A fast port scanner in 100 lines of Rust

Wed, Aug 11, 2021

To write a fast port scanner, a programming language requires:

- **A Good I/O model**, not to eat all the resources of the system.
- **High-level abstractions and a good packaging system** to isolate low-level code and reuse it easily.
- **To be type and memory safe**, because who wants offensive tools with vulnerabilities?
- **And, ideally, to be compiled**, because most of the time, it's worth trading a little bit of compile time for extreme runtime speed.

Guess what? These are precisely Rust's selling points. So let see how to build a high-speed port scanner in 100 lines of Rust.

A port scanner is basically composed of 3 parts:

- **A list of ports to scan**
- **A port scanning algorithm** (see this list on nmap's website)
- **A concurrency primitive**, to scan port concurrently

A Generic port list

Scanning all the *65535* ports is often wasteful and useless. Thus, we first need to extract the list of the most common open ports in the wild. Fortunately, the nmap project already has such a list:

```
// from awk '$2~/tcp$/' /usr/share/nmap/nmap-services | sort -r -k3 | head -n
pub const MOST_COMMON_PORTS_1002: &[u16] = &[
    5601, 9300, 80, 23, 443, 21, 22, 25, 3389, 110, 445, 139, 143, 53, 135, 3
    995, 993, 5900, 1025, 587, 8888, 199, 1720, 465, 548, 113, 81, 6001, 1000
    1026, 2000, 8443, 8000, 32768, 554, 26, 1433, 49152, 2001, 515, 8008, 491
    5000, 5631, 631, 49153, 8081, 2049, 88, 79, 5800, 106, 2121, 1110, 49155,
    427, 49156, 543, 544, 5101, 144, 7, 389, 8009, 3128, 444, 9999, 5009, 707
    1900, 3986, 13, 1029, 9, 5051, 6646, 49157, 1028, 873, 1755, 2717, 4899,
    3001, 5001, 82, 10010, 1030, 9090, 2107, 1024, 2103, 6004, 1801, 5050, 19
    // ...
];
```

Then, we need to be able to return either this list or all the 65535 ports. In Rust, the way to achieve this is with an Iterator.

But, as Iterator is a trait, we need to use a Trait Object to be able to return an Iterator of 2 different types.

```
fn get_ports(full: bool) -> Box<dyn Iterator<Item = u16>> {
    if full {
```

```
        Box::new((1..=u16::MAX).into_iter())
    } else {
        Box::new(ports::MOST_COMMON_PORTS_1002.to_owned().into_iter())
    }
}
```

Scanning a single port

To scan a port, we will use the **TCP connect** technique, as it's the easiest one to implement and requires no special privilege or raw socket. The 20% which brings us 80% of the results.

```
async fn scan_port(target: IpAddr, port: u16, timeout: u64) {
    let timeout = Duration::from_secs(timeout);
    let socket_address = SocketAddr::new(target.clone(), port);

    if tokio::time::timeout(timeout, TcpStream::connect(&socket_address))
        .await
        .is_ok()
    {
        println!("{}", port);
    }
}
```

Extreme concurrency

If you are a recurrent reader of this blog you should have guessed the perfect concurrency primitive for the task (if not, you can subscribe here): A Stream :)

```
async fn scan(target: IpAddr, full: bool, concurrency: usize, timeout: u64) {
    let ports = stream::iter(get_ports(full));

    ports
        .for_each_concurrent(concurrency, |port| scan_port(target, port, time
        .await;
}
```

Parsing CLI arguments

And finally, we need to parse the CLI arguments and all the configuration boilerplate to run our scanner:

```
use clap::{App, Arg};
use futures::{stream, StreamExt};
use std::{
    net::{IpAddr, SocketAddr, ToSocketAddrs},
    time::Duration,
};
use tokio::net::TcpStream;

mod ports;

#[tokio::main]
```

```rust
async fn main() -> Result<(), anyhow::Error> {
    let cli_matches = App::new(clap::crate_name!())
        .version(clap::crate_version!())
        .about(clap::crate_description!())
        .arg(
            Arg::with_name("target")
                .help("The target to scan")
                .required(true)
                .index(1),
        )
        .arg(
            Arg::with_name("concurrency")
                .help("Concurrency")
                .long("concurrency")
                .short("c")
                .default_value("1002"),
        )
        .arg(
            Arg::with_name("verbose")
                .help("Display detailed information")
                .long("verbose")
                .short("v"),
        )
        .arg(
            Arg::with_name("full")
                .help("Scan all 65535 ports")
                .long("full"),
        )
        .arg(
            Arg::with_name("timeout")
                .help("Connection timeout")
                .long("timeout")
                .short("t")
                .default_value("3"),
        )
        .setting(clap::AppSettings::ArgRequiredElseHelp)
        .setting(clap::AppSettings::VersionlessSubcommands)
        .get_matches();

    let full = cli_matches.is_present("full");
    let verbose = cli_matches.is_present("verbose");
    let concurrency = cli_matches
        .value_of("concurrency")
        .unwrap()
        .parse::<usize>()
        .unwrap_or(1002);
    let timeout = cli_matches
        .value_of("timeout")
        .unwrap()
        .parse::<u64>()
        .unwrap_or(3);
    let target = cli_matches.value_of("target").unwrap();

    if verbose {
        let ports = if full {
            String::from("all the 65535 ports")
        } else {
```

```
            String::from("the most common 1002 ports")
        };
        println!(
            "Scanning {} of {}. Concurrency: {:?}. Timeout: {:?}",
            &ports, target, concurrency, timeout
        );
    }

    let socket_addresses: Vec<SocketAddr> = format!("{}:0", target).to_socket

    if socket_addresses.is_empty() {
        return Err(anyhow::anyhow!("Socket_addresses list is empty"));
    }

    scan(socket_addresses[0].ip(), full, concurrency, timeout).await;

    Ok(())
}
```

```
$ cargo run --release -- kerkour.com
80
8080
8443
443
```

Conclusion

Even if Rust is a strongly typed and compiled language, we just built a complex program as easily as if it's were in a scripting language, but way faster and way safer.

The next steps? Implementing more port scanning strategies.

The code is on GitHub

As usual, you can find the code on GitHub: github.com/skerkour/kerkour.com (please don't forget to star the repo 🐿)

Want to learn Rust and offensive security? Take a look at my book **Black Hat Rust**. All early-access supporters get a special discount and awesome bonuses: https://academy.kerkour.com/black-hat-rust?coupon=BLOG.
Warning: this offer is limited in time!

Tags: rust, programming, tutorial, hacking, security

Related posts

- Hello Simplon
- How to execute shellcodes from memory in Rust
- HTTP Security Headers: Why? How? What?
- Black Hat Rust: July Update
- 15k inserts/s with Rust and SQLite