HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

BÁO CÁO PROJECT 2

Chủ đề: Ngôn ngữ Rust và Bảo mật

Giảng viên hướng dẫn: Nguyễn Đức Toàn

Sinh viên: Phạm Đặng Tấn Dũng

Mã số sinh viên: 20225569

Ngày 1 tháng 4 năm 2025

Mục lục

Chương 1

Week 1

1.1 Rust Basics

1.1.1 Variables in Rust

Biến trong Rust là bất biến (immutable) theo mặc định, nghĩa là không thể thay đổi sau khi được gán giá trị. Để khai báo biến có thể thay đổi, ta dùng từ khóa `mut`.

```
let x = 5;  // immutable
let mut y = 10;  // mutable
y = 15;  // ok
```

1.1.2 Constants in Rust

Hằng số trong Rust được khai báo bằng từ khóa `const` và phải có kiểu dữ liệu rõ ràng. Giá trị của hằng số không thể thay đổi.

```
const MAX_POINTS: u32 = 100_000;
```

1.1.3 Shadowing in Rust

Shadowing cho phép khai báo lại một biến với cùng tên, che giấu biến cũ.

```
let x = 5;
let x = x + 1;  // x is 6
```

1.1.4 Data Types in Rust

Rust có các kiểu dữ liệu cơ bản như số nguyên (`i32`, `u32`), số thực (`f32`, `f64`), boolean (`bool`), và ký tự (`char`). Ngoài ra còn có tuple và array.

```
let integer: i32 = 42;
let float: f64 = 3.14;
let boolean: bool = true;
let character: char = 'a';
let tuple: (i32, f64, char) = (500, 6.4, 'b');
let array: [i32; 3] = [1, 2, 3];
```

1.1.5 Console Input in Rust

Để nhận input từ console, ta sử dụng `std::io::stdin`.

```
use std::io;
let mut input = String::new();
io::stdin().read_line(&mut input).expect("Failed to read line"
    );
```

1.1.6 Arithmetic and Type Casting in Rust

Rust hỗ trợ các phép toán số học cơ bản. Type casting được thực hiện bằng từ khóa `as`.

```
let sum = 5 + 10;
let product = 4 * 30;
let quotient = 56.7 / 32.2;
let remainder = 43 % 5;
let casted = 5 as f64;
```

1.1.7 Condition If Else in Rust

Câu lệnh điều kiện trong Rust tương tự như các ngôn ngữ khác.

```
let number = 3;
if number < 5 {
    println!("condition was true");
} else {
    println!("condition was false");
}
```

1.1.8 Functions in Rust

Hàm trong Rust được khai báo bằng từ khóa `fn`. Hàm có thể trả về giá trị.

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

1.1.9 Expressions and Statements in Rust

Expressions trả về giá trị, trong khi statements không. Trong Rust, khối mã có thể là expression.

```
let y = {
    let x = 3;
    x + 1  // expression, no semi-colon
};
```

1.1.10 Heap and Stack in Rust

Stack lưu trữ dữ liệu có kích thước cố định, trong khi heap lưu trữ dữ liệu có kích thước động. Rust quản lý bộ nhớ heap thông qua ownership.

```
// Stack
let s = 5;  // i32, default

// Heap
let s = String::from("hello");  // String, dynamic size
```

1.2 Types of Attacks

Cyberattacks là các hành vi tấn công vào hệ thống máy tính hoặc mạng nhằm gây hại, đánh cắp thông tin, hoặc làm gián đoạn dịch vụ. Dưới đây là các loại tấn công phổ biến:

- **Tấn công không có mục tiêu rõ ràng**: Thường do thanh thiếu niên tò mò thực hiện, không có mục đích cụ thể nhưng vẫn gây thiệt hại tài chính.
- **Tấn công chính trị**: Nhằm truyền tải thông điệp chính trị, ví dụ: thay đổi nội dung website (defacement) hoặc tấn công từ chối dịch vụ (DoS).

- **Pentest (Penetration Testing)**: Kiểm tra bảo mật hệ thống bằng cách mô phỏng tấn công, đôi khi chỉ để đáp ứng yêu cầu tuân thủ mà không thực sự hiệu quả.
- **Red Team**: Mô phỏng tấn công thực tế với phạm vi rộng hơn (như phishing, xâm nhập vật lý) để đánh giá khả năng phòng thủ.
- **Bug Bounty**: Chương trình thưởng cho việc phát hiện và báo cáo lỗ hổng bảo mật, tuy nhiên đôi khi chỉ mang tính hình thức.
- **Tội phạm mạng (Cybercrime)**: Bao gồm đánh cắp dữ liệu, ransomware, gian lận thẻ tín dụng; nổi bật với vụ rò rỉ công cụ NSA năm 2017.
- **Gián điệp công nghiệp**: Đánh cắp bí mật thương mại để giành lợi thế cạnh tranh.
- **Chiến tranh mạng (Cyberwar)**: Tấn công mạng trong chiến tranh, ví dụ: worm Stuxnet – vũ khí số đầu tiên trên thế giới.

1.2.1 Phases of an Attack

Một cuộc tấn công mạng thường trải qua các giai đoạn sau:

- **Reconnaissance (Thu thập thông tin)**: Tìm hiểu về mục tiêu qua dữ liệu công khai (passive) hoặc quét mạng trực tiếp (active).
- **Exploitation (Khai thác)**: Xâm nhập ban đầu bằng cách sử dụng lỗ hổng (exploits) hoặc kỹ thuật xã hội (social engineering).
- **Lateral Movements (Di chuyển bên trong)**: Duy trì truy cập và mở rộng quyền hạn trong hệ thống, thường bằng các công cụ như RATs (Remote Access Tools).
- **Data Exfiltration (Trích xuất dữ liệu)**: Đánh cắp dữ liệu từ hệ thống, cần thực hiện cẩn thận để tránh bị phát hiện.
- **Clean Up (Dọn dẹp)**: Xóa dấu vết (logs, file tạm, cơ sở hạ tầng) để tránh bị truy ra.

1.2.2 Profiles of Attackers

Những kẻ tấn công có thể thuộc nhiều nhóm với vai trò khác nhau trong một đội tấn công:

- **The Hacker**: Người có kỹ năng tấn công cao, chịu trách nhiệm thu thập thông tin và khai thác lỗ hổng.

- **The Exploit Writer**: Phát triển công cụ và exploits để xâm nhập hệ thống.
- **The Developer**: Xây dựng các công cụ tùy chỉnh (credential dumpers, proxies, implants) để tránh bị phát hiện.
- **The System Administrator**: Quản lý và bảo mật cơ sở hạ tầng của kẻ tấn công, hỗ trợ trong các giai đoạn khai thác và di chuyển.
- **The Analyst**: Cung cấp kiến thức chuyên môn để phân tích dữ liệu và ưu tiên mục tiêu.

1.2.3 Rust Error Handling

Trong Rust, error handling được thực hiện một cách rõ ràng và an toàn, thường sử dụng kiểu dữ liệu `Result<T, E>` để biểu diễn kết quả có thể thành công hoặc thất bại. Ví dụ:

```
fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Code maybe return error
    Ok(())
}
```

Trong đó, `Box<dyn std::error::Error>` cho phép trả về bất kỳ loại lỗi nào implement trait `std::error::Error`.

1.2.4 Reading Files in Rust

Đọc file trong Rust có thể được thực hiện bằng cách sử dụng `std::fs::File` và `std::io::BufReader`. Ví dụ:

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

let file = File::open("filename.txt")?;
let reader = BufReader::new(file);
for line in reader.lines() {
    println!("{}", line?);
}
```

1.2.5 Lifetime Annotation

Lifetime annotation trong Rust được sử dụng để chỉ định phạm vi mà các tham chiếu hợp lệ, đảm bảo rằng các tham chiếu không tồn tại lâu hơn dữ liệu mà chúng trỏ đến. Ví dụ:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Trong ví dụ này, `'a` là lifetime annotation chung cho cả hai tham chiếu đầu vào và đầu ra.

Trong Rust, lifetime annotation được sử dụng để chỉ định phạm vi mà các tham chiếu hợp lệ, đảm bảo rằng các tham chiếu không tồn tại lâu hơn dữ liệu mà chúng trỏ đến, từ đó ngăn chặn các tham chiếu lơ lửng. Tuy nhiên, việc sử dụng quá nhiều lifetime annotation có thể làm cho mã nguồn trở nên phức tạp và khó đọc hơn, đặc biệt đối với người mới học. Hơn nữa, trong nhiều trường hợp, trình biên dịch có thể tự động suy ra lifetime, do đó không cần thiết phải ghi chú rõ ràng. Vì vậy, nên hạn chế sử dụng lifetime annotation và chỉ dùng khi thực sự cần thiết, chẳng hạn trong các tình huống phức tạp mà trình biên dịch không thể suy ra đúng lifetime.

1.2.6 Rc (Reference Counting)

`Rc` (Reference Counting) là một smart pointer trong Rust cho phép chia sẻ quyền sở hữu của dữ liệu giữa nhiều phần của chương trình. Nó đếm số lượng tham chiếu đến dữ liệu và tự động giải phóng khi không còn tham chiếu nào. Ví dụ mã giả:

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

let data = Rc::new(5);
let clone1 = Rc::clone(&data);
let clone2 = Rc::clone(&data);
println!("{}", *clone1);  // 5
```

1.2.7 Arc (Atomic Reference Counting)

`Arc` (Atomic Reference Counting) tương tự như `Rc` nhưng an toàn cho việc sử dụng trong môi trường đa luồng. Nó sử dụng atomic operations để đếm số lượng tham chiếu. Ví dụ mã giả:

```
use std::sync::Arc;
```

```rust
use std::thread;

let data = Arc::new(5);
let clone1 = Arc::clone(&data);
thread::spawn(move || {
    println!("{}", *clone1);  // 5
});
```

Chương 2

Week 2

2.1 Multi-threading port scanning program

2.1.1 Giới thiệu

Trong tuần 2, em sẽ tập trung vào khả năng đa luồng của Rust và viết chương trình quét cổng đa luồng. Một chương trình quét cổng tương tự đã được viết và so sánh để làm nổi bật tốc độ khi sử dụng đa luồng trong Rust.

2.1.2 Dependencies

```
[dependencies]
thiserror = "1.0"
anyhow = "1.0"
rayon = "1.5"
trust-dns-resolver = "0.21"
reqwest = { version = "0.11", default-features = false,
   features = ["json", "blocking", "rustls-tls"] }
serde = { version = "1", features = ["derive"] }

```

Listing 2.1: Dependencies

2.2 Main execution thread (`main.rs`)

1. **Lấy đối số dòng lệnh:**

```rust
let args: Vec<String> = env::args().collect();

if args.len() != 2 {
    return Err(Error::CliUsage.into());
}

let target = args[1].as_str();

```

Listing 2.2: Lấy đối số dòng lệnh (main.rs)

- `env::args().collect()`: Thu thập các đối số dòng lệnh vào một vector `args`.
- `if args.len() != 2`: Kiểm tra số lượng đối số.
- `let target = args[1].as_str();`: Lấy tên miền mục tiêu.

2. **Tạo HTTP client:**

```rust
let http_timeout = Duration::from_secs(5);
let http_client = Client::builder()
    .redirect(redirect::Policy::limited(4))
    .timeout(http_timeout)
    .build()?;

```

Listing 2.3: Tạo HTTP client (main.rs)

- `Duration::from_secs(5)`: Tạo `Duration` 5 giây.
- `Client::builder()`: Bắt đầu xây dựng `reqwest::blocking::Client`.
- `.redirect(redirect::Policy::limited(4))`: Cấu hình chuyển hướng tối đa 4 lần(Có thể do trang web đã được di chuyển, máy chủ muốn sử dụng phiên bản HTTPS thay vì HTTP, cần đăng nhập trước khi xem nội dung, URL ban đầu là một URL rút gọn..).
- `.timeout(http_timeout)`: Đặt thời gian chờ 5 giây.
- `.build()?`: Hoàn tất và trả về `Result`.

3. **Tạo thread pool (rayon):**

```rust
let pool = rayon::ThreadPoolBuilder::new()
    .num_threads(256)
    .build()
    .unwrap();
```

```
5
```

Listing 2.4: Tạo thread pool (main.rs)

- `rayon::ThreadPoolBuilder::new()`: Bắt đầu xây dựng `rayon::ThreadPool`.
- `.num_threads(256)`: Cấu hình 256 luồng.
- `.build().unwrap()`: Hoàn tất và trả về `ThreadPool`.

4. **Thực hiện quét (trong thread pool):**

```
pool.install(|| {
    let scan_result: Vec<Subdomain> = subdomains::
   enumerate(&http_client, target)
        .unwrap()
        .into_par_iter()
        .map(ports::scan_ports)
        .collect();
});

```

Listing 2.5: Thực hiện quét (main.rs)

- `pool.install(|| ... );`: Đăng ký closure để thực thi trong thread pool.
- `subdomains::enumerate(&http_client, target).unwrap()`: Lấy danh sách subdomain.
- `.into_par_iter()`: Sử dụng parallel iterator.
- `.map(ports::scan_ports)`: Áp dụng `scan_ports` cho từng subdomain (song song).
- `.collect()`: Thu thập kết quả vào `Vec<Subdomain>`.

5. **In kết quả:**

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

Listing 2.6: In kết quả (main.rs)

2.3 Details

2.3.1 `subdomains::enumerate(subdomains.rs)`

1. **Truy vấn `crt.sh`:**

```
let entries: Vec<CrtShEntry> = http_client
    .get(&format!("https://crt.sh/?q=%25.{}&output=json", target))
    .send()?
    .json()?;

```

Listing 2.7: Truy vấn crt.sh (subdomains.rs)

- `http_client.get(...)`: Tạo yêu cầu GET đến `crt.sh`.
- `.send()?`: Gửi yêu cầu.
- `.json()?`: Phân tích phản hồi JSON thành `Vec<CrtShEntry>`.

2. **Xử lý và lọc dữ liệu:**

```
    let mut subdomains: HashSet<String> = entries
            .into_iter()
            .flat_map(|entry| {
                entry
                    .name_value
                    .split('\n')
                    .map(|subdomain| subdomain.trim().to_string())
                    .collect::<Vec<String>>()
            })
            .filter(|subdomain: &String| subdomain != target)
            .filter(|subdomain: &String| !subdomain.contains('*'))
            .collect();
        subdomains.insert(target.to_string());
```

Listing 2.8: xử lí và lọc (subdomains.rs)

- `entries.into_iter()`: Chuyển vector `entries` thành iterator.
- `.flat_map(...)`: Biến đổi mỗi `CrtShEntry` thành một iterator.

- `entry.name_value.split('\n')`: Tách `name_value`.
- `.map(|subdomain| subdomain.trim().to_string())`: Loại bỏ khoảng trắng, chuyển String
- `collect()`: Đưa các subdomain vào vector
- `.filter(|subdomain: &String| subdomain != target)`: Loại bỏ tên miền.
- `.filter(|subdomain: &String| !subdomain.contains('*'))`: Loại bỏ subdomain chứa `*`.
- `.collect()`: Thu thập vào `HashSet<String>`.

3. **Tạo `Vec<Subdomain>` và kiểm tra DNS:**

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

```

Listing 2.9: Tạo Vec<Subdomain> và kiểm tra DNS(subdomains.rs)

2.3.2 `subdomains::resolves` (`subdomains.rs`)

1. **Cấu hình và tạo DNS resolver:**

```
        let mut opts = ResolverOpts::default();
        opts.timeout = Duration::from_secs(4);
        let dns_resolver = Resolver::new(ResolverConfig::
    default(), opts).expect("resolver");

```

Listing 2.10: Cấu hình và tạo(subdomains.rs)

2. **Thực hiện tra cứu DNS:**

```
    dns_resolver.lookup_ip(domain.domain.as_str()).is_ok()

```

Listing 2.11: Thực hiện tra cứu(subdomains.rs)

2.3.3 `ports::scan_ports` (ports.rs)

1. **Tạo** `SocketAddr`**:**

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

if socket_addresses.is_empty() {
    return subdomain;
}

```

Listing 2.12: Tạo SocketAddr(ports.rs)

2. **Quét các cổng (song song):**

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

```

Listing 2.13: quét cổng(ports.rs)

2.3.4 `ports::scan_port` (ports.rs)

1. **Đặt cổng và thử kết nối:**

```
let timeout = Duration::from_secs(3);
socket_address.set_port(port);

let is_open = TcpStream::connect_timeout(&socket_address,
   timeout).is_ok();

```

Listing 2.14: Đặt cổng(port.rs)

2. **Trả về** `Port`**:**

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

```
2
```

Listing 2.15: trả về(port.rs)

2.4 Conclusion

Qua thực nghiệm, thời gian thực hiện việc quét một địa chỉ khi sử dụng đa luồng khá nhanh(khoảng 3-5s) trong khi nếu thực thi trên một luồng duy nhất, thời gian khoảng 10 phút(tất nhiên còn phụ thuộc vào việc đặt thời gian chờ đợi phản hồi). Tuy nhiên, việc lập trình đa luồng giúp giảm đáng kể tốc độ thời gian thực thi chương trình.

Tuy nhiên việc mỗi luồng vẫn phải chờ đợi thời gian trả về của request khiến chương trình chưa đạt tốc độ thực thi tối đa. Tuần 3 sẽ nghiên cứu về áp dụng async/await(lập trình bất đồng bộ) cải thiện tốc độ của chương trình.

Chương 3

Week 3

3.1 Introduction

Để tăng tốc việc quét các cổng và trả về, việc kết hợp multi-threading với async/await sử dụng thư viện `rayon` và Async/Await sử dụng `tokio` và `futures` sẽ được áp dụng. Tiếp theo là so sánh giữa multi threading(Rayon) và async/await(Tokio). Cuối cùng là đánh giá sự khác biệt về mặt code và hiệu năng khi áp dụng Async/Await để quét cổng và subdomain.

Tuần sau em sẽ tìm hiểu về trait, modular hóa chương trình scanner và sử dụng trait objects để tạo ra các modules quét khác nhau.

3.2 Details

3.2.1 Nhắc lại về chức năng

Cả hai phiên bản đều thực hiện hai chức năng chính:

1. **Liệt kê Subdomain**: Sử dụng `crt.sh` để tìm kiếm các subdomain liên quan đến tên miền mục tiêu.

2. **Quét Cổng**: Quét các cổng phổ biến trên mỗi subdomain để xác định cổng nào đang mở.

Sự khác biệt chính nằm ở cách chúng thực hiện đồng thời hai chức năng này, cụ thể là cách Rayon và Tokio xử lí các yêu cầu.

3.2.2 Thay đổi trong `main.rs`

Phiên bản Multi-threading

```rust
use rayon::prelude::*;
use reqwest::{blocking::Client, redirect};
use std::{env, time::Duration};

mod error;
pub use error::Error;
mod model;
mod ports;
mod subdomains;
use model::Subdomain;
mod common_ports;

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

    if args.len() != 2 {
        return Err(Error::CliUsage.into());
    }

    let target = args[1].as_str();

    let http_timeout = Duration::from_secs(5);
    let http_client = Client::builder()
        .redirect(redirect::Policy::limited(4))
        .timeout(http_timeout)
        .build()?;

    //custom threadpool with 256 threads
    let pool = rayon::ThreadPoolBuilder::new()
        .num_threads(256)
        .build()
        .unwrap();

    // pool.install
    pool.install(|| {
        let scan_result: Vec<Subdomain> = subdomains::
   enumerate(&http_client, target)// l y  danh s ch subdomain
     c a  t n  m i n
```

```
            .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!();
        }
    });

    Ok(())
}
```

Listing 3.1: main.rs (Multi-threading)

Điểm nổi bật:

- Sử dụng `rayon::prelude::*` để kích hoạt parallel iterators.
- Sử dụng `reqwest::blocking::Client` cho các request HTTP đồng bộ.
- Tạo thread pool tùy chỉnh với `rayon::ThreadPoolBuilder`.
- Sử dụng `into_par_iter()` và `map()` để thực hiện quét subdomain và port song song.

Phiên bản Async/Await

```
use futures::{stream, StreamExt};
use reqwest::Client;
use std::{
    env,
    time::{Duration, Instant},
};

mod error;
pub use error::Error;
mod model;
```

```rust
mod ports;
mod subdomains;
use model::Subdomain;
mod common_ports;

#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let args: Vec<String> = env::args().collect();

    if args.len() != 2 {
        return Err(Error::CliUsage.into());
    }

    let target = args[1].as_str();

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

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

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

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


    let scan_duration = scan_start.elapsed();
    println!("Scan completed in {:?}", scan_duration);

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

```
48          for port in &subdomain.open_ports {
49              println!("    {}: open", port.port);
50          }
51
52          println!("");
53      }
54
55      Ok(())
56  }
```

Listing 3.2: main.rs (Async/Await)

Điểm nổi bật:

- Sử dụng `futures::stream` và `StreamExt` cho stream concurrency.
- Sử dụng `reqwest::Client` cho các request HTTP không đồng bộ.
- Sử dụng macro `[tokio::main]` để đánh dấu hàm `main` là async và runtime của tokio.
- Sử dụng `buffer_unordered()` để giới hạn mức độ đồng thời khi quét port cho các subdomain.
- Sử dụng `.await?` để chờ kết quả của các future và xử lý lỗi.

Thay đổi chính: Thay vì sử dụng thread pool và parallel iterator của `rayon`, phiên bản Async/Await sử dụng `tokio` runtime và các stream của `futures` để quản lý concurrency. Các thao tác I/O như HTTP request và TCP connection trở thành không đồng bộ, giúp chương trình không bị block khi chờ đợi các thao tác này hoàn thành.

3.2.3 Thay đổi trong `ports.rs`

File `ports.rs` chịu trách nhiệm quét các cổng trên một subdomain.

Phiên bản Multi-threading

```
1  use crate::{
2      common_ports::MOST_COMMON_PORTS_100,
3      model::{Port, Subdomain},
4  };
5  use rayon::prelude::*;
```

```rust
use std::net::{SocketAddr, ToSocketAddrs};
use std::{net::TcpStream, time::Duration};

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

    if socket_addresses.is_empty() {
        return subdomain;
    }

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

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

    let is_open = TcpStream::connect_timeout(&socket_address, timeout).is_ok();

    Port { port, is_open }
}
```

Listing 3.3: ports.rs (Multi-threading)

Điểm nổi bật:

- Sử dụng `rayon::prelude::*` để quét cổng song song.
- Sử dụng `std::net::TcpStream::connect_timeout()` cho TCP connection (blocking).

Phiên bản Async/Await

```rust
use crate::{
    common_ports::MOST_COMMON_PORTS_100,
    model::{Port, Subdomain},
};
use futures::StreamExt;
use std::net::{SocketAddr, ToSocketAddrs};
use std::time::Duration;
use tokio::net::TcpStream;
use tokio::sync::mpsc;

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

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

    let socket_address = socket_addresses[0];

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

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

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

```
    await;
40                  if port.is_open {
41                      let _ = output_tx.send(port).await;
42                  }
43              }
44          })
45          .await;
46      // close channel
47      drop(output_tx);
48
49      let output_rx_stream = tokio_stream::wrappers::
    ReceiverStream::new(output_rx);
50      ret.open_ports = output_rx_stream.collect().await;
51
52      ret
53  }
54
55  async fn scan_port(mut socket_address: SocketAddr, port: u16)
    -> Port {
56      let timeout = Duration::from_secs(3);
57      socket_address.set_port(port);
58
59      let is_open = matches!(
60          tokio::time::timeout(timeout, TcpStream::connect(&
    socket_address)).await,
61          Ok(Ok(_)),
62      );
63
64      Port {
65          port: port,
66          is_open,
67      }
68  }
```

Listing 3.4: ports.rs (Async/Await)

Điểm nổi bật:

- Sử dụng `tokio::net::TcpStream::connect()` cho TCP connection (non-blocking).
- Sử dụng `tokio::time::timeout()` để giới hạn thời gian chờ connection.
- Sử dụng `tokio::sync::mpsc::channel` và `futures::StreamExt::for_each_concurrent()` để quản lý concurrency khi quét port.

- Hàm `scan_ports` và `scan_port` được đánh dấu là `async fn` và sử dụng `.await` để chờ các future.

Thay đổi chính: Phiên bản Async/Await chuyển sang sử dụng các API không đồng bộ của `tokio` cho TCP connection và timeout. Thay vì parallel iterator, nó sử dụng channels và stream concurrency để kiểm soát số lượng task quét port đồng thời. Điều này giúp tối ưu hóa hiệu suất trong môi trường I/O-bound.

3.2.4 Thay đổi trong `subdomains.rs`

File `subdomains.rs` thực hiện việc liệt kê subdomain từ `crt.sh` và kiểm tra DNS resolution.

Phiên bản Multi-threading

```
use crate::{
    model::{CrtShEntry, Subdomain},
    Error,
};
use reqwest::blocking::Client;
use std::{collections::HashSet, time::Duration};
use trust_dns_resolver::{
    config::{ResolverConfig, ResolverOpts},
    Resolver,
};

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()
        .flat_map(|entry| {
            entry
                .name_value
                .split('\n')
```

```
                .map(|subdomain| subdomain.trim().to_string())
                .collect::<Vec<String>>()
        })
        .filter(|subdomain: &String| subdomain != target)
        .filter(|subdomain: &String| !subdomain.contains('*'))
    // filter subdomain having wildcards
        .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)
}

pub fn resolves(domain: &Subdomain) -> bool {
    let mut opts = ResolverOpts::default();
    opts.timeout = Duration::from_secs(4);

    let dns_resolver = Resolver::new(
        ResolverConfig::default(),
        opts,
    )
    .expect("subdomain resolver: building DNS client");
    dns_resolver.lookup_ip(domain.domain.as_str()).is_ok()
}
```

Listing 3.5: subdomains.rs (Multi-threading)

Điểm nổi bật:

- Sử dụng `reqwest::blocking::Client` cho HTTP request (blocking).
- Sử dụng `trust_dns_resolver::Resolver` cho DNS resolution (blocking).
- Sử dụng `.filter(resolves)` để lọc subdomain dựa trên DNS resolution (sequential).

Phiên bản Async/Await

```rust
use crate::{
    model::{CrtShEntry, Subdomain},
    Error,
};
use futures::stream;
use futures::StreamExt;
use reqwest::Client;
use std::{collections::HashSet, time::Duration};
use trust_dns_resolver::{
    config::{ResolverConfig, ResolverOpts},
    name_server::{GenericConnection, GenericConnectionProvider, TokioRuntime},
    AsyncResolver,
};

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

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

    let mut dns_resolver_opts = ResolverOpts::default();
    dns_resolver_opts.timeout = Duration::from_secs(4);

    let dns_resolver = AsyncResolver::tokio(
        ResolverConfig::default(),
        dns_resolver_opts,
    )
    .expect("subdomain resolver: building DNS client");

    // clean and dedup results
    let mut subdomains: HashSet<String> = entries
        .into_iter()
```

```
        .map(|entry| {
            entry
                .name_value
                .split("\n")
                .map(|subdomain| subdomain.trim().to_string())
                .collect::<Vec<String>>()
        })
        .flatten()
        .filter(|subdomain: &String| subdomain != target)
        .filter(|subdomain: &String| !subdomain.contains("*"))
        .collect();
    subdomains.insert(target.to_string());

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

    Ok(subdomains)
}

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

Listing 3.6: subdomains.rs (Async/Await)

Điểm nổi bật:

- Sử dụng `reqwest::Client` cho HTTP request (non-blocking) và `.await?`.
- Sử dụng `trust_dns_resolver::AsyncResolver` cho DNS resolution (non-blocking).
- Sử dụng `futures::stream::iter()` và `filter_map()` để thực hiện DNS resolution đồng thời cho các subdomain.
- Hàm `enumerate` và `resolves` được đánh dấu là `async fn` và sử dụng `.await` cho các future.

Thay đổi chính: Tương tự như `ports.rs`, phiên bản Async/Await chuyển sang sử dụng các API không đồng bộ cho HTTP request và DNS resolution. Sử dụng stream và `filter_map()` cho phép thực hiện DNS resolution đồng thời cho nhiều subdomain, tăng tốc quá trình liệt kê subdomain.

3.2.5 Đặc điểm ứng dụng quét cổng và subdomain

Ứng dụng quét cổng và subdomain chủ yếu là **I/O-bound**. Thời gian thực thi chủ yếu phụ thuộc vào thời gian chờ đợi các thao tác mạng (HTTP request, DNS resolution, TCP connection). CPU utilization thường không cao, đặc biệt khi số lượng task đồng thời lớn.

3.2.6 Multi-threading (Rayon)

- **Ưu điểm**:
  - Dễ sử dụng cho các tác vụ parallel data processing nhờ parallel iterators.
  - Tận dụng tốt CPU đa nhân cho các tác vụ CPU-bound.
- **Nhược điểm**:
  - Chi phí tạo và quản lý thread có thể đáng kể, đặc biệt khi số lượng thread lớn.
  - Context switching giữa các thread tốn kém tài nguyên.
  - Không tối ưu cho các tác vụ I/O-bound, vì thread có thể bị block khi chờ đợi I/O.

3.2.7 Async/Await (Tokio)

- **Ưu điểm**:
  - Hiệu quả cao cho các tác vụ I/O-bound.
  - Chi phí context switching thấp hơn so với thread, vì sử dụng cooperative multitasking.
  - Phù hợp với các ứng dụng mạng, nơi có nhiều thao tác I/O đồng thời.
- **Nhược điểm**:
  - Phức tạp hơn trong lập trình so với multi-threading, đòi hỏi hiểu rõ về futures, async/await, và event loop.
  - Không tối ưu cho các tác vụ CPU-bound thuần túy.

3.2.8 So sánh tốc độ dự kiến

Do ứng dụng quét cổng và subdomain là I/O-bound, phiên bản Async/Await dự kiến sẽ có hiệu năng tốt hơn hoặc tương đương so với phiên bản Multi-threading, đặc biệt khi tăng số lượng kết nối đồng thời.

- **Liệt kê Subdomain**: Async/Await có thể nhanh hơn do thực hiện HTTP request và DNS resolution không đồng bộ, tránh block thread khi chờ đợi.
- **Quét Cổng**: Async/Await có thể hiệu quả hơn trong việc quản lý hàng ngàn kết nối TCP đồng thời mà không tốn quá nhiều tài nguyên như thread.

Để có kết quả đo đạc chính xác, cần thực hiện benchmark trên cùng một môi trường và mục tiêu quét. Tuy nhiên, về mặt lý thuyết, Async/Await thường vượt trội hơn trong các ứng dụng mạng I/O-bound như thế này.

3.2.9 Bảng so sánh tốc độ (ước tính)

Em sẽ thực hiện quét cổng và subdomain trên cùng một mục tiêu(trang web olm.vn) và đo thời gian thực thi. Bảng dưới đây là kết quả ước tính (cần benchmark thực tế để có số liệu chính xác):

Phương pháp	Thời gian thực thi (ước tính)	Nhận xét
Multi-threading (Rayon)	60 giây	Có overhead thread
Async/Await (Tokio)	10 giây	Tối ưu cho I/O

3.3 Kết luận

Việc chuyển từ Multi-threading sang Async/Await trong ứng dụng quét cổng và subdomain mang lại những thay đổi đáng kể về cải thiện hiệu năng.

- **Thay đổi Code**: Cấu trúc code thay đổi để phù hợp với mô hình lập trình không đồng bộ của Async/Await. Các API blocking được thay thế bằng API non-blocking, và sử dụng futures, streams, channels để quản lý concurrency.
- **Hiệu năng**: Async/Await có thể cải thiện hiệu năng, đặc biệt trong các ứng dụng I/O-bound như quét cổng và subdomain, do giảm overhead context switching và tối ưu hóa việc xử lý các thao tác mạng đồng thời.

Để có kết luận chính xác về hiệu năng, cần thực hiện benchmark thực tế và đo đạc thời gian thực thi trong các điều kiện khác nhau. Tuy nhiên, dựa trên phân tích lý thuyết và đặc điểm của ứng dụng, Async/Await là một lựa chọn tốt để xây dựng các ứng dụng mạng hiệu năng cao trong Rust.

Tài liệu tham khảo

[1] Sylvain Kerkour, *Black Hat Rust*.

[2] Tech With Tim, *Rust Programming Tutorial*.

[3] Let's Get Rusty, *Rust Survival Guide*.

Link mã nguồn ở: Project 2 Rust