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 18 tháng 3 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;
```

```
use std::thread;

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

# 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