back into r0. The hexadecimal for this instruction is `0x42`. This instruction is illustrated in Figure B-4.

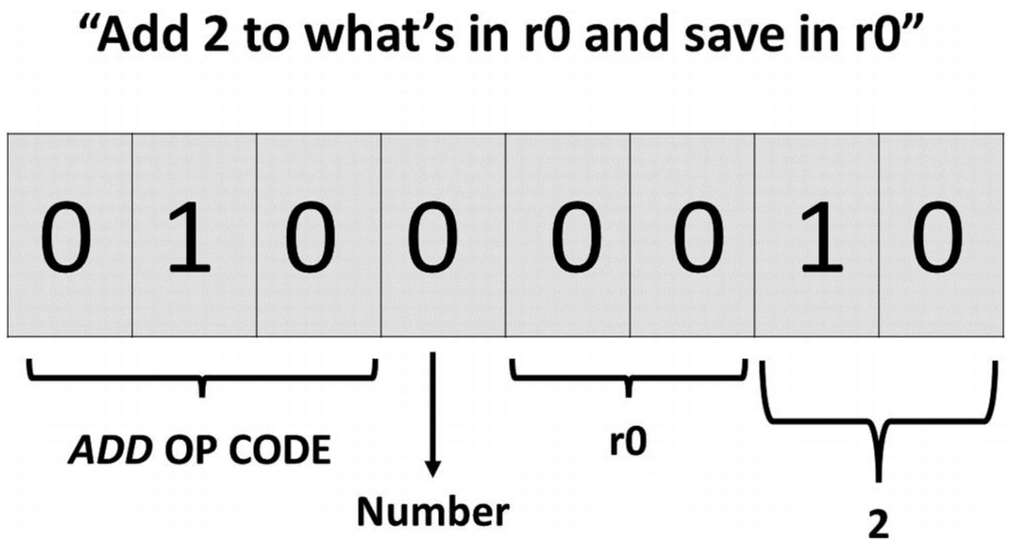


Figure B-4 An example *ADD* instruction

The third type are the jump instructions. The first is the *JUMP* instruction, also called the unconditional jump. This instruction has only 1 parameter. Bits 4–5 specify the register which holds the address of where the execution should jump to. The last 3 bits are ignored. The second is the ‘Jump if not zero’ instruction (or *JNZ*), also called the conditional jump. This takes 2 parameters. The first bit (bit 4) after the instruction itself specifies if the processor should jump forward or backward. The next 4 bits (bits 5–8) are a literal value of how many instructions to jump forward (0) or backward (1). When this instruction is encountered, the CPU checks the value inside a specific register (in actual systems, it’s a flag, instead of a full register value) to see if its value is zero. If the condition is met, then it will jump execution to a different place in the program and continue execution from there. In this hypothetical CPU, we will use r3 to be the register used to check for zero for a conditional jump. A JNZ instruction, or something like it, is roughly how if-statements are implemented at a low level.

In our final example, consider the 8 bits: ‘11101001’.

- Bits 1–3: 111 means Jump if not zero.
- Bit 4: 0 means a jump forward.
- Bits 5–8: 1001 is binary for 9. This means the processor should jump forward by 3 instructions.

This instructions, therefore, means “Jump if not zero 9”. One piece of information is missing here: the value of r3 is not given. As explained above, the JNZ checks that register for the “not zero” check. So, depending on whether the value in r3 is zero or not, the jump may or may not be performed. The hexadecimal for this instruction is `0xE9`. I have illustrated this JNZ instruction in Figure B-5.

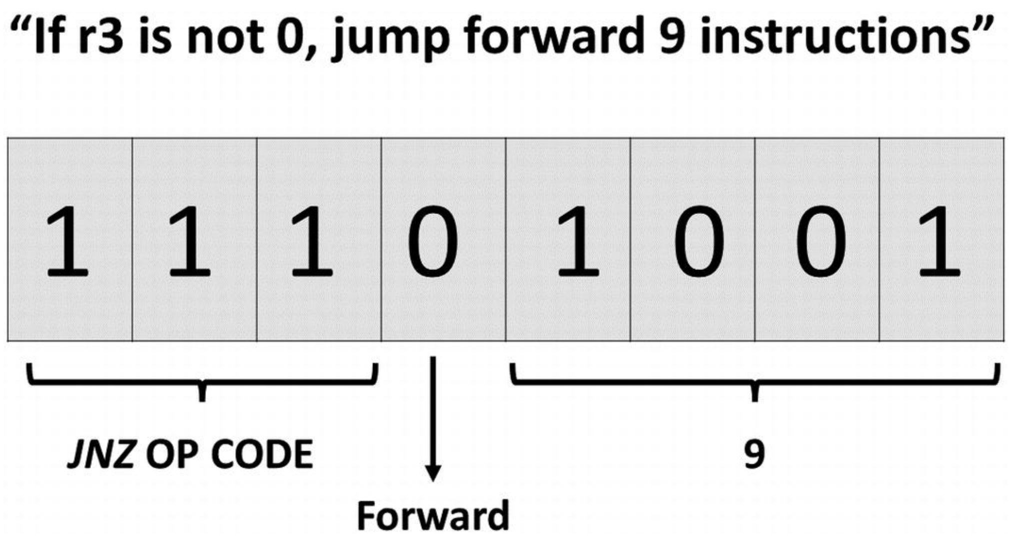


Figure B-5 An example *JNZ* instruction

You should notice that there are only 2 bits ever allotted for specifying registers. This means that for our hypothetical CPU, there can only be 4 registers named as r0, r1, r2, and r3. Similarly, the literal value can be at most 3 (0b11) when there are 2 bits, as is the case for most of our instructions; but for the JNZ instruction, we have 4 bits, which means that the maximum value can be 15.

Example 1: Count to 3 Now, let’s walk through a simple program which increments register r0 until it reaches 3. That is, when the value of register r0 reaches 3, the program ends.

As you may have noticed, our assembly instructions are extremely limited. For example, using these instructions, the biggest number that can be added directly to another number is *3* (remember, there are only two bits in an add instruction for a number to add, so that means the biggest number that can be added is 3). So it will take a little bit of work to do some of the things we want to do.

The basic instructions for our program will be as follows:

1. Add 3 to r2 (assume all registers start at 0)

2. Add 1 to r0
3. Multiply 0 into r3
4. Add r2 into r3
5. Subtract r0 from r3
6. JNZ -4

To show how the processor would see it, I have listed the binary that goes along with each instruction in the table below:

Instruction	Binary	Meaning	Comment
1.	01001011	Add, r2, 3	Initialize r2 to 3
2.	01000001	Add, r0, 1	Increment r0 by 1
3.	10000000	Mul, r3, 0	Reset r3 to 0
4.	01011110	Add, r3, r2	Get the value of r2 in r3
5.	01111100	Sub, r3, r0	Subtract r0 from r3 to check if the target value is reached
6.	11110100	JNZ, -4	If r3 is not zero, i.e., the target was not reached, jump back by 4 instructions to beginning of loop for next iteration.

If you want to try your hand at this exercise, please try to “run” this program on your own pencil and paper before reading how it works. Keep track of all four registers and make sure to only jump-if-not-zero if r3 is not zero. When you are finished, keep reading!

Did you try it? Let's walk through the logic. The following repeats the instructions but from a semantic perspective (i.e., what they mean for the program).

1. Initialize r2 to the value 3 by adding 3 (again, assume all registers start at 0)
2. Increase r0 by 1
3. Clear the value of r3 by setting it back to 0
4. Set the value of r3 to be the same as r2, which is 3
5. Subtract r0 from r3, which is 3 (i.e. $3 - r0$).
6. If $3 - r0$ is not zero, r0 is not 3 so jump back 4 instructions to instruction 2

That last step, the JNZ step, will always jump back *until* r0 is 3. Once it is 3, when subtracted from r3, r3 becomes 0, so the JNZ stops looping.

The value of the registers is important to understand as the program continues to execute. Table B-1 shows the values of each of the registers as the processor executes all the instructions and it loops over them until the value in r0 reaches the target. It is a lot to look at so I will highlight r0 when it is incremented, which is the counter we are tracking, and the r3 values on each JNZ check.

Table B-1 The table shows the values of each of the registers as the program executes and loops over the instructions

Instruction	r0	r1	r2	r3	Comment
Add 3 to r2	0	0	3	0	initialization
Add 1 to r0	**1**	0	3	0	increment r0
Multiply 0 into r3 3	1	0	3	0	reset r3 to 0
Add r2 into r3	1	0	3	3	copy r2 into r3

Instruction	**r0**	**r1**	**r2**	**r3**	**Comment**
Subtract r0 from r3	1	0	3	2	subtract 3 from r3
JNZ -4	1	0	3	**2**	r3 is not 0, loop
Add 1 to r0	**2**	0	3	2	increment r0
Multiply 0 into r3	2	0	3	0	reset r3 to 0
Add r2 into r3	2	0	3	3	copy r2 into r3
Subtract r0 from r3	2	0	3	1	subtract 3 from r3
JNZ -4	2	0	3	**1**	r3 is not 0, loop
Add 1 to r0	**3**	0	3	1	increment r0
Multiply 0 into r3	3	0	3	0	reset r3 to 0
Add r2 into r3	3	0	3	3	copy r2 into r3
Subtract r0 from r3	3	0	3	0	subtract 3 from r3
JNZ -4	3	0	3	*0*	r3 is 0
End of program	3	0	3	0	–

Example 2: Average Numbers in Memory This next exercise averages ten numbers in memory and stores the result in memory as well. More precisely, the program will load the numbers from memory in memory locations 0 through 9 (10 numbers total) and sum them. It will then divide the total by ten (keeping just the whole-number part) and store that value into memory at location 10.

The instructions for the program are:

- ADD r3 3 – set r3 to 3
- MULTIPLY r3 3 – increase r3 to 9
- ADD r3 1 – increase r3 to 10
- SUBTRACT r3 1 – decrement the memory address for load
- LOAD r0 r3 – load number into r0
- ADD r2 r0 – sum loaded number into total
- JNZ -3 – if r3 isn’t 0, jump back to load; the loop should sum memory addresses 0–9
- MULTIPLY r0 0 – clear r0

- ADD r0 3 – set r0 to 3
- MULTIPLY r0 3 – r0 is now 9
- ADD r0 1 – r0 is now 10
- DIVIDE r2 r0 – divide by 10
- STORE r2 r0 – Store value of r2 at location 10 in r0

Note that the biggest number we can add directly into a register at one time is 3. We can add bigger numbers if they are already stored in a register, but if not, 3 is the max. So in this example, to load in the number 10, for example, we have to get creative. You will see in the first couple of instructions that r3 is first set to 3, then multiplied by 3 to get 9, and then incremented by 1. This allows our r3 to be set to 10.

Of course real processors have bigger instructions and not so many limitations. However, even real processors sometimes have to be “creative” too and for similar reasons. There is always some kind of operation that cannot be done with a single or obvious instruction and combinations must be used.

It may also help to remember how LOAD and STORE work. LOAD has two registers as parameters. It uses the memory address stored in the second to load the value from memory and then stores it in the first register. So suppose that register r3 has the value 5 in it. The instruction “LOAD r0 r3” would go out to memory address 5, load the value there, and store it in r0.

Please try this using paper and pencil. It is not as hard as it might look. Just make sure you keep track of your registers! You will need to start with some numbers loaded into memory. I suggest using the numbers 1 through 10 (stored in memory locations 0 through 9). Using this configuration, this program will sum the numbers 1 through 10 and generate the average.

How did you do? If you did it right, and you used the numbers 1 through 10, you should have the value 5 stored in memory location 10.

If you struggled, here is a basic recap.

1. The first three instructions get a value of 10 into r3; this is one more than the maximum memory location for the numbers we want to add
2. This probably seems odd because right after adding 1 to r3 we subtract 1; but we need to have the subtract before the JNZ so we had to start r3 at 10 instead of 9; after subtracting 1 it is 9, which is the first memory location to load
3. The load instruction loads a value from memory based on the address in r3; the first time through this address is 9
4. The loaded number (in r0) is added into r2; because all values start at 0, this value is now the first loaded number
5. If r3 isn’t 0 (and it is currently 9), jump back 3 instructions
6. We are now repeating our loop; this time the load value is 8 (remember we subtracted one from r3?)
7. After loading the value in memory location 8, it is added into r2, so the total is now the first two numbers
8. The loop repeats until r3 is zero; the JNZ is checked right after loading the last number at memory location 0
9. After all loops are done, r2 will hold the total of the ten summed numbers
   We now need a register with the value of 10; Register

10. r0 is set to 0, then 3, then 9 and finally 10
11. The total in r2 is divided by the 10 value in r0 to get the average
12. We want to store the average in memory location 10; the value 10 is already in r0 and we use it as the memory address in the STORE instruction

---

C: Computer Communications and Networking

Computer Networks

A *network*, as used in this book and in most network security contexts, refers to computing resources, generically identified as *nodes*, that are connected together directly or indirectly across one or more communication media and can engage in intentional data exchange across the media. Note that media is the plural of medium. There are multiple physical media used for communications exchange but the two most common are radio waves (e.g., WiFi) and coaxial cables (e.g., Ethernet).

Modern networks almost always use an approach called *packet switching*. The telephone system used to use a system called *circuit switching*. (Imagine a switchboard operator plugging and unplugging cables to physically connect phone calls.) Although circuit switching still exists in some components, it has almost completely been replaced by packet switching. And packet switching has always been the primary means of *data* transmissions (as opposed to voice). For these reasons network security deals with packet switching networks, and they are the only type of network talked about in this book.

Some of the characteristics of a packet-switched network are:

1. Communications can be, and usually are, *general* rather than specific to a certain application
2. Devices of all types and purposes can participate in the network (the generality necessitates generic names such as “node”)
3. Nodes send data to each other in discrete blocks called packets
4. Nodes have addresses used to identity source and destinations of packets
5. Smaller networks can be aggregated into larger inter-networks by using a shared node connected to both networks
6. Routing enables packets to systematically move across interconnected networks to find destination addresses
7. Applications hosted on nodes communicate with each other over networks and inter-networks using protocols to harness the generic communications for their specific needs

I will expand on these points in the following paragraphs. It should be noted that *nodes* on a network may be physical computers, virtual computers, a process or program on a computer, a specialized networking devices, or any other computing component that can have an address on the network.

The Network Protocol Stack

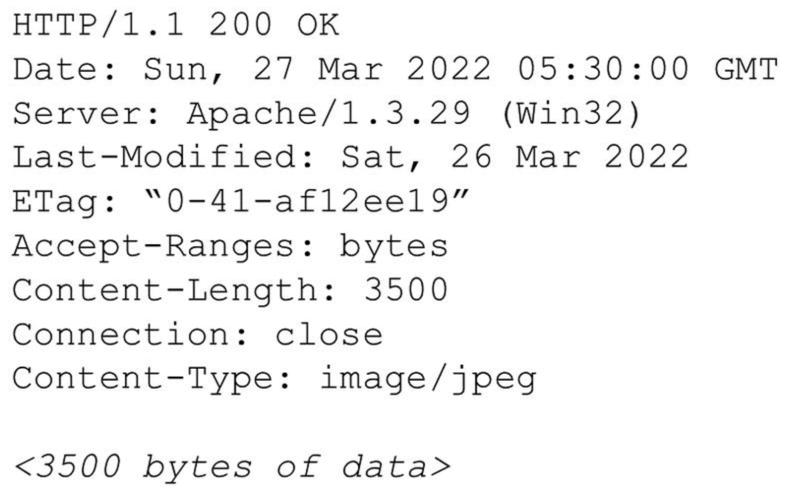
A networking *protocol* is a term used to describe the rules of communication between two nodes connected on a network. The protocol tells each machine how to form, transmit, process, and respond to messages. In everyday life, humans use “protocols” all the time in communicating with other humans. For example, when a person picks up the phone, he or she typically says “Hello?” While most of us wouldn’t think of this as a “rule”, the communications could break down pretty quickly if someone picked up a ringing phone and said, “Goodbye!” We use these rules to help manage the complexities of social interactions.

Computers are not that different. If one computer wants to “talk” to another computer, it must know how to say “hello,” what kind of messages to send, and when to say “goodbye.”

One of the most commonly used protocols today is HTTP, the HyperText Transfer Protocol. HTTP is the common protocol for the World Wide Web. When someone connects to a web site using a web browser, the browser sends a message, called an HTTP request, to the computer server that hosts the web site. The web site responds to the HTTP request with its own message, called an HTTP response. While many types of network messages are written in a binary format, HTTP messages are human readable. An example HTTP request is shown in Figure C-1 and an HTTP response in Figure C-2.

```
GET /sports HTTP/1.1
Host: cnn.com
User-Agent: Mozilla/5.0
Accept: text/html, text/xml, text/plain, image/jpeg
Accept-Language: en-us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
```

Figure C-1 An HTTP GET request is commonly sent by browsers to get a webpage



```
HTTP/1.1 200 OK
Date: Sun, 27 Mar 2022 05:30:00 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 26 Mar 2022
ETag: "0-41-af12ee19"
Accept-Ranges: bytes
Content-Length: 3500
Connection: close
Content-Type: image/jpeg

<3500 bytes of data>
```

Figure C-2 Webservers send back HTTP response messages to browsers

HTTP messages are described in greater detail in Chapter 9. The point of this example is to emphasize what a protocol is. The HTTP protocol is the set of rules that tells the browser how an HTTP request message is constructed and tells the server how to put together a legitimate response to the browser.

To better manage the complexities and configuration issues associated with computer communications, transmissions are typically governed by a collection of protocols called a protocol stack. Because there are so many necessary steps to global network communications, no one protocol should handle them all. Network architects create network protocols that chain together, with each element in the chain handling a different part of the communication processing. The chain of protocols is sometimes called a *network stack* [205, Chapter 1].

In the case of HTTP, for example, the request message sent by the browser only has information about the request and the context of the request. This is not enough to get the message to where it needs to be and it cannot be sent yet. Instead, the HTTP message is first passed to another protocol called TCP, the Transmission Control Protocol. One of the key functions of TCP is to create a *session* that enables multiple packets to be grouped together.

Packets

Data is not transmitted in a continuous stream. It is broken up into chunks that are generically called *packets*. How a packet is constructed and processed is defined by one or more protocols. A packet usually includes two major components: the data, also called the body or payload, and the metadata. The metadata contains information like how many bytes are in the packet, or when it was sent, or information about what type of message it is. The data is always sent or received in a sequence, so it can be depicted as a single line of binary data. If the metadata is prepended to the data, it is usually called a *header*, and most packets have a header. Sometimes metadata is appended to the data, and this is called a *trailer*.

Although HTTP messages by themselves are packets, they are generally not referred to as such. Because HTTP messages are easy to visualize because of their human readability, I will use the two messages, or packets, from Figures C-1 and C-2, to explain packets, headers, and bodies.

In HTTP, the packets are divided into a header and a body for data. Describing the HTTP packets as human readable is, in reality, only half true because the data is sometimes binary. The header is a human readable set of directives, each demarcated by a line separator, which is why even though the data is transmitted in a single sequence, it is displayed broken up into lines. After the header, which is indicated by a blank line (i.e., a line separator immediately following the previous line separator), is the body.

Looking at Figure C-1, you will notice *there is no body*. But that makes sense because when a browser sends a request to a web server, it is not transmitting any data. It is requesting data. In Figure C-2, there is a header and a body. The body is the actual data being sent back by the server. It contains the text and images that the browser will use to render the webpage. But the header is essential. It has information about whether the server found the requested information. If you have ever seen a message on your browser that says "404, Not Found", it was because it received an HTTP response that indicated the server could not find the URL. An HTTP response with a 404 error code is depicted in Figure C-3. On the very first line is the 404 code that a browser uses to know that the web page does not exist. Notice also that there is not data (body) in Figure C-3 because there is no data to send. Error messages like this one *can* send an error message as the body but it is not required.

```
HTTP/1.1 404 NOT FOUND
Date: Sun, 27 Mar 2022 05:30:00 GMT
Server: Apache/1.3.29 (Win32)
Content-Length: 0
Connection: close
Content-Type: text/html
```

Figure C-3 The 404 "not found" response web servers send when the requested URL does not exist

When HTTP data is found, the header also has basic information about how much data there is. Referring back to Figure C-2, notice the line in the header that contains the words "Content-Length". This field tells the browser how many bytes of data are in the body. Without this information, the browser would have no way of knowing when the HTTP response packet ends.

All packets, like these HTTP packets, are more or less atomic, meaning each packet is meant to be processed as a unit. At the same time, however *there is no guarantee that two or more packets will be delivered together, in order, or at all*. Worse, the nature of computer networks means that when data is actually sent, the packets will be too large and will need to be broken up into smaller pieces. These pieces, just like any other packets, cannot be guaranteed to arrive together or in order or even at all.

HTTP cannot solve all of these problems by itself. Instead, HTTP is going to hand off processing to another protocol called TCP. Protocols can work together through a process called *encapsulation*. The idea is that an HTTP packet can be completely encapsulated within another packet, like TCP. This means that entire HTTP packet, header and all, becomes the *data* for the TCP packet. The TCP packet has its own metadata for its own operations in its own header. TCP does not need to know anything about the data. TCP, for example, does not need to know that the data it is carrying is HTTP. All TCP packets have to worry about is performing their assignment with the data they have been given (whatever it is).

Sessions

Not only can packets like HTTP be encapsulated in packets like TCP, but HTTP operations are ecnapsulated within TCP operations. What I mean is, HTTP is trying to request a page from a website and receive a response. That operation is encapsulated inside TCP's operations. TCP operations include creating a *session*.

A session, at least in this context, is the concept of all the data that is meant to be grouped together across packets. The TCP protocol is designed to create *reliable* sessions. TCP is responsible for making sure that data will be received, will be received in order, and received without error. As I explained, packets are not grouped together by the nature of networks. It requires a communications protocol to force them into a session.

TCP does this by putting indexing information into its headers as well as some error detection information. This information is transmitted in TCP packets. When those packets are received, TCP examines the indexing information to make sure packets are put in order. It also checks the error detection information to see if the packet needs to be discarded. If everything is correct, it sends back a separate acknowledgement about the data that was received. If the sender does not receive an acknowledgement within a specific period of time, it will *resend* the data.

This is really crucial. Although a browser and a web server are initiating requests and responses, separate TCP components process their own requests and responses in order to make sure that the HTTP messages are received and delivered. Also notice that there is a decapsulation process on the other side of encapsulation. Just as the HTTP message was encapsulated in TCP on the sender side, that same message is decapsulated on the receiver side *after* TCP has ensured that it is in order and correct. This process is depicted in Figure C-4.

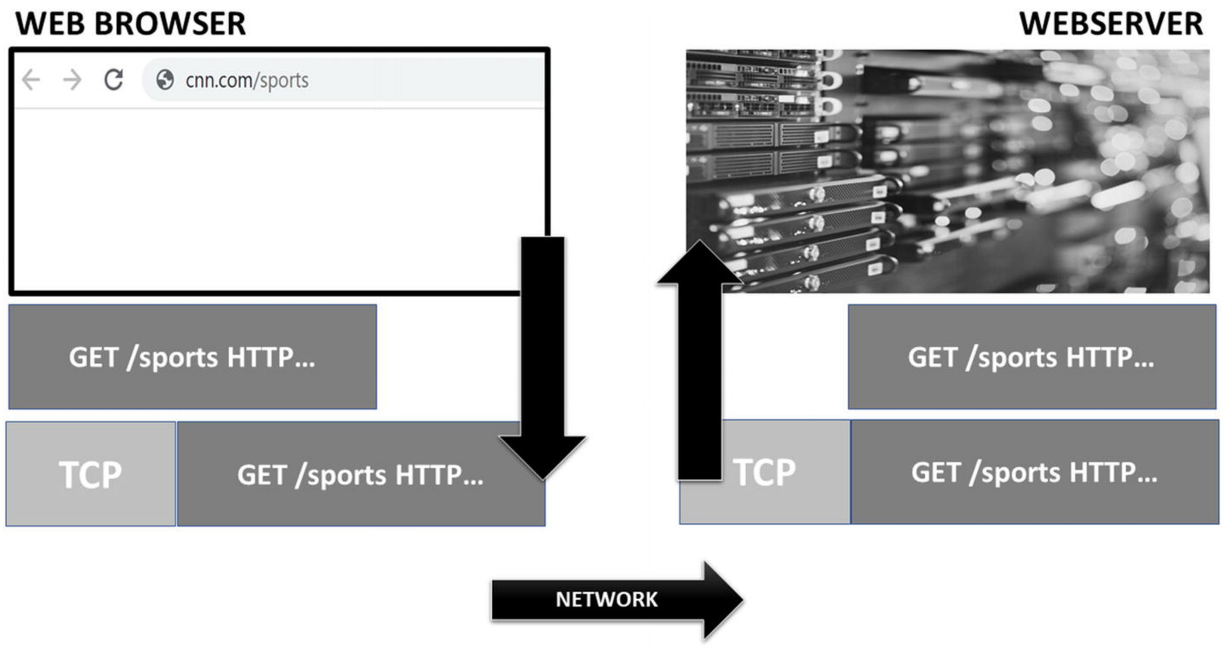


Figure C-4 Webservers send back HTTP response messages to browsers

Ports

TCP serves one other function besides creating sessions. It also performs an operation called *multiplexing*. Multiplexing means mixing data together from multiple sources into a single channel. The browser is not the only program transmitting data on the network, after all. There has to be a way to identify data meant for the browser, an email program, a chat program, video games, Internet music programs, and so forth. Also, a server, such as a web server, has to be able to distinguish between traffic from multiple clients! TCP solves this problem with port numbers. TCP port numbers are just a number between 0 and 65535.

When an outbound connection, such as the request from a browser for a webpage, is sent, an unused TCP port number is assigned as the “source port”. This port number enables the information sent back from the server to be sent to the correct program. For example, many users have multiple webpages open at once. Each one is a separate TCP session and each has its own randomly assigned source port number. This ensures that the responses go to the correct places. The outbound TCP session also includes a "destination port" indicating which application should process the data at the server. When data arrives, the process is demultiplexed, which means the single channel is split back up into its individual components. In Figure C-5, incoming packets are demultiplexed into data for the webserver and data for the Minecraft server running on the same machines. This figure only shows using the destination TCP ports but full demultiplexing uses other bits of information to, as described in the next paragraphs [205, Chapter 5].

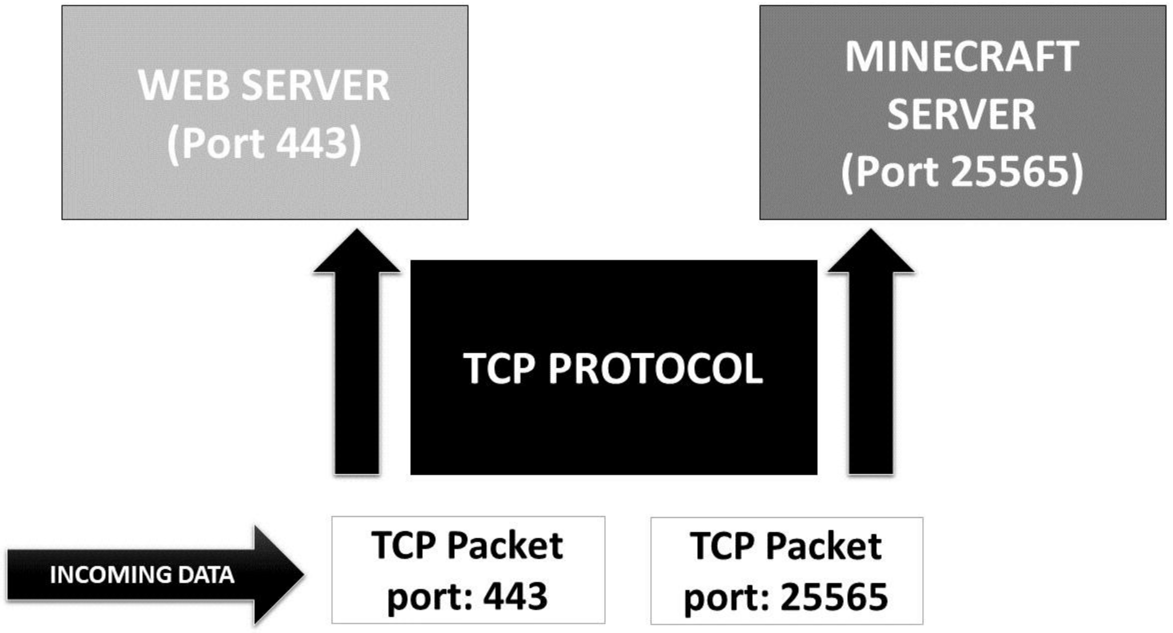


Figure C-5 The TCP protocol uses port numbers to multiplex data. When data is received, it is demultiplixed based, in part, off the destination ports shown here

In order to receive data, a server, such as a web server, will register or reserve an unused port (number) from the operating system in order to receive incoming requests. For commonly used applications, the server will typically use a "well-known" port. Web servers, for example, usually use port 80 for regular traffic (HTTP) and port 443 for encrypted traffic (HTTPS). When the browser sends data to a webserver, the browser indicates to TCP the destination port (e.g., 80 or 443 depending on whether the data is supposed to be encrypted or not). This port is put into the TCP header along with the randomly chosen source port before the TCP packet is sent over the network. When this data arrives at the server, the destination port is checked. The server that reserved, or claimed, that port, will create a unique session for this specific client. The server uses these unique sessions to differentiate the communications from different clients [205, Chapter 5].

Addresses

TCP, however, does not solve all network problems either. For example, TCP packets *do not have any address information*. Just like pen-and-paper letters put into envelopes, source and destination addresses are required to ensure that the letter arrives at the correct location. TCP port numbers are used for finding the right session on a given computer, but does not indicate which computer to arrive at in the first place. (In terms of an address on an envelope, a port number is somewhat similar to an apartment or suite number). So TCP is not enough. We need another protocol with addressing.

Enter the Internet Protocol (IP). As mentioned in the summary on packet-switched networking, there is the concept of a network and a collection of networks (inter-network). The IP protocol is designed to enable communications across interconnected networks. The global Internet (note the capital “I”) is *the* interconnected network that almost everyone in the world interacts with. For simplicity, I will stop talking about internets (small “i”) and only talk about the Internet (capital “I”) from this point on. While internets exist that are not connected to the Internet, it is such a small fraction that it is not worthwhile to spend time trying to distinguish between the two. The technology is the same anyway.

The IP protocol provides IP addresses, which are global addresses used for routing messages. Although we typically use domain names on the Internet (e.g., “cnn.com”), these domain names are converted into IP addresses for actually getting packets to the right places. There are two versions of the IP protocol in use. These are version 4 (IPv4) and version 6 (IPv6). An IPv4 address is comprised of four numbers, each between 0 and 255. The numbers are dotted together like this:

```
192.168.0.1
```

An IPv6 address is more complicated. An IPv6 address has 8 4-digit hexadecimal numbers concatenated with colons. Like this:

```
fdf8:f53b:82e4::53
```

In IPv6, leading zeros are omitted so 0053 is shown as 53.

For understanding the concepts in this book it is not necessary to understand much of these addresses. All that matters at this point is to understand that IP addresses are used to *route* data all over the world in the Internet. The IP protocol is designed to bridge individual networks.

Network Structures

At this point, it is a good time to explain the different levels of networking. A *Local Area Network*, or LAN, is comprised of nodes all connected on the same medium. The medium could be radio waves, such as WiFi. Another medium are a system of coaxial cables, such as Ethernet. The idea behind a LAN is that all of the nodes are able to talk to each other directly. There is no need for routing because all devices on the LAN can directly talk to all other devices on the LAN. The Internet, on the other hand, is a *Wide Area Network*.

Another requirement for a LAN is that broadcast is possible. For various reasons, it is sometimes necessary to “flood” a network with a particular message. LANs usually have a special mechanism for broadcasting a message to all devices on the LAN. For this reason a LAN is sometimes said to be a *broadcast domain* and all the computers on the LAN are part of the same broadcast domain.

Clearly it is impossible to have all of the computers in the world on the same LAN. But it is possible to create devices that have the capacity to connect to two different LANs at the same time. That is, the device is a node on *both* networks and can communicate on both media. Such a device interconnects the two LANs and acts as a *gateway* between the two. As a gateway device, it can receive data from one network and route it over to the other.

As with TCP encapsulating application protocols like HTTP, IP encapsulates TCP. So in the example of a browser sending an HTTP message, the HTTP message is encapsulated in a TCP packet. And the TCP packet is encapsulated in an IP packet. The IP packet has a source IP address and a destination address in its header. This enables the IP packet to be delivered across the Internet, and then the TCP packet can be extracted and processed. This double encapsulation is shown in Figure C-6.

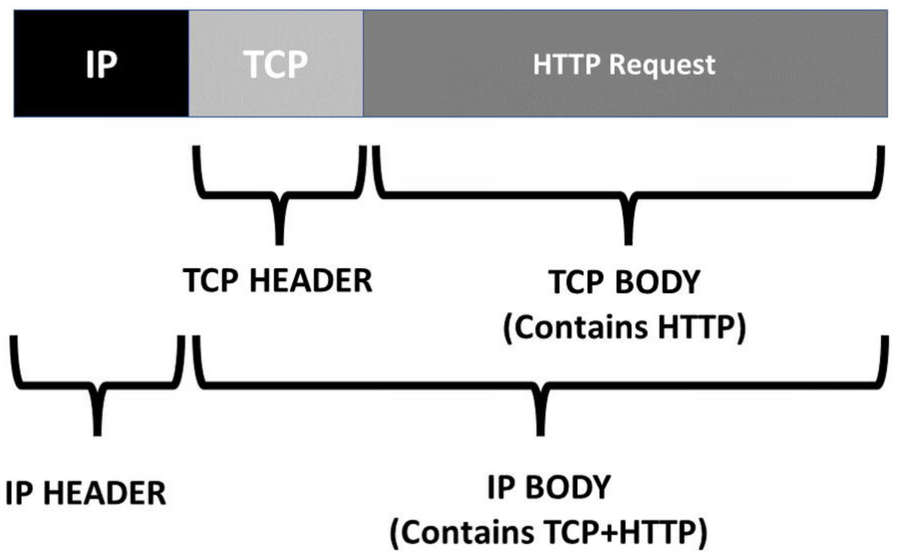


Figure C-6 An HTTP GET request encapsulated as the data of a TCP packet. The entire TCP packet (including the HTTP GET request) is encapsulated as the data of the IP packet

Together, TCP and IP are capable of connecting any two processes anywhere on the Internet. In computer science, this is sometimes described as a *logical* connection. The word "logical" is often used to describe something that does not exist as a purely physical element but has, instead, been built, at least in part, using computer commands, instructions, and protocols (i.e., computer "logic"). For example, physical hard drives can be used by computers to simulate *multiple* hard drives. Each one of the simulated drives is called a *logical drive*. Similarly, a browser and a website on opposite sides of the world do not have a direct, physical connection between the two processes. It is a logical connection created by the combination of the TCP and IP protocols.

The individual LANs also have their own protocols. There is a WiFi protocol for wireless LANs and an Ethernet protocol for wired LANs as well. These protocols enable communications between the devices directly on the LAN. The LAN has to have its own addressing scheme, which means that a node has both an IP address and a LAN address. The LAN address is called a Media Access Control (MAC) address.

MAC addresses are used exclusively for communicating between the nodes on the LAN. Like all the other protocols, LAN protocols encapsulate the higher (IP protocol) data. This enables the data to be sent around the LAN and then extracted for processing. If the data needs to go off the LAN, the IP data is encapsulated into a MAC packet, transmitted to a gateway. The data is then extracted and re-encapsulated in a MAC packet for the other network before being transmitted on the new medium (Figure C-7).

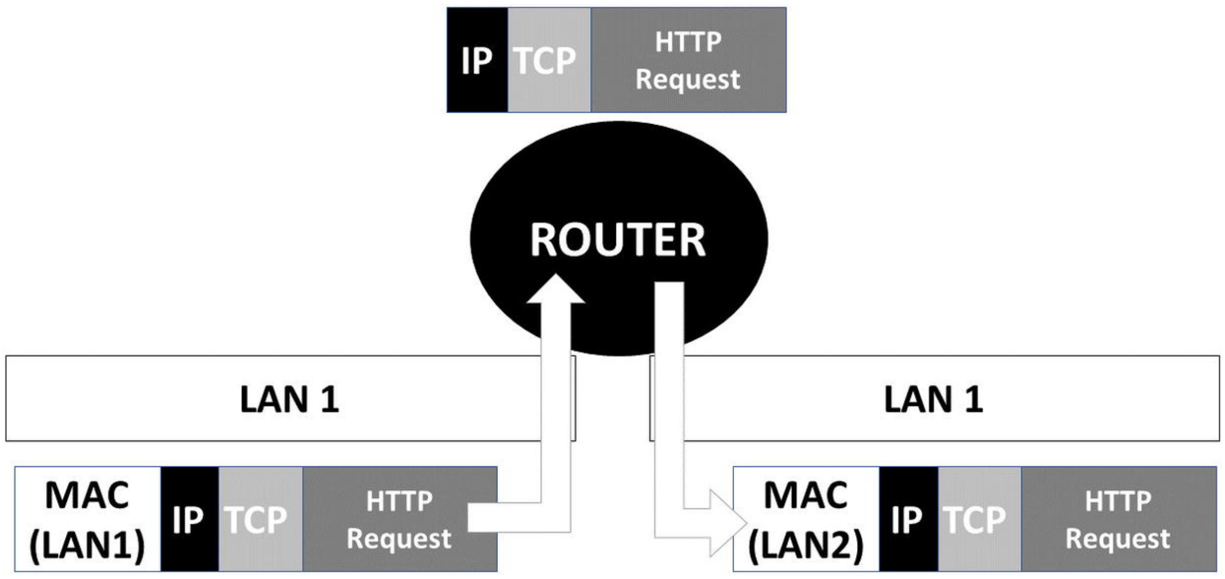


Figure C-7 A gateway is connected to two LANs, each with ther own LAN medium. The gateway can route a packet from one to the other, but must remove the MAC encapsulation for the first network and reencapsulate with a MAC packet for the second

Protocol Stacks

This may seem really complicated. For example, why can't HTTP do everything? Why do we need to do all of this protocol encapsulation and so forth?

First of all, trying to create HTTP to do all of these steps together would be a nightmare. The complexity of the implementation would be significantly higher which is typically associated with more bugs and security vulnerabilities.

Second, without modularity, the system is too brittle and cannot easily be reconfigured. HTTP is typically encapsulated by TCP. But there are other application protocols that do not use TCP and instead use an alternative called UDP. UDP, like TCP, has ports so that data can still be multiplexed and demultiplexed. Unlike TCP however, UDP does not try to create sessions, reorder packets, or resend lost packets. UDP is terrible for things like file downloads, which is why it is not used for most HTTP traffic.[1] However, real-time video or audio generally prefer UDP because resending data will generally cause more problems for these kinds of applications than it will fix.

There are even some systems that need to be able to use TCP or UDP depending on circumstances. By having a modular protocol stack, different protocols can be inserted at different levels. Without modularity, everything would have to be rewritten from top to bottom to accomodate a change in the middle.

Another example of modularity is the IP protocol. You will recall that there are two versions of IP: IPv4 and IPv6. By having a modular protocol stack, IPv4 and IPv6 can be swapped in and out without impacting the other protocols in the stack.

This type of approach permits each protocol to focus on its own job. HTTP does not have to deal with ensuring delivery or Internet routing or communicating with other devices on the WiFi. It treats everything below it as a transportation service and does not have to know or be responsive to internal implementation details.

The OSI Model

Because there can be different approaches for solving similar problems, the different layers of the protocol stack are conceptually assigned different responsibilities. Whatever protocols exist at that layer have to implement those responsibilities. For example, TCP and UDP are protocols that exist at the same layer. Although they solve problems differently, they are both intended to operate above a layer that solves routing and interconnectivity, but below a layer of application data.

A conceptual reference model for network stacks was created in the 1980s called the Open Systems Interconnection model (OSI) model. This model did not identify specific protocols such as TCP or IP, but rather defined what a protocol at a specific numbered layer should do. The OSI model defined 7 layers as shown in Figure C-8. The space in the figure is too limited to fully illustrate the encapsulation as you move down the stack, the size for a packet increases. This is, of course, because as data moves down the stack, additional header data must be added by each successive protocol. On the receiving side, this process is reversed as headers are stripped off until only the application data remains.

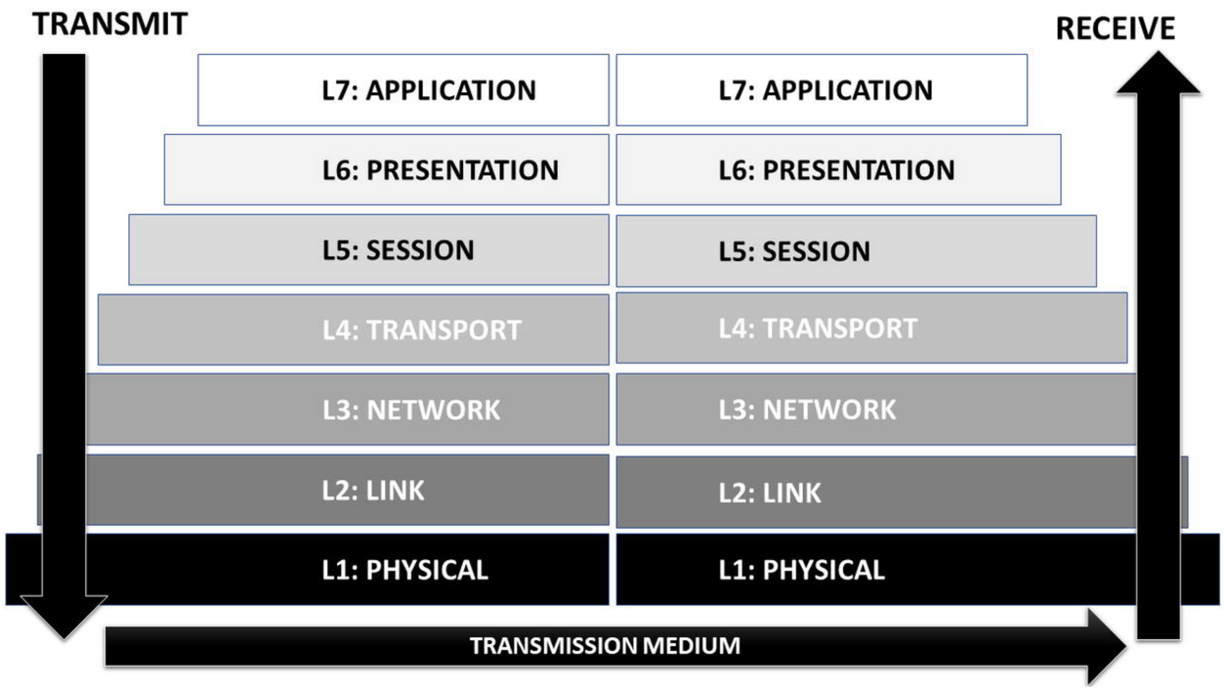


Figure C-8 The OSI model of an idealized protocol stack

The OSI model is an idealistic concept. In our HTTP example, we did not describe seven layers. Within the current Internet architecture, most networking applications and devices deal with only layer 2[2] (such as WiFi and Ethernet), layer 3 (IP), layer 4 (TCP and UDP), and layer 7 (application data such as HTTP). Nevertheless, OSI is considered the standard way of discussing protocols and networking stacks and the layer numbers are still used. So, for example, TCP is considered a layer 4 protocol, and HTTP is considered a layer 7 protocol. Even though UDP is also a layer 4 protocol, TCP is so commonly used that this model is often referred to as the TCP/IP stack. This stack is visualized in Figure C-9.

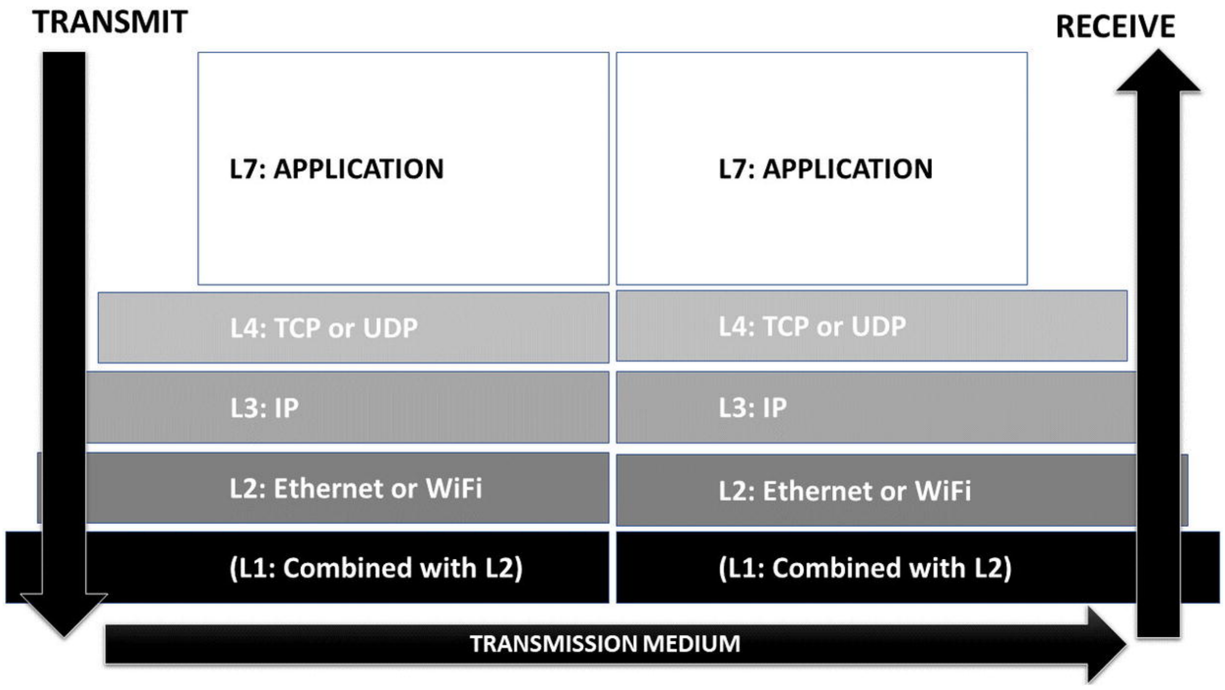


Figure C-9 The TCP/IP protocol stack

The amazing generality of packet switched networks enables nodes to combine different applications together in order to perform more complicated tasks. A good example of this is the Domain Name System (DNS). I mentioned DNS with respect to IP addresses. An IP address is required to connect one computer to another on the Internet and yet you do not have to enter an IP address into a browser. Having to use an IP address would be problematic because IP addresses would be almost impossible to remember, domain names (e.g., “amazon.com”) may be assigned to *multiple* IP addresses for various reasons, and a domain name may need to change the IP address it is bound to.

DNS and DHCP

Because Internet communications are general, it is not necessary to build into the network a separate layer for converting names to addresses. Instead, the DNS server can run on the network like any other application. When a browser wants to connect to “google.com,” it first connects to a DNS server and looks up the IP address for google.com. Once it gets a response from the DNS name server with the IP address, it can then begin the actual HTTP transmission like the one described in this appendix’s example.

Of course, the browser has to have the DNS server’s IP address loaded into the system somewhere. For obvious reasons, the DNS server cannot have a domain name. But using a DNS system in this manner the browser only needs to be configured to have one IP address and all other addresses can be resolved from domain names.

To understand how DNS resolves domain names, it is necessary to start with how domain names are constructed. Domain names are hierarchical moving from right to left (even though they are read from left to right). So, taking “www.google.com” as an example, there are four divisions: the root, “com”, “google”, and “www”. Because DNS is hierarchical, there is are DNS servers (distributed around the world) that hold root DNS data with information about the next level.

The next level chunk in a domain name is called the *Top-Level Domain* or TLD. Each TLD has its own operator that handles primary administration of that level of the doman name system. VeriSign Global Registry Services is the operator for the .com TLD as of the time of this writing. VeriSign is responsible for operating DNS servers that can authoritatively identify all .com domains such as “google.com”, “amazon.com”, “netflix.com”, and so forth. Even though VeriSign operates the domain, most domain names are sold through resellers such as GoDaddy or Google Domains.

Organizations can have multiple subdomains. Google, for example, has a “www” subdomain and an “images” subdomain (“www.google.com” and “images.google.com”). Another lower set of DNS servers can authoritatively resolve these subdomains within an organization. A very small portion of this hierarchy is illustrated in Figure C-10.

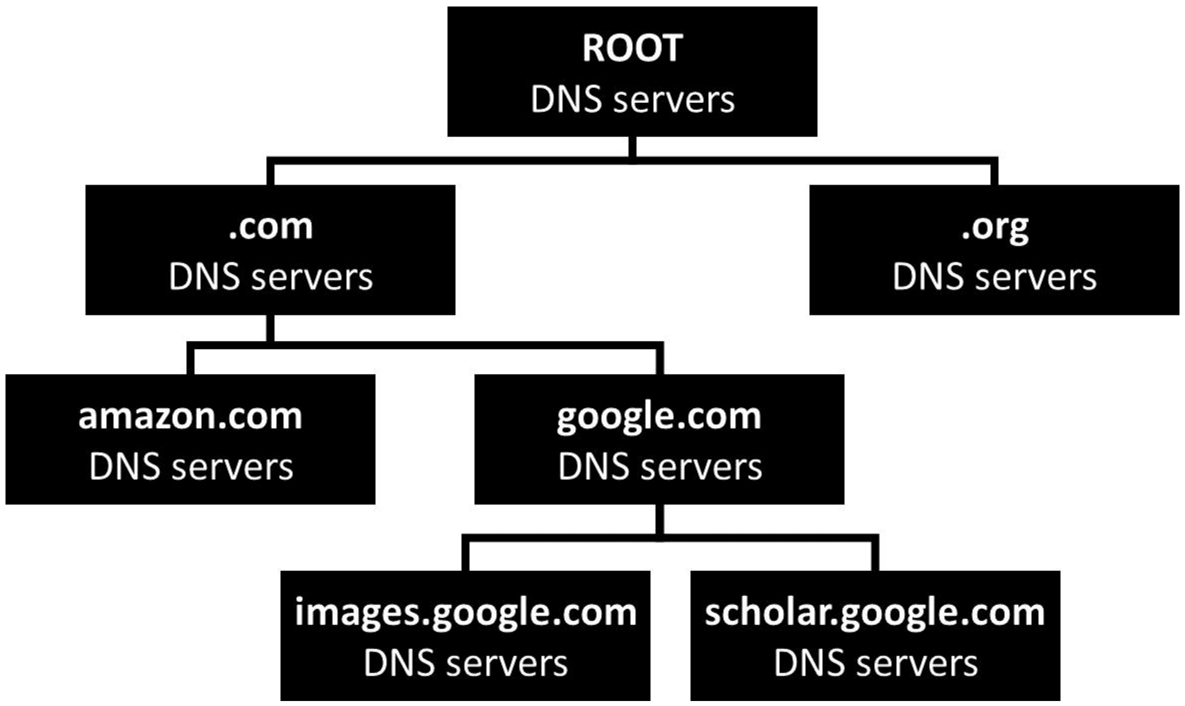


Figure C-10 A small subset of the DNS hierarchy. The root DNS servers provide authoritative lookups for the top-level domains, such as .com and .org. Then, servers for .com and .org identify nameservers within the domain, such as google.com. Google itself has internal DNS servers for subdomains such as images and scholar

So, in summary, when a computer needs to resolve, for example, “images.google.com”, it could start with a query to the root servers to obtain the addresses for the DNS servers for “.com.” Next, it would query the .com servers for “google.com.” Once it had Google’s DNS servers, it would query those for the IP address of “images.google.com”.

Not only is DNS hierarchical across a domain name, it is also recursive. Recursive is a computer science term that referse to doing the same operation repeatedly in a way that gets closer and closer to the solution until a final solution is reached.

In DNS, recursion is used to spread results across DNS servers throughout the entire world-wide Internet. DNS servers can be placed in any network including local networks. It is not uncommon for a DNS server to be co-located with a gateway. A computer starts by querying the local DNS server. If the local DNS server does not know the mapping of the domain to the address, it can do a recursive query to a DNS server further “up stream.” As a last resort, it can query the authoritative server for the mapping. Once it has a mapping, it can store it in its own tables for subsequent requests.

I mention briefly that every device on a LAN has to have an IP address that matches a pattern that is established for the LAN. For example, there are networks where all addresses must be prefixed with 192.168.1 (e.g., 192.168.1.100, 192.168.1.35). The way these patterns are used and expressed is not necessary for this book so I will not invest space going into more details. But what matters is that each device on a network has to have an IP address compatible with that specific LAN.

In order to make devices more plug-and-play, especially devices that might change networks regularly and would need a different IP address for each network, most devices are configured to request IP address information automatically. *Dynamic Host Configuration Protocol*, or DHCP, is used to retrieve an IP address along with identification of the correct gateway and local DNS servers. Networks operate a DHCP server to hand out IP addresses to requesting devices.

Client Server Architecture

You have probably heard the word “server” and you have seen it used already in this appendix. This word can be used to mean a lot of different things depending on context. Sometimes it is used to refer to an actual computer that is dedicated to acting as a server. But a more technical definition of *server* is a process (i.e., a running program) that listens on a computer network for incoming connections and then responds to subsequent requests from those connections. So a computer would only be referred to as a server if it has at least one process hosted on the machine that is a server process. For purposes of this book, I will use the word “server” to refer to the running process. A web server, therefore, is the process that responds to HTTP requests (e.g., from browsers). If I need to refer to the hosting computer system, I will use the term “server machine.” Note that within the protocol stack, layer-3 (IP protocol) has addresses for machines and layer-4 (TCP or UDP protocols) has ports for the process. Machines only need an address. Servers require a port.

The counterparty of a server is called a *client*. A client is a process that initiates a network connection to a server in order to make requests of the server. As with the term server, I will use the term client to refer to the process (e.g., a browser). If I need to refer to the hosting computer system, I will use the term “client machine.”

Because clients and servers are processes, it is possible to have a client and a server running on the same machine. Even though the devices are on the same machine, the client still makes a network connection to the server and the two processes exchange information using the same network stack (e.g., TCP, IP, etc.).

Distributed computing is any computer-based system that needs to have at least two components running as different nodes, connected cooperatively together over a network. A *client-server architecture* is one model of distributed computing. Typically, a server is configured with a sufficient amount of resources to service requests from some anticipated number of clients. The server will generally require more processor, memory, and storage resources than an individual client because a single server will have to provide service to potentially many clients at once. The server can also enable communications between the clients with each other, but the server mediates all such communications. This means that the server also controls client interactions and is responsible for enforcing “good behavior” and security policies.

There are other models of distributed computing, such as *peer-to-peer* networks. But the client-server model is the most commonly used configuration in the Internet.

For example, consider online banking. A bank configures a online bank webserver at a specific domain name (e.g., “bankofamerica.com”, “chase.com”, “wellsfargo.com”, etc.). The bank provisions a server to accept requests from millions of clients (browsers). The server must respond to many requests simultaneously and must have a sufficient amount of resources to do so. Clients may interact with each other like, for example, sending money from one person’s account to another person’s account. But the server mediates all of these operations.

References

1. Cortex XDR incident handling v3.
2. Crypto done right!
3. Duo auth API.
4. Owasp security knowledge framework.
5. Owasp top ten.
6. Phishing examples archive.
7.

Security summit alert: Renewed alert about phishing e-mail scam targeting payroll or human resource departments.

8. Software assurance maturity model.

9. Ssl/tls strong encryption: How-to.

10. Wstg—stable.

11. TA13-032: Oracle Java multiple vulnerabilities. 2 2013. Last updated: 19 Oct 2016.

12. Istr: Internet security threat report, Phrack vol. 22, Apr 2017. http://phrack.org/issues/49/14.html.

13. Crisis: Security vs performance. 04 2018.

14. Active deception to combat advanced threats. Technical report, Attivo Networks, 2019.

15. IEEE standard for authenticated encryption with length expansion for storage devices, 2019.

16. Istr: Internet security threat report, vol. 24, Feb 2019.

17. The quantum computer and its implications for public-key crypto systems. Technical report, Entrust Datacard, 2019.

18. Ransomware guide. Technical report, 09 2020.

19. CVE-2021-44228. 11 2021.

20. Forensic methodology report: How to catch NSO group’s pegasus. 07 2021.

21. Packet flow sequence in PAN-OS. 06 2021.

22. 2022 annual state of phishing report. Technical report, Cofense, 2022.

23. Application sandbox. 10 2022.

24. CVE-2022-45808. 11 2022.

25. Intel SGX deprecation review. 01 2022.

26. The state of ransomware 2022. Technical report, Sophos, 04 2022.

27. What are computer viruses, 2022.

28. Mcafee endpoint security 10.6.0—threat prevention product guide—windows. Technical report, Trellix, 2023.

29. Acronis. 2018. A guide to ransomware and how acronis active protection can help.

30. Adkins, A. 1997. Secret war: The navajo code talkers in world war II. *New Mexico Historical Review* 72(4): 10.

31. Agarwal, A., S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom. 2022. Spook.js: Attacking chrome strict site isolation via speculative execution. In *43rd IEEE Symposium on Security and Privacy (S&P'22)*.

32. Aggarwal, C.C. 2011. *Social Network Data Analytics*. Springer. [zbMATH]

33. Ahmed, J., H.H. Gharakheili, Q. Raza, C. Russell, and V. Sivaraman. 2019. Real-time detection of DNS exfiltration and tunneling from enterprise networks. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 649–653.

34. Ahmed, M., L. Sharif, M. Kabir, and M. Al-Maimani. 2012. Human errors in information security. *International Journal of Advanced Trends in Computer Science and Engineering* 1(3): 82–87.

35. Aleph One (Unknown Author). 1996. Smashing the stack for fun and profit. 7(49): 11. newsgroup article or web page (?).

36. Almeshekah, M.H., E.H. Spafford, and M.J. Atallah. 2013. Improving security using deception. Technical Report 13, Purdue University, 11. CERIAS Tech Report 2013-13.

37. Alshamrani, A., S. Myneni, A. Chowdhary, and D. Huang. 2019. A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities. *IEEE Communications Surveys and Tutorials* 21(2): 1851–1877.

38. Alessandro Bulgarelli, M.A., and M.C. Francesca Mazzoni. 2005. Honeyspam: Honeypots fighting spam at the source. In *Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI'05)*, Cambridge, MA, ed. by D. Katabi and B. Krishnamurthy. USENIX Association.

39. Anderson, R.J. 1993. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS'93)*, New York, 215–227. Association for Computing Machinery.

40. Anderson, R.J. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3 ed. Wiley Publishing.

41. Antonakakis, M., T. April, M. Bailey, M. Bernhard, E. Bursztein, J.

Cochran, Z. Durumeric, J.A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. 2017. Understanding the Mirai botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*, 1093–1110. USENIX Association.

42. Arm Limited, Cambridge, England. 2018. *Arm TrustZone Technology for the Armv8-M Architecture*, version 2.1 edition.

43. Arm Limited, Cambridge, England. 2022. *Introduction to the Armv8-M Architecture and Its Programmers Model*, version 1.0 edition

44. Armour, P. 2000. The five orders of ignorance. *Communications of the ACM* 43(10): 17–20.

45. Arntz, P. 2013. What is host intrusion prevention system (HIPS) and how does it work?

46. Arntz, P. 2021. Ryuk ransomware develops worm-like capability.

47. Badger, L., D. Sterne, D. Sherman, K. Walker, and S. Haghighat. 1995. Practical domain and type enforcement for unix. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, 66–77.

48. Barkley, J. 1995. Application engineering in health care. In *Proceedings of the 2nd Annual CHIN Summit*.

49. Barkley, J., K. Beznosov, and J. Uppal. 1999. Supporting relationships in access control using role based access control. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control (RBAC'99)*, New York, 55–65. Association for Computing Machinery.

50. Bazaliy, M., C. Neckar, G. Sinclair, and in7egral. 2016. Technical analysis of the pegasus exploits on IoS. Technical report, Lookout

51. Bell, D. 2005. Looking back at the Bell-La Padula model. In *21st Annual Computer Security Applications Conference (ACSAC'05)*

52. Bell, D.E., and L.J. LaPadula. 1973. Secure computer systems: Mathematical foundations. *Draft MTR, The MITRE Corporation*, 2.

53. Bellovin, S.M., and W.R. Cheswick. 1994. Network firewalls. *IEEE Communications Magazine* 32(9): 50–57.
[zbMATH]

54. Belokosztolszki, A. 2004. Role-based access control policy administration. Technical Report 586, University of Cambridge. UCAM-CL-TR-586.

55. Beres, Y., A. Baldwin, M.C. Mont, and S. Shiu. 2007. On identity

assurance in the presence of federated identity management systems. In *Proceedings of the 2007 ACM Workshop on Digital Identity Management (DIM'07)*, New York, 27–35. Association for Computing Machinery.

56. Berners-Lee, T. 1996. WWW: Past, present, and future. *Computer* 29(10): 69–77.

57. Bernstein, D.J. 2005. Salsa20 design. *Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago, Chicago*.

58. Beyer, B.A.E., C.M. Beske, J. Peck, and M. Saltonstall. 2017. Migrating to beyondcorp: Maintaining productivity while improving security. *Login***42**(2). ISSN 1044-6397.

59. Biba, K.J. 1977. Integrity considerations for secure computer systems. Technical report, MITRE Corporation.

60. Bishop, M. 2019. *Computer Security Art and Science*, 2nd ed. Addison-Wesley Professional.

61. Boshmaf, Y., I. Muslukhov, K. Beznosov, and M. Ripeanu. 2013. Design and analysis of a social botnet. *Computer Networks* 57(2): 556–578.

62. Bossler, A., and T. Holt. 2009. On-line activities, guardianship, and malware infection: An examination of routine activities theory. *International Journal of Cyber Criminology (IJCC) ISSN* 3: 974–2891.

63. Boyd, D.M., and N.B. Ellison. 2007. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication* 13(1): 210–230.

64. Braden, R., and J. Postel. 1987. Requirements for internet gateways (1009).

65. Brossard, D., G. Gebel, and M. Berg. 2017. A systematic approach to implementing ABAC. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control (ABAC'17)*, New York, 53–59. Association for Computing Machinery.

66. Bruce, S. 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.-2nd*. Wiley.
[zbMATH]

67. Bush, G., P. Luu, and M.I. Posner. 2000. Cognitive and emotional influences in anterior cingulate cortex. *Trends in Cognitive Sciences* 4(2): 215–222.

68. Cartwright, A., E. Cartwright, J. MacColl, G. Mott, S. Turner, J. Sullivan,

and J.R. Nurse. 2023. How cyber insurance influences the ransomware payment decision: theory and evidence. *The Geneva Papers on Risk and Insurance-Issues and Practice*, 1–32 (48).

69. Cawthra, J., M. Ekstrom, L. Lusty, J. Sexton, and J. Sweetnam. 2020. Data integrity: Detecting and responding to ransomware and other destructive events. Special Publication (NIST SP) 1800-26, National Institute of Standards and Technology, Gaithersburg.

70. Cerdeira, D., N. Santos, P. Fonseca, and S. Pinto. 2020. SoK: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, 1416–1432.

71. Chatel, M. 1996. Classical versus transparent IP proxies (1919).

72. Chaumette, S., O. Ly, and R. Tabary. 2011. Automated extraction of polymorphic virus signatures using abstract interpretation. In *2011 5th International Conference on Network and System Security*, 41–48. IEEE.

73. Cheswick, W.R., S.M. Bellovin, and A.D. Rubin. 2003. *Firewalls and Internet Security*, 2nd ed. Addison Wesley Professional.

74. Chung, D. Ferraiolo, D. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. 2019. Guide to attribute based access control (ABAC) definition and considerations. Special Publication (NIST SP) 800-162, National Institute of Standards and Technology, Gaithersburg.

75. Cluley, G. 2013. The dying art of computer viruses. *Virus Bulletin* 2. www.virusbulletin.com/virusbulletin/2013/08/dying-art-computer-viruses.

76. Cohen, F. 1987. Computer viruses: Theory and experiments. *Computers and Security* 6(1): 22–35.
[MathSciNet]

77. Cohen, F. 2004. The use of deception techniques: Honeypots and decoys.

78. Columbus, L. 2019. 74% of data breaches start with privileged credential abuse.

79. Community, C. 2023. 3cx desktopapp security alert.

80. Community, C. 2023. Threat alerts from sentinelone for desktop update initiated from desktop client.

81. Cook, D.M., B. Waugh, M. Abdipanah, O. Hashemi, and S.A. Rahman. 2014. Twitter deception and influence: Issues of identity, slacktivism, and puppetry. *Journal of Information Warfare* 13(1): 58–71.

82.

Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. 1998. StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks. In *7th USENIX Security Symposium (USENIX Security 98)*, San Antonio. USENIX Association.

83. Cowan, C., P. Wagle, C. Pu, S. Beattie, and J. Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX'00)*, vol. 2, 119–129. IEEE.

84. Craciun, V.C., A. Mogage, and E. Simion. 2019. Trends in design of ransomware viruses. In *Innovative Security Solutions for Information Technology and Communications*, ed. J.-L. Lanet and C. Toma, 259–272. Springer International Publishing.

85. Cranor, L. 2016. Time to rethink mandatory password changes.

86. CrowdStrike. 2023. Crowdstrike prevents 3cxdesktopapp intrusion campaign.

87. Curry, S. 2019. Cracking my windshield and earning $10,000 on the tesla bug bounty program.

88. Cushing, T. 2017. How minecraft led to the Mirai botnet. *TechDirt*.

89. Dame-Boyle, A. 2015. EFF at 25: Remembering the case that established code as speech.

90. Dang, T.H., P. Maniatis, and D. Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (ASIA CCS'15), New York, 555–566. Association for Computing Machinery.

91. Davies, D.W., and W.L. Price. 1984. *Security for Computer Networks: An Introduction to Data Security in Teleprocessing and Electronic Funds Transfer*. New York: Wiley.

92. Devlin, R. 2016. Data loss prevention—Devlin. Technical report, SANS Institute.

93. Dhaka, D., and M. Mehrotra. 2019. Cross-domain spam detection in social media: A survey. In *Emerging Technologies in Computer Engineering: Microservices in Big Data Analytics: Second International Conference, ICETCE 2019*, Jaipur, 98–112. Springer.

94. DiMaggio, J. 2022. *The Art of Cyberwarfare: An Investigator's Guide to Espionage, Ransomware, and Organized Cybercrime*. No Starch Press.

95. Dotzon, C. 2019. *Practical Cloud Security: A Guide for Secure Design and Deployment*. Sebastopol: O'Reilly Media.

96. Douceur, J.R. 2002. The sybil attack. In *Peer-to-Peer Systems: First International Workshop (IPTPS 2002)*, Cambridge, MA, 251–260. Springer.

97. Duckett, C. 2020. Zoom concedes custom encryption is substandard as citizen lab pokes holes in it.

98. Dullien, T. 2020. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* 8(2): 391–403.

99. Dyskstra, J., and D. Hough. 2021. Action bias and the two most dangerous words in cybersecurity.

100. Economy, E.C. 2018. The great firewall of China: Xi Jinping's internet shutdown. *The Guardian*

101. Elgamal, D.T., and K.E. Hickman. 1995. The SSL Protocol. Internet-Draft draft-hickman-netscape-ssl-00, Internet Engineering Task Force, Work in Progress.

102. Elsworthy, E. 2020. Australia fires see spike in fraudster behaviour.

103. Federal Bureau of Investigation. 2018. The Morris worm: 30 years since first major attack on the internet.

104. Ferraiolo, D.F., J.F. Barkley, and D.R. Kuhn. 1999. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)* 2(1): 34–64.

105. Ferraiolo, D.F., and D.R. Kuhn. 1992. Role-based access controls. In *15th National Computer Security Conference*, 554–563. National Institute of Standards and Technology.

106. Ferrara, E. 2019. The history of digital spam. *Communications of the ACM* 62(8): 82–91.

107. Findley, N. 1993. *Shadowplay*. Penguin Group.

108. FingerpringJS, Inc. Frequently asked questions.

109. Force, J.T. 2018. Risk management framework for information systems and organizations. Special Publication (NIST SP) 800-37r2, National Institute of Standards and Technology, Gaithersburg.

110. Franklin, B. Benjamin Franklin quotes.

111. Garber, M. 2014. There are 64 tiananmen terms censored on China’s internet today. *The Atlantic*.

112. Gartner. Endpoint detection and response (EDR) solutions reviews and ratings.

113. Geerthik, S. 2013. Survey on internet spam: Classification and analysis. *International Journal of Computer Technology and Applications* 4(3): 384.

114. Grassi, P., J. Fenton, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y.-Y. Choong, K. Greene, and M. Theofanos. 2020. Digital identity guidelines: Authentication and lifecycle management. Special Publication (NIST SP) 800-63B, National Institute of Standards and Technology, Gaithersburg.

115. Green, M. 2011. How (not) to use symmetric encryption.

116. Green, M. 2023. A few thoughts on cryptographic engineering.

117. Greenberg, A. 2021. The full story of the stunning RSA hack can finally be told.

118. Griffiths, J. 2020. ‘I love you’: How a badly-coded computer virus caused billions in damage and exposed vulnerabilities which remain 20 years on. *CNN Business*.

119. Grimes, R.A. 2020. *Hacking Multifactor Authentication*. Wiley.

120. Grother, P., W. Salamon, and R. Chandramouli. 2013. Biometric specifications for personal identity verification. Special Publication (NIST SP) 800-76r2, National Institute of Standards and Technology, Gaithersburg.

121. Group, J.T.F.T.I.I.W. 2020. Security and privacy controls for federal information systems and organizations. Special Publication (NIST SP) 800-53r5, National Institute of Standards and Technology, Gaithersburg.

122. Gryaznov, D. 1999. Scanners of the Year 2000: Heuristics. In *Proceedings of the 5th International Virus Bulletin*.

123. Guerrero-Saade, J.A. 2023. Smoothoperator: Ongoing campaign trojanizes 3cxdesktopapp in supply chain attack.

124. Guttman, B., and E.A. Roback. 2017. An introduction to computer security: The NIST handbook. Special Publication (NIST SP) 800-12r1, National Institute of Standards and Technology, Gaithersburg.

125. Hadnagy, C. 2018. *Social Engineering: The Science of Human Hacking*,

2nd ed. Wiley.

126. Halderman, J.A., and E.W. Felten. 2006. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th Conference on USENIX Security Symposium – Volume 15 (USENIX-SS'06)*. USENIX Association.

127. Hanson, R. 2010. Confronting the negativity bias.

128. Hassold, C. 2022. The victimology of ransomware: 4,200 ransomware victims and counting.

129. Hauk, C. 2023. What is browser fingerprinting? How it works and how to stop it. *Pixel Privacy*.

130. Havenridge, J. 2015. Passwords are like underwear.

131. Hernacki, B., J. Bennett, and T. Lofgren. 2004. Symantec deception server experience with a commercial deception system. In *Recent Advances in Intrusion Detection*, ed. E. Jonsson, A. Valdes, and M. Almgren, 188–202. Berlin/Heidelberg: Springer.

132. Hilchenbach, B. 1997. Observations on the real-world implementation of role-based access control. In *Proceedings of the 20th National Information Systems Security Conference*, 341–352.

133. Hirstein, W., and V.S. Ramachandran. 1997. Capgras syndrome: A novel probe for understanding the neural representation of the identity and familiarity of persons. *Proceedings. Biological Sciences* 264: 437–444.

134. Hitchens, M., and V. Varadharajan. 2000. Design and specification of role based access control policies. *IEE Proceedings-Software* 147(4): 117–129.

135. Hoglund, G., and G. McGraw. 2004. *Exploiting Software*. Addison-Wesley Professional.

136. Honan, M. 2012. How apple and Amazon security flaws led to my epic hacking.

137. Howard, M., D. LeBlanc, and J. Viega. 2009. *24 Deadly Sins of Software Security*. McGraw-Hill.

138. Hu, C.T., D. Ferraiolo, R. Chandramouli, and D. Kuhn. 2017. *Attribute Based Access Control*. Norwood: Artech House.

139. Huffman, S. 2000. The navajo code talkers: A cryptologic and linguistic perspective. *Cryptologia* 24(4): 289–320.

140. Hunt, M.G., R. Marx, C. Lipson, and J. Young. 2018. No more FOMO: Limiting social media decreases loneliness and depression. *Journal of*

Social and Clinical Psychology 37(10): 751–768.

141. Internal Revenue Service. 2020. Irs warns against covid-19 fraud; other financial schemes.

142. ITL, C. 2019. Binary hardening in IoT products.

143. Johansson, J.M., and R. Grimes. 2008. The great debate: Security by obscurity. *TechNet Magazine*.

144. Judge, P., D. Alperovitch, and W. Yang. 2005. Understanding and reversing the profit model of spam (position paper). In *Proceedings of the 4th Workshop on the Economics of Information Security*.

145. Junod, P. 2001. On the complexity of Matsui's attack. In *Selected Areas in Cryptography*, ed. S. Vaudenay and A.M. Youssef, 199–211. Berlin/Heidelberg: Springer.

146. Kabakus, A.T., and R. Kara. 2017. A survey of spam detection methods on Twitter. *International Journal of Advanced Computer Science and Applications* 8(3): 29–38.

147. Kahn, D. 1996. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner.

148. Karami, M., and D. McCoy. 2013. Understanding the emerging threat of DDoS-as-a-Service. In *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 13)*, Washington, DC. USENIX Association.

149. Karger, P., and R. Schell. 2002. Multics security evaluation: Vulnerability analysis. In *18th Annual Computer Security Applications Conference, 2002. Proceedings*, 127–146.

150. Kennedy, D.M. 1999. Victory at sea. *The Atlantic Monthly* 51–76. www.theatlantic.com/magazine/archive/1999/03/victory-at-sea/306272/.

151. Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. SeL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, New York, 207–220. Association for Computing Machinery.

152. Knudsen, L.R., and J.E. Mathiassen. 2001. A chosen-plaintext linear attack on des. In *Fast Software Encryption*, ed. G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, 262–272. Berlin/Heidelberg: Springer.

153. Kocher, P., J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019.

Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

154. Kominski, I. 1989. Computer use in the United States: 1989. Technical Report 171, U.S. Department of Commerce, Bureau of the Census, Washington, DC.

155. Krebs, B. 2016. Seagate phish exposes all employee W-2s.

156. Kshetri, N., and J. Voas. 2017. Do crypto-currencies fuel ransomware? *IT Professional* 19(5): 11–15.

157. Lab, T.D. Why do we prefer doing something to doing nothing? The action bias, explained.

158. Lampson, B. 1993. *Principles for Computer System Design*. New York: Association for Computing Machinery.
[zbMATH]

159. Landwehr, C.E. 1981. Formal models for computer security. *ACM Computing Surveys* 13(3): 247–278.

160. Larochelle, D., and D. Evans. 2001. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 2001)*, Washington, DC. USENIX Association.

161. Leveson, N.G., and C.S. Turner. 1993. An investigation of the therac-25 accidents. *Computer* 26(7): 18–41.

162. Lhee, K.-S. and S.J. Chapin. 2003. Buffer overflow and format string overflow vulnerabilities. *Software Practice and Experience* 33(5): 423–460.

163. Lipp, M., M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*.

164. Luo, T., Z. Xu, X. Jin, Y. Jia, and X. Ouyang. 2017. Iotcandyjar: Towards an intelligent-interaction honeypot for IoT devices, 1–11.

165. Luo, X., R. Brody, A. Seazzu, and S. Burd. 2011. Social engineering: The neglected human factor for information security management. *Information Resources Management Journal (IRMJ)* 24(3): 1–8.

166. Luostarinen, K., A. Naumenko, and M. Pulkkinen. 2006. Identity and access management for remote maintenance services in business networks. In *Project E-Society: Building Bricks*, ed. R. Suomi, R. Cabral,

J.F. Hampe, A. Heikkilä, J. Järveläinen, and E. Koskivaara, Boston, 1–12. Springer.

167. Luotonen, A., and K. Altis. 1994. World-wide web proxies. *Computer Networks and ISDN Systems* 27(2): 147–154. Selected Papers of the First World-Wide Web Conference.

168. Maffeo, S.E. 2000. *Most Secret and Confidential, Intelligence in the Age of Nelson*. Naval Institute Press.

169. Mairh, A., D. Barik, K. Verma, and D. Jena. 2011. Honeypot in network security: A survey. In *Proceedings of the 2011 International Conference on Communication, Computing and Security (ICCCS'11)*, New York, 600–605. Association for Computing Machinery.

170. Mann, D.E., and S.M. Christey. 1999. Towards a common enumeration of vulnerabilities. In *2nd Workshop on Research with Security Vulnerability Databases*, West Lafayette.

171. Marczak, B., and J. Scott-Railton. 2016. The million dollar dissident. Technical Report 78, University of Toronto.

172. Marczak, B., and J. Scott-Railton. 2020. Move fast and roll your own crypto: A quick look at the confidentiality of zoom meetings. Technical Report 126, University of Toronto.

173. Marinho, T. 2018. Ransomware encryption techniques. *Medium*.

174. Markoff, J. 1988. Author of computer 'Virus' is son of N.S.A. expert on data security. *The New York Times*.

175. Matsui, M. 1994. Linear cryptanalysis method for des cipher. In *Advances in Cryptology—EUROCRYPT'93*, ed. T. Helleseth, 386–397. Berlin/Heidelberg: Springer.

176. McGilvray, D. 2021. *Executing Data Quality Projects: Ten Steps to Quality Data and Trusted InformationTM* . Elsevier Inc.

177. McRae, C.M., and R.B. Vaughn. 2007. Phighting the phisher: Using web bugs and honeytokens to investigate the source of phishing attacks. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 270c.

178. Meadows, C. 1995. Applying the dependability paradigm to computer security. In *Proceedings of 1995 New Security Paradigms Workshop*, 75–79.

179. Menezes, A.J., S.A. Vanstone, and P.C.V. Oorschot. 1996. *Handbook of Applied Cryptography*, 1st ed. Boca Raton: CRC Press, Inc.

[zbMATH]

180. Merriam-Webster. Adversary.

181. Merriam-Webster. Identity.

182. Metz, C. 1999. AAA protocols: Authentication, authorization, and accounting for the internet. *IEEE Internet Computing* 3(6): 75–79.

183. Microsoft Corporation. 2023. Memory integrity and VBS enablement.

184. Microsoft Corporation. 2023. Virtualization-based security system resource protections.

185. Microsoft Corporation. 2023. Virtualization-based security (VBS).

186. Mitnick, K.D., and W.L. Simon. 2003. *The Art of Deception: Controlling the Human Element of Security*. Wiley.

187. Morrison, S. 2020. Hackers stole $13,103.91 from me. Learn from my mistakes.

188. Moussaileb, R., B. Bouget, A. Palisse, H. Le Bouder, N. Cuppens, and J.-L. Lanet. 2018. Ransomware's early mitigation mechanisms. In *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES 2018)*, New York. Association for Computing Machinery.

189. Mphago, B., O. Bagwasi, B. Phofuetsile, and H. Hlomani. 2015. Deception in dynamic web application honeypots: Case of Glastopf. In *Proceedings of the International Conference on Security and Management (SAM'15)*, Las Vegas, ed. by K. Daimi and H.R. Arabnia.

190. Munson, R.V. 2001. Telling wonders: Ethnographic and political discourse in the work of herodotus.

191. Muttik, I. 1999. Macro viruses—Part 1. *Virus Bulletin* 13–14. www.virusbulletin.com/uploads/pdf/magazine/1999/199909.pdf.

192. Najm, Z., D. Jap, B. Jungk, S. Picek, and S. Bhasin. 2018. On comparing side-channel properties of AES and chacha20 on microcontrollers. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 552–555.

193. National Institute of Standards and Technology. 2023. Post quantum cryptography FAQs: To protect against the threat of quantum computers, should we double the key length for AES now?

194. Nir, Y., and A. Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols (7539).

195. Norman, D. 1983. Design rules based on analyses of human error. *Communications of the ACM* 26: 254–258.

196. O'Kane, P., S. Sezer, and D. Carlin. 2018a. Evolution of ransomware. *IET Networks* 7(5): 321–327.

197. O'Kane, P., S. Sezer, and D. Carlin. 2018b. Evolution of ransomware. *IET Networks* 7(5): 321–327.

198. Oracle. 2013. Java applet and web start code signing.

199. Osborn, B., J. McWilliams, B. Beyer, and M. Saltonstall. 2016. Beyondcorp: Design to deployment at Google. *;login:* 41: 28–34.

200. Oz, H., A. Aris, A. Levi, and A.S. Uluagac. 2022. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Computing Surveys* 54(11s): 1–37.

201. Palmer, D. 2019. Mydoom: The 15-year-old malware that's still being used in phishing attacks in 2019.

202. Palmer, D. 2022. Android malware: A million people downloaded these malicious apps before they were finally removed from Google play.

203. Park, Y., and S.J. Stolfo. 2012. Software decoys for insider threat. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*, New York, 93–94. Association for Computing Machinery.

204. Parshall, J., A. Tully, and J.B. Lundstrom. 2005. *Shattered sword: The untold story of the Battle of Midway*. Washington, DC: Potomac Books.

205. Peterson, L.L., and B.S. Davie. 2021. *Computer Networks*, 6th ed. Morgan Kaufmann.
[zbMATH]

206. Pinto, S., and N. Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys* 51(6): 1–36.

207. Pouget, F., M. Dacier, and H. Debar. 2003. White paper: Honeypot, honeynet, honeytoken: Terminological issues. Technical Report RR-03-081, Eurecom.

208. Poulsen, K. 2003. Matrix sequel has hacker cred. *The Register*.

209. Prodhan, G. 2010. Secret coding inventors finally win recognition.

210. Provos, N. 2004 A virtual honeypot framework. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego. USENIX Association.

211. Rachlin, H., and B.A. Jones. 2008. Social discounting and delay discounting. *Journal of Behavioral Decision Making* 21(1): 29–43.

212. Ramachandran, V.S. 2007. VS Ramachandran: 3 clues to understanding your brain.

213. Ranum, M.J. 1994. Thinking about firewalls. In *Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II)*.

214. Rao, J.M., and D.H. Reiley. 2012. The economics of spam. *Journal of Economic Perspectives* 26(3): 87–110.

215. Rescorla, E. 2018. The Transport Layer Security (TLS) Protocol Version 1.3 (8446).

216. Rescorla, E., and T. Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2 (5246).

217. Rice, L. 2020. *Container Security: Fundamental Technology Concepts That Protect Containerized Applications*. Sebastopol: O’Reilly Media.

218. Ristic I., et al. 2006. Web application firewall evaluation criteria. Technical report, Web Application Security Consortium.

219. Ritchie, H., and M. Roser. 2019. Causes of death.

220. Roemer, R., E. Buchanan, H. Shacham, and S. Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* 15(1): 1–34.

221. Root, E. 2021. Spook.js, a scary bedtime story.

222. Rose, S., O. Borchert, S. Mitchell, and S. Connelly. 2020. Zero trust architecture. Special Publication (NIST SP) 800-207, National Institute of Standards and Technology, Gaithersburg.

223. Ross, R., S. Katzke, and L. Johnson. 2006. FIPS-200. Minimum security requirements for federal information and information systems. Technical report, National Institute of Standards and Technology Federal Information Processing Standards (NIST FIPS), Gaithersburg.

224. Ross, R., V. Pillitteri, K. Dempsey, M. Riddle, and G. Guissanie. 2020. Protecting controlled unclassified information in nonfederal systems and organizations. Special Publication (NIST SP) 800-171r2, National Institute of Standards and Technology, Gaithersburg.

225. Rowe, N.C., and J. Rrushi. 2016. *Introduction to Cyberdeception*, 1 ed. Springer International Publishing Switzerland.

226. Russinovich, M. 2005. Inside Sony's rootkit. *Virus Bulletin* 11–14. www.virusbulletin.com/uploads/pdf/magazine/2005/200512.pdf.

227. Sagarin, B.J., and K.D. Mitnick. 2012. The path of least resistance. In *Six Degrees of Social Influence: Science, Application, and the Psychology of Robert Cialdini*, ed. D.T. Kenrick, N.J. Goldstein, and S.L. Braver, chapter 3, 27–38. Oxford University Press.

228. Sandhu, R. 1996. Rationale for the rbac96 family of access control models. In *Proceedings of the First ACM Workshop on Role-Based Access Control (RBAC'95)*, New York, 9–17. Association for Computing Machinery.

229. Sandhu, R., D. Ferraiolo, R. Kuhn, et al. 2000. The NIST model for role-based access control: Towards a unified standard. In *ACM Workshop on Role-Based Access Control*, vol. 10.

230. Sandhu, R.S. 1998. Role-based access control. In *Advances in Computers*, vol. 46, 237–286. Elsevier.

231. Sanz, E.P., J.M. Gómez Hidalgo, and J.C. Cortizo Pérez. 2008. Email spam filtering. In *Software Development*, Advances in Computers, vol. 74, 45–114. Elsevier.

232. Sarna-Starosta, B., and S.D. Stoller. 2004. Policy analysis for security-enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, 1–12. Available at http://www.cs.sunysb.edu/~stoller/WITS2004.html.

233. Scarfone, K., and P. Mell. 2007. Guide to intrusion detection and prevention systems (IDPS). Special Publication (NIST SP) 800-94, National Institute of Standards and Technology, Gaithersburg.

234. Schaller, M., J. Park, and J. Faulkner. 2003. Prehistoric dangers and contemporary prejudices. *European Review of Social Psychology* 14(1): 105–137.

235. Schneier, B. Schneier on security.

236. Securities and Exchange Commission, U. 2023. Administrative proceeding file no. 3-21306.

237. Seitz, J., and T. Arnold. 2021. *Black Hat Python: Python Programming for Hackers and Pentesters*, 2nd ed. No Starch Press.

238. Seltzer, L. 2020. 2020 ransomware attacks still mostly through unsecured RDP.

239.

Shacham, H., M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, New York, 298–307. Association for Computing Machinery.

240. Shakevsky, A., E. Ronen, and A. Wool. 2022. Trust dies in darkness: Shedding light on Samsung's TrustZone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, 251–268. USENIX Association.

241. Shinder, T.W. 2008. *The Best Damn Firewall Book Period*, 2nd ed. Syngress.

242. Siddiqui, R. 2013. Alan turing: A note on his role as world war II cryptanalyst. *International Journal of Applied Engineering and Technology ISSN: 2277-212X (Online)* 3: 21–26.

243. Sikorski, M., and A. Honig. 2012. *Practical Malware Analysis*. San Francisco: No Starch Press.

244. Sonnemaker, T. 2021. Verkada allowed at least 100 employees, including interns and sales staff, to access customers' camera feeds. *Business Insider*.

245. Souppaya, M., and K. Scarfone. 2022. Guide to enterprise patch management planning. Special Publication (NIST SP) 800-40r4, National Institute of Standards and Technology, Gaithersburg.

246. Souppaya, M., K. Scarfone, and D. Dodson. 2022. Secure software development framework (SSDF) version 1.1. Special Publication (NIST SP) 800-218, National Institute of Standards and Technology, Gaithersburg.

247. Spafford, E.H. 1989. The internet worm program: An analysis. *SIGCOMM Computer Communications Review* 19(1): 17–57.

248. Spitzner, L. 2003. Honeypots: Catching the insider threat. In *19th Annual Computer Security Applications Conference, 2003*, 170–179. IEEE.

249. Spitzner, L. 2003. Honeytokens: The other honeypot.

250. Stallings, W. 2013. *Cryptography and Network Security: Principles and Practice*, 6th ed. Prentice Hall Press.

251. Stallings, W. 2016. ICAM: A foundation for trusted identities in cyberspace. *IT Professional* 18(1): 26–33.

252. Stewart, J.M., and D. Kinsey. 2020. *Network Security, Firewalls, and*

VPNs, 3rd ed. Jones & Bartlett Learning.

253. Stoll, C. 1989. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. New York: Doubleday.

254. Stueh. 2023. Comment on: [3cx breach update].

255. Stuttard, D., and M. Pinto. 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security*, 2nd ed. Wiley.

256. Suljkanovic, S. 2005. Honeypots or honey delusions. Technical report, SANS Institute.

257. Swanson, M., J. Hash, and P. Bowen. 2006. Guide for developing security plans for federal information systems. Special Publication (NIST SP) 800-18r1, National Institute of Standards and Technology, Gaithersburg.

258. Symonds, C. 2011. *The Battle of Midway*, Pivotal Moments in American History. Oxford University Press.

259. Szappanos, G. 2004. Virus analysis 2: We're all doomed. *Virus Bulletin* 9–13. www.virusbulletin.com/uploads/pdf/magazine/2004/200403.pdf.

260. Thakur, M.A., and R. Gaikwad. 2015. User identity and access management trends in it infrastructure—An overview. In *2015 International Conference on Pervasive Computing (ICPC)*, 1–4.

261. The PC Security Channel. 2023. 3cx: How this malware almost hacked every business.

262. Thorkildssen, H.W. 2004. Spam-different approaches to fighting unsolicited commercial email a survey of spam and spam countermeasures. *Network and System Administration Research Surveys* 1: 45–55.

263. Toulas, B. 2022. New Intel chips won't play blu-ray disks due to SGX deprecation.

264. Toulas, B. 2023. 75k wordpress sites impacted by critical online course plugin flaws. *Bleeping Computer*. www.bleepingcomputer.com/news/security/75k-wordpress-sites-impacted-by-critical-online-course-plugin-flaws/.

265. Tracy, M., W. Jansen, K. Scarfone, and T. Winograd. 2007. Guidelines on securing public web servers. Special Publication (NIST SP) 800-44r2, National Institute of Standards and Technology, Gaithersburg.

266. Tunggal, A.T. 2022. What is role-based access control (RBAC)? Examples, benefits, and more.

267. Tuohy, W. 2007. *America’s Fighting Admirals*. Zenith Press.

268. Tzu, S. 2002. *Sun Tzu: Art of War*. Trans. Ralph D. Sawyer. Basic Books.

269. Ur, B., F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L.F. Cranor, H. Dixon, P. Emami Naeini, H. Habib, N. Johnson, and W. Melicher. 2017. Design and evaluation of a data-driven password meter. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI’17)*, New York, 3775–3786. Association for Computing Machinery.

270. Vaas, L. 2022. Samsung shattered encryption on 100m phones.

271. van der Dennen, J., K. Thienpont, and R.L. Cliquet. 2000. Of badges, bonds and boundaries: Ingroup/outgroup differentiation and ethnocentrism revisited. In *In-group/out-group behaviour in modern societies: An evolutionary perspective*, NIDI/CBGS Publications, ed. Robert Cliquet and Kristiaan Thienpont, 37–74. Nederlands Interdisciplinair Demografisch Instituut (NIDI); Centrum voor Bevolkings- en Gezinsstudien (CBGS).

272. Vehent, J. 2018. *Securing DevOps: Security in the Cloud*. Shelter Island/New York: Manning Publications Co.

273. VeraCrypt. VeraCrypt documentation: Authenticity and integrity.

274. VeraCrypt. VeraCrypt documentation: Header key derivation, salt, and iteration count.

275. Verizon. 2017. Verizon 2017 data breach investigations report.

276. Vijayan, J. 2023. 3cx breach widens as cyberattackers drop second-stage backdoor.

277. Vinck, A.J.H. 2012. Introduction to public key cryptography. Accessed 08 Oct 2018.

278. Vopson, M.M. 2021. The world’s data explained: how much we’re producing and where it’s all stored.

279. Ward, R., and B. Beyer. 2014. Beyondcorp: A new approach to enterprise security. *;login:* 39(6): 6–11.

280. Whitney, L. 2021. Microsoft power apps misconfiguration exposes data from 38 million records.

281. Wilcox, J. 2015. *Solving the ENIGMA: History of the Cryptanalytic Bombe*. National Security Agency Center for Cryptologic History.

282. Williams, C. 2016. Double ko! Capcom’s street fighter v installs hidden

rootkit on PCs. *The Register*.

283. Wilson, Y., and A. Hingnikar. 2022. *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2*, 2nd ed. Apress.

284. Wiseman, B. 2017. Page not found: A brief history of the 404 error.

285. Woods, D.W. 2023. A turning point for cyber insurance. *Communications of the ACM* 66(3): 41–44.

286. Yang, S. 2022. As China shuts out the world, internet access from abroad gets harder too. *LA Times*.

287. Yaworski, P. 2019. *Real-World Bug Hunting: A Field Guide to Web Hacking*. No Starch Press.

288. Yiu, J. 2017. Software development in ARMv8-M architecture. Presented at Embedded World 2017.

289. Yurtseven, İ, S. Bagriyanik, and S. Ayvaz. 2021. A review of spam detection in social media. In *2021 6th International Conference on Computer Science and Engineering (UBMK)*, 383–388. IEEE.

290. Yuryna Connolly, L., D.S. Wall, M. Lang, and B. Oddson. 2020. An empirical study of ransomware attacks on organizations: An assessment of severity and salient factors affecting vulnerability. *Journal of Cybersecurity* 6(1): tyaa023.

Index

Footnotes

1 This is changing however. Some HTTP traffic is now carried over the QUIC protocol which is layered on top of UDP.

2 Technically, layer 1 is still present and some applications do make the distinction. But may components just combine layer 2 and layer 1 into a single element.