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 16 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.

Chương 4

Thêm Module vào Scanner với Trait Objects

4.1 Giới thiệu

Tuần này tập trung vào việc làm cho chương trình quét cổng (scanner) trở nên linh hoạt và dễ mở rộng hơn. Mục tiêu là có thể thêm các "module"quét khác nhau (ví dụ: module kiểm tra tên miền phụ, module quét lỗ hổng HTTP cụ thể) vào scanner một cách dễ dàng.

4.2 Generics và Hạn chế

Generics trong Rust (ví dụ `fn func<T>(input: T)`) cho phép viết mã có thể tái sử dụng với nhiều kiểu dữ liệu khác nhau. Tuy nhiên, khi tạo một collection như `Vec<T>`, Rust cần biết kích thước chính xác của `T` tại thời điểm biên dịch. Nếu chúng ta muốn tạo một `Vec` chứa các module quét khác nhau (ví dụ: một `struct SubdomainScanner` và một `struct HttpVulnerabilityScanner`), chúng có thể có kích thước bộ nhớ khác nhau. Do đó, trình biên dịch sẽ từ chối tạo `Vec` chứa trực tiếp các đối tượng này vì chúng không "đồng dạng"(cùng kích thước).

4.3 Traits (Đặc điểm)

4.3.1 Định nghĩa và Mục đích

Traits trong Rust tương tự như interfaces trong các ngôn ngữ khác. Chúng định nghĩa một tập hợp các hành vi (phương thức) mà một kiểu dữ liệu phải triển khai. Traits cho phép định nghĩa các "hợp đồng"chung mà các kiểu dữ liệu khác nhau có thể tuân thủ.

```
pub trait Module {
    fn name(&self) -> String;
    fn description(&self) -> String;
}
#[async_trait]
pub trait HttpModule: Module { // HttpModule c ng  p h i  l
   Module
    async fn scan(&self, http_client: &Client, endpoint: &str)
        -> Result<Option<HttpFinding>, Error>;
}
```

Bằng cách này, chúng ta có thể đảm bảo rằng mọi module HTTP đều có hàm `scan`, và cũng có các thuộc tính cơ bản như `name` và `description` từ trait `Module`.

4.3.2 Triển khai Trait

Một kiểu dữ liệu cụ thể (struct hoặc enum) có thể "hứa"thực hiện hợp đồng của trait bằng cách sử dụng từ khóa `impl`.

```
struct GitHeadDisclosure; // Module checks file .git/HEAD

impl Module for GitHeadDisclosure {
    fn name(&self) -> String { "Git HEAD Disclosure".to_string
   () }
    fn description(&self) -> String { "Checks for .git/HEAD
   file exposure".to_string() }
}

#[async_trait]
impl HttpModule for GitHeadDisclosure {
    async fn scan(&self, http_client: &Client, endpoint: &str)
        -> Result<Option<HttpFinding>, Error> {
        // ... implementation details ...
```

```
13      }
14  }
```

4.4 Trait Objects (Đối tượng Trait)

4.4.1 Vấn đề Kích thước Khác nhau

Như đã đề cập, chúng ta không thể tạo `Vec<GitHeadDisclosure>` và thêm một `DotEnvDisclosure` vào đó. Chúng ta cũng không thể tạo `Vec<HttpModule>` vì `HttpModule` là một trait, không phải là một kiểu cụ thể có kích thước cố định. Trình biên dịch không biết cần cấp phát bao nhiêu bộ nhớ cho mỗi phần tử.

4.4.2 Giải pháp: Con trỏ và Dynamic Dispatch

Trait Objects giải quyết vấn đề này bằng cách sử dụng con trỏ đến các đối tượng thực tế. Thay vì lưu trữ các đối tượng có kích thước khác nhau trực tiếp trong `Vec`, chúng ta lưu trữ các con trỏ đến chúng.

- **Con trỏ có kích thước cố định:** Tất cả các con trỏ (ví dụ: `Box`, `&`) đều có cùng kích thước bộ nhớ, bất kể đối tượng mà chúng trỏ tới lớn hay nhỏ.

- **Cú pháp `dyn Trait`:** Để tạo một trait object, chúng ta sử dụng cú pháp `dyn Trait`, ví dụ: `Box<dyn HttpModule>` hoặc `&dyn HttpModule`. Từ khóa `dyn` (dynamic) chỉ ra rằng kiểu cụ thể sẽ được xác định động tại thời gian chạy.

- **Fat Pointers:** Một trait object thực chất là một "fat pointer con trỏ mập. Nó không chỉ chứa địa chỉ của dữ liệu (con trỏ tới đối tượng struct) mà còn chứa địa chỉ của một bảng phương thức ảo (vtable). Vtable này chứa các con trỏ đến các hàm cụ thể đã được triển khai cho trait đó bởi kiểu dữ liệu cụ thể.

- **Dynamic Dispatch:** Khi một phương thức được gọi trên một trait object (ví dụ `module.scan(...)`), chương trình sẽ sử dụng vtable tại thời gian chạy để tìm và gọi đúng hàm đã được triển khai cho kiểu dữ liệu cụ thể đó. Quá trình này được gọi là dynamic dispatch, trái ngược với static dispatch (xác định hàm tại thời điểm biên dịch) được sử dụng với generics. Dynamic dispatch mang lại sự linh hoạt nhưng có thể tốn thêm một chút chi phí hiệu năng do việc tra cứu vtable.

4.5 Áp dụng vào Chương trình Quét Cổng

Với trait objects thì có thể tạo một danh sách chứa nhiều loại module quét khác nhau, miễn là chúng đều triển khai cùng một trait (`HttpModule`).

```
// initiate different module HTTP lists
let http_modules: Vec<Box<dyn HttpModule>> = vec![
    Box::new(http::GitHeadDisclosure::new()),
    Box::new(http::DotEnvDisclosure::new()),
    Box::new(http::GitlabOpenRegistrations::new()),
];

for module in http_modules {
    println!("* Scanning with module: {}", module.name());
    match module.scan(&http_client, &target_url).await {
        Ok(Some(finding)) => {
            println!("    -> Finding: {:?}", finding);
            // ....
        }
        Ok(None) => {
            // ....
        }
        Err(err) => {
            eprintln!("Error scanning with {}: {}", module.
   name(), err);
        }
     }
}
```

Trong đoạn mã trên:

1. Chúng ta tạo một `Vec` chứa các `Box<dyn HttpModule>`. Mỗi `Box` là một con trỏ trên heap tới một module cụ thể (ví dụ: `GitHeadDisclosure`, `DotEnvDisclosure`). Vì tất cả các `Box` có cùng kích thước, nên `Vec` này hợp lệ.

2. Chúng ta có thể lặp qua vector này và gọi phương thức `scan` trên từng `module`. Mặc dù trình biên dịch không biết kiểu cụ thể của từng module tại thời điểm biên dịch, nó biết rằng mỗi phần tử đều triển khai trait `HttpModule` và có phương thức `scan`.

3. Tại thời gian chạy, thông qua dynamic dispatch và vtable, lời gọi `module.scan(...)` sẽ thực thi đúng phiên bản của hàm `scan` tương ứng với kiểu dữ liệu cụ thể của module đó.

Điều này làm cho chương trình scanner trở nên rất linh hoạt. Chúng ta có thể dễ dàng thêm các module mới bằng cách tạo một struct mới và triển khai trait `HttpModule` (và `Module`) cho nó, sau đó chỉ cần thêm một thực thể của nó vào vector `http_modules` mà không cần sửa đổi logic cốt lõi của vòng lặp quét.

4.6 Kết luận

Chương 4 giới thiệu một khái niệm quan trọng trong Rust là Traits và Trait Objects. Traits định nghĩa các hành vi chung, trong khi Trait Objects (`dyn Trait`) cung cấp một cơ chế mạnh mẽ để làm việc với các kiểu dữ liệu khác nhau có cùng hành vi tại thời gian chạy. Bằng cách sử dụng con trỏ và dynamic dispatch, trait objects cho phép tạo ra các collection không đồng nhất về kiểu cụ thể nhưng đồng nhất về hành vi, giải quyết vấn đề về kích thước khác nhau trong bộ nhớ. Áp dụng kỹ thuật này vào chương trình quét cổng giúp tạo ra một hệ thống module hóa, linh hoạt và dễ dàng mở rộng.

Chương 5

Web Crawling cho OSINT

5.1 Giới thiệu Tổng quan

Tình báo nguồn mở (OSINT) đóng vai trò then chốt trong việc thu thập thông tin từ các nguồn dữ liệu công khai rộng lớn trên Internet. Tuy nhiên, việc khai thác hiệu quả nguồn tài nguyên này đòi hỏi các công cụ tự động hóa mạnh mẽ. Các máy tìm kiếm thông thường, mặc dù hữu ích, lại bị giới hạn bởi các quy tắc như `robots.txt` và không thể truy cập nội dung yêu cầu xác thực.

Để giải quyết những thách thức này, việc xây dựng một web crawler tùy chỉnh là một giải pháp cần thiết.

5.2 Cấu trúc của Crawler

Hệ thống crawler được thiết kế theo một kiến trúc module hóa, bao gồm các thành phần logic chính sau đây, nhằm đảm bảo khả năng bảo trì, mở rộng và quản lý hiệu quả:

- **Start URLs:** Điểm khởi đầu của quá trình crawl, là một tập hợp các địa chỉ URL ban đầu được cung cấp cho hệ thống.
- **Spiders:** Các thành phần chuyên biệt, được lập trình để tương tác và trích xuất thông tin từ một loại website hoặc API cụ thể. Mỗi spider thực hiện hai nhiệm vụ cốt lõi:
  - **Scraping (Thu thập và Trích xuất):** Gửi yêu cầu đến URL mục tiêu, tải và phân tích nội dung (HTML, JSON,...), sau đó trích xuất dữ liệu cần thiết (gọi là *items*) và các liên kết URL mới để tiếp tục quá trình crawl.

  - **Processing (Xử lý):** Nhận các *items* đã được trích xuất và thực hiện các hành động tiếp theo, như lưu trữ, hiển thị, hoặc phân tích sâu hơn.

- **Crawler Engine (Bộ điều khiển):** Hoạt động như bộ não trung tâm, điều phối luồng công việc. Thành phần này quản lý hàng đợi URL cần truy cập, phân phối công việc cho các spider, thu nhận kết quả (items và URL mới), đảm bảo các URL không bị crawl lặp lại, và quyết định thời điểm dừng toàn bộ hệ thống.

Kiến trúc này cho phép cấu hình và vận hành các tác vụ scraping và processing với các mức độ đồng thời (concurrency) khác nhau, giúp tối ưu hóa tài nguyên và tuân thủ các quy tắc ứng xử mạng (ví dụ: tránh làm quá tải máy chủ đích).

5.3 Lựa chọn Rust và Ưu điểm Kỹ thuật

Sử dụng Rust cho việc triển khai crawler này vì những lợi thế kỹ thuật đặc biệt phù hợp với các ứng dụng mạng và xử lý dữ liệu cường độ cao:

- **Xử lý Bất đồng bộ Hiệu quả:** Mô hình `async/.await` của Rust, kết hợp với runtime `tokio`, cho phép quản lý hàng ngàn tác vụ I/O (đặc biệt là kết nối mạng) đồng thời một cách hiệu quả mà không tốn kém tài nguyên như mô hình đa luồng truyền thống.

- **Quản lý Bộ nhớ An toàn và Hiệu năng:** Rust không sử dụng Garbage Collector. Hệ thống sở hữu (ownership) và mượn (borrowing) đảm bảo an toàn bộ nhớ tại thời điểm biên dịch, loại bỏ các độ trễ không đoán trước do GC và cho phép sử dụng bộ nhớ hiệu quả, ổn định, đặc biệt quan trọng khi xử lý lượng lớn dữ liệu tạm thời trong quá trình parsing.

- **Độ Tin cậy khi Phân tích Dữ liệu:** Việc phân tích (parsing) dữ liệu từ các nguồn không đáng tin cậy là một điểm tiềm ẩn nhiều lỗi. Các tính năng an toàn bộ nhớ và hệ thống xử lý lỗi tường minh (`Result`, `Option`) của Rust giúp xây dựng các bộ parser mạnh mẽ, giảm thiểu nguy cơ crash và các lỗ hổng bảo mật liên quan đến bộ nhớ.

5.4 Phân tích cụ thể

5.4.1 (main.rs)

Dự án được tổ chức với thư mục src chứa mã nguồn chính, bao gồm file điểm vào main.rs, module điều khiển crawler.rs, và module spiders chứa các spider cụ thể (cvedetails.rs, github.rs, quotes.rs).

File main.rs đóng vai trò khởi tạo và điều phối ban đầu:

- **Xử lý Tham số Dòng lệnh:** Sử dụng thư viện clap để định nghĩa và phân tích các đối số dòng lệnh, cho phép người dùng chỉ định spider cần chạy (run -spider <name>) hoặc liệt kê các spider có sẵn (spiders).
- **Khởi tạo Môi trường:** Thiết lập logging thông qua env_logger.
- **Lựa chọn và Chạy Spider:** Dựa trên tham số đầu vào, tạo một instance của Crawler và spider tương ứng (ví dụ: CveDetailsSpider, GitHubSpider, QuotesSpider). Spider được bọc trong Arc để có thể chia sẻ an toàn giữa các tác vụ bất đồng bộ. Cuối cùng, hàm crawler.run(spider).await được gọi để bắt đầu quá trình crawl.

```
1
2 #[tokio::main]
3 async fn main() -> Result<(), anyhow::Error> { // anyhow for
     error handling
4
5     let cli = Command::new( /* ... */ ).get_matches();
6
7     env::set_var("RUST_LOG", "info,crawler=debug");
8     env_logger::init();
9
10    if let Some(_) = cli.subcommand_matches("spiders") {
11        // spiders
12    } else if let Some(matches) = cli.subcommand_matches("run"
     ) {
13        let spider_name = matches.value_of("spider").unwrap();
      // safety due to required(true)
14        // initiate Crawler with concurrency and delay
15        let crawler = Crawler::new(Duration::from_millis(200),
      2, 500);
16
```

```
        match spider_name {
            "cvedetails" => {
                // wrap in Arc
                let spider = Arc::new(spiders::cvedetails::
    CveDetailsSpider::new());
                // run crawler
                crawler.run(spider).await;
            }
            "github" => {
                let spider = Arc::new(spiders::github::
    GitHubSpider::new());
                crawler.run(spider).await;
            }
            "quotes" => {
                // QuotesSpider::new() is async, so await
                let spider = spiders::quotes::QuotesSpider::
    new().await?;
                let spider = Arc::new(spider);
                crawler.run(spider).await;
            }
            _ => return Err(Error::InvalidSpider(spider_name.
    to_string()).into()),
        };
     }

     Ok(())
}
```

Listing 5.1: Khởi tạo và chạy spider trong src/main.rs

5.4.2 Trừu tượng hóa Spider: Trait `Spider` (`spiders/mod.rs`)

Để `Crawler Engine` có thể hoạt động với nhiều loại spider khác nhau mà không cần biết chi tiết bên trong, một trait `Spider` được định nghĩa làm giao diện chung. Trait này sử dụng Associated Type để định nghĩa kiểu dữ liệu `Item` mà mỗi spider sẽ tạo ra.

```
use crate::error::Error;
use async_trait::async_trait;

pub mod cvedetails;
pub mod github;
pub mod quotes;
```

```rust

#[async_trait]
pub trait Spider: Send + Sync { // safe to send and sync

    type Item: Send + 'static; // Item also Send

    fn name(&self) -> String;
    fn start_urls(&self) -> Vec<String>; // URL list

    // scraping, return items and new URLs
    async fn scrape(&self, url: String) -> Result<(Vec<Self::Item>, Vec<String>), Error>;

    // handle scraped data
    async fn process(&self, item: Self::Item) -> Result<(), Error>;
}
```

Listing 5.2: Định nghĩa Trait Spider trong src/spiders/mod.rs

Mọi spider cụ thể (như `CveDetailsSpider`) sẽ cần `impl Spider for ...` và cung cấp hiện thực cho các phương thức này.

5.4.3 Bộ điều khiển Trung tâm: `Crawler` (`crawler.rs`)

File `crawler.rs` chứa định nghĩa struct `Crawler` và logic điều phối chính trong phương thức `run`.

Cấu trúc `Crawler`: Lưu trữ các tham số cấu hình như độ trễ (`delay`) và mức độ đồng thời cho crawling và processing.

```rust
pub struct Crawler {
    delay: Duration,
    crawling_concurrency: usize,
    processing_concurrency: usize,
}

impl Crawler {
    pub fn new(
        delay: Duration,
        crawling_concurrency: usize,
        processing_concurrency: usize,
    ) -> Self {
```

```
        Crawler {
            delay,
            crawling_concurrency,
            processing_concurrency,
        }
    }

}
```

Listing 5.3: Định nghĩa Struct Crawler trong src/crawler.rs

Phương thức `run`**:** Thực hiện các bước:

1. **Khởi tạo:** Tạo các kênh `mpsc` để giao tiếp (`urls_to_visit`, `items`, `new_urls`), `HashSet` cho `visited_urls`, bộ đếm `AtomicUsize` cho `active_spiders` (bọc trong `Arc`), và `Barrier` để đồng bộ hóa kết thúc.

```
let mut visited_urls = HashSet::<String>::new();
let crawling_concurrency = self.crawling_concurrency;
let crawling_queue_capacity = crawling_concurrency * 400;
   // capacity
let processing_concurrency = self.processing_concurrency;
let processing_queue_capacity = processing_concurrency *
   10;
let active_spiders = Arc::new(AtomicUsize::new(0)); //
   atomic

// create MPSC channels
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);
// Barrier waits the control loop, scrapers, processors
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; // send url
   to channel
```

```
}
```

Listing 5.4: Khởi tạo tài nguyên trong Crawler::run

2. **Khởi chạy Tác vụ Nền:** Gọi `self.launch_processors(...)` và `self.launch_scrapers(...)` để tạo các tác vụ `tokio` chạy nền, xử lý việc scraping và processing một cách đồng thời.

3. **Vòng lặp Điều khiển Chính:** Liên tục kiểm tra kênh `new_urls_rx` để nhận kết quả từ scraper. Nếu nhận được URL mới chưa có trong `visited_urls`, thêm vào set và gửi vào kênh `urls_to_visit_tx`.

```
loop {
    // receive output from scraper (non-blocking)
    if let Some((visited_url, new_urls)) = new_urls_rx.try_recv().ok() {
        visited_urls.insert(visited_url); // mark done URL

        for url in new_urls {
            if !visited_urls.contains(&url) { // new URL
                visited_urls.insert(url.clone()); // then add
                log::debug!("queueing: {}", url);
                // send to queue
                let _ = urls_to_visit_tx.send(url).await;
            }
        }
    }

    // stop condition: empty channel and no working scrapers
    if new_urls_tx.capacity() == crawling_queue_capacity
    && urls_to_visit_tx.capacity() == crawling_queue_capacity // urls_to_visit is empty
    && active_spiders.load(Ordering::SeqCst) == 0
    {
        log::info!("crawler: No more work detected. Exiting control loop.");
        break;
    }

    // to avoid busy-waiting
    sleep(Duration::from_millis(5)).await;
```

```
}

log::info!("crawler: Control loop exited.");

// announce stream scraper ended
drop(urls_to_visit_tx);

// wait for all (scrapers, processors) finish at barrier
barrier.wait().await;
```

Listing 5.5: Vòng lặp điều khiển và điều kiện dừng trong Crawler::run

4. **Kết thúc và Đồng bộ hóa:** Sau vòng lặp, đóng kênh `urls_to_visit_tx` và đợi ở `barrier.wait().await` để đảm bảo mọi tác vụ con đã hoàn thành trước khi hàm `run` trả về.

Quản lý Đồng thời (`launch_scrapers`, `launch_processors`)**:** Các hàm này sử dụng `tokio::spawn` để tạo tác vụ nền. Bên trong, `tokio_stream::wrappers::ReceiverStream` và `StreamExt::for_each_concurrent` được dùng để xử lý các URL (hoặc items) từ kênh nhận với mức độ đồng thời được giới hạn bởi `crawling_concurrency` (hoặc `processing_concurrency`).

```

tokio_stream::wrappers::ReceiverStream::new(urls_to_visit)
    .for_each_concurrent(concurrency, |queued_url| {
        // ... clone Arc<...> and tx channels ...
        async move {
            active_spiders.fetch_add(1, Ordering::SeqCst); // increment
            // call the spider.scrape(queued_url).await
            // handle output by sending items through items_tx, send new_urls through new_urls_tx
            sleep(delay).await;
            active_spiders.fetch_sub(1, Ordering::SeqCst); // decrement
        }
    })
    .await;

drop(items_tx); // close the items channel after finish scrape stream
barrier.wait().await;
```

```
// ... }) ...
```

Listing 5.6: Quản lý concurrency trong launch_scrapers

Bộ đếm `active_spiders` và `barrier` đóng vai trò quan trọng trong việc phối hợp và đảm bảo kết thúc đúng đắn.

5.4.4 Hiện thực các Spider Cụ thể

Các kịch bản khác nhau:

1. `CveDetailsSpider` **(HTML Parsing -** `spiders/cvedetails.rs`**):**

- Sử dụng `reqwest::Client` để tải HTML từ `cvedetails.com`.
- Định nghĩa struct `Cve` để lưu trữ thông tin lỗ hổng.
- Phương thức `scrape` sử dụng thư viện `select` để phân tích tài liệu HTML bằng CSS selectors (`Attr("id", ...)`, `Class(...)`, `Name(...)`). Nó duyệt qua các hàng (`<tr>`) của bảng, trích xuất dữ liệu từ các ô (`<td>`), và tạo các đối tượng `Cve`. Nó cũng tìm các link phân trang (`pagingb a`) để trả về cho việc crawl tiếp theo.

```rust
#[derive(Debug, Clone)]
pub struct Cve {
    name: String,
    url: String,
    cwe_id: Option<String>,
    score: f32,
    availability: String,
}

let http_res = self.http_client.get(url).send().await?.text().
    await?;
let document = Document::from(http_res.as_str());
let mut items = Vec::new();

// select rows in data table
let rows = document.select(Attr("id", "vulnslisttable").
    descendant(Class("srrowns")));
for row in rows {
    let mut columns = row.select(Name("td"));
    let _ = columns.next(); // skip '#'
```

```
19      let cve_link = columns.next().unwrap().select(Name("a")).
    next().unwrap();
20      let cve_name = cve_link.text().trim().to_string();
21      let cve_url = self.normalize_url(cve_link.attr("href").
    unwrap());
22      let score: f32 = columns.next().unwrap().text().trim().
    parse().unwrap_or(0.0); // parsing error
23
24  }
25
26  // next link
27  let next_pages_links = document
28      .select(Attr("id", "pagingb").descendant(Name("a")))
29      .filter_map(|n| n.attr("href"))
30      .map(|url| self.normalize_url(url))
31      .collect::<Vec<String>>();
32
33  Ok((items, next_pages_links))
```

Listing 5.7: Struct Cve trong cvedetails.rs

2. `GitHubSpider` **(JSON API -** `spiders/github.rs`**):**

- Sử dụng `reqwest::Client` được cấu hình với header `Accept` và `User-Agent` phù hợp cho GitHub API.
- Định nghĩa struct `GitHubItem` với `#[derive(Deserialize)]` để `serde` tự động parse JSON.
- Phương thức `scrape` gọi API, sử dụng `response.json::<Vec<GitHubItem>>().await?` để deserialize kết quả. Logic phân trang được xử lý bằng cách kiểm tra số lượng kết quả trả về; nếu đủ `expected_number_of_results`, nó sử dụng regex (`page_-regex`) để tìm số trang hiện tại và tạo URL cho trang tiếp theo.

```
1  use serde::Deserialize;
2
3  #[derive(Debug, Clone, Deserialize)]
4  pub struct GitHubItem {
5      login: String,
6      id: u64,
7      node_id: String,
8      html_url: String,
9      avatar_url: String,
```

```
}


// direct deserialize from response
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 {
     //use regex to find and replace page num
     let captures = self.page_regex.captures(&url).ok_or_else
    (|| Error::Internal("Failed to capture page number".into())
    )?;
     let old_page_number_str = captures.get(1).unwrap().as_str
    ();
     let old_page_number: usize = old_page_number_str.parse()
         .map_err(|_| Error::Internal(format!("Failed to parse
    page number: {}", old_page_number_str)))?;
     let new_page_number = old_page_number + 1;

     let next_url = url.replace(
         &format!("&page={}", old_page_number), //formatting
         &format!("&page={}", new_page_number)
     );
     vec![next_url]
} else {
     Vec::new()
};

Ok((items, next_pages_links))
```

Listing 5.8: Struct GitHubItem và logic phân trang trong github.rs

3. `QuotesSpider` **(JavaScript-required Page -** `spiders/quotes.rs`**):**

- Sử dụng thư viện `fantoccini` để điều khiển trình duyệt headless (thông qua WebDriver). Client `fantoccini::Client` được bọc trong `tokio::sync::Mutex` để đảm bảo an toàn khi chia sẻ giữa các tác vụ scraper.
- Phương thức `scrape` lấy lock từ `Mutex`, điều khiển trình duyệt truy cập URL (`webdriver.goto(...)`), lấy mã nguồn HTML *sau khi* JavaScript đã thực thi (`webdriver.source().await?`), sau đó dùng `select` để phân tích HTML này và trích xuất các câu quote, tác giả.

```rust
use fantoccini::{Client}; //import Client
use tokio::sync::Mutex; //import Mutex

pub struct QuotesSpider {
    // Client wrapped in Mutex for sharing safety
    webdriver_client: Mutex<Client>,
}

// in scrape()
let mut items = Vec::new();
let html = { //independent scope for MutexGuard to drop early
    let mut webdriver = self.webdriver_client.lock().await;
    webdriver.goto(&url).await?;
    // adding webdriver.wait() for complicated dynamic loading webpage
    webdriver.source().await? // get HTML after JS
}; //drop MutexGuard, free lock

let document = Document::from(html.as_str());
// use select to analyze doc
let quotes = document.select(Class("quote"));
for quote_node in quotes {
    items.push(QuotesItem { /* ... */ });
}


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)) // use helper()
    .collect::<Vec<String>>();

Ok((items, next_pages_link))
```

Listing 5.9: Sử dụng Mutex và WebDriver(quotes.rs)

5.4.5 Xử lý Lỗi

- **Sử dụng** `Result<T, E>`**:** Hầu hết các hàm có thể thất bại (I/O, parsing, ...) đều trả về `Result`..

- **Sử dụng** `anyhow::Error` **trong** `main`**:** Giúp đơn giản hóa việc xử lý lỗi ở cấp cao nhất bằng cách cho phép các loại lỗi khác nhau được quy về một kiểu lỗi chung.

Việc xử lý lỗi tường minh là một điểm mạnh của Rust, giúp tăng độ tin cậy của crawler.

5.5 Tổng kết

Kiến trúc module hóa với trait `Spider` làm trung tâm cho phép dễ dàng mở rộng hỗ trợ cho các nguồn dữ liệu mới. Việc quản lý trạng thái và điều phối đồng thời được thực hiện một cách an toàn và hiệu quả thông qua các cơ chế như kênh `mpsc`, kiểu dữ liệu nguyên tử `AtomicUsize`, và cơ chế đồng bộ hóa `Barrier`. Các ví dụ triển khai spider cụ thể cho thấy khả năng xử lý đa dạng các loại website và API, từ HTML tĩnh đến các trang yêu cầu JavaScript phức tạp.

Nhìn chung, Rust có thể xây dựng các hệ thống mạng hiệu năng cao, đáng tin cậy, phù hợp cho các ứng dụng đòi hỏi khắt khe như web crawling và thu thập dữ liệu quy mô lớn trong lĩnh vực OSINT. Trong tuần sau em sẽ viết thêm 1 spider cho 1 trang web cụ thể và tìm hiểu về chương 6: Finding Vulnerabilities.

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