**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

**Chương 1: Các Chủ đề Đồ án liên quan đến Rust và Bảo mật**

**1.1. Đồ án 1: Evil Twin Attack (Cấu hình mạng và Captive Portal tĩnh)**

1.1.1. Mục tiêu  
Xây dựng một điểm truy cập WiFi giả mạo (Evil Twin) để lừa các thiết bị tự động kết nối và thu thập thông tin đăng nhập của người dùng thông qua một trang đăng nhập giả mạo (Captive Portal).

1.1.2. Mô tả chung  
Đồ án cần thiết lập cấu hình mạng cần thiết trên một thiết bị chạy Linux (như Raspberry Pi hoặc laptop chạy Linux) để tạo điểm truy cập giả mạo có cùng Tên mạng (SSID) với một mạng WiFi thật. Khi thiết bị của nạn nhân kết nối, lưu lượng HTTP sẽ bị chuyển hướng (redirect) đến một máy chủ web đơn giản phục vụ trang đăng nhập giả. Thông tin đăng nhập sẽ được thu thập và ghi lại.

Phần ứng dụng Rust sẽ là một công cụ hỗ trợ tự động hóa việc tạo script cấu hình mạng.

1.1.3. Các bước thực hiện chính  
Quá trình thực hiện Evil Twin Attack bao gồm các bước chính sau:

1. Chuẩn bị thiết bị: Cài đặt hệ điều hành Linux (ví dụ: Raspberry Pi OS Lite, Ubuntu) trên thiết bị có ít nhất một card WiFi hỗ trợ chế độ Access Point (AP mode) và một card mạng thứ hai (Ethernet hoặc WiFi khác) để quản lý và kết nối internet (nếu cần).
2. Cấu hình Interface mạng: Gán địa chỉ IP tĩnh cho interface WiFi sẽ dùng làm AP giả.
3. Thiết lập Access Point giả: Sử dụng công cụ hostapd để tạo điểm truy cập WiFi với SSID và cấu hình bảo mật giống với mạng WiFi thật muốn giả mạo.
4. Thiết lập DHCP và DNS giả: Sử dụng công cụ dnsmasq để cấp địa chỉ IP cho các thiết bị kết nối và chuyển hướng (redirect) tất cả các yêu cầu DNS về địa chỉ IP của thiết bị tấn công.
5. Cấu hình chuyển hướng lưu lượng: Sử dụng iptables (hoặc nftables) để cấu hình firewall và NAT, đặc biệt là chuyển hướng lưu lượng HTTP (port 80) từ các thiết bị kết nối đến máy chủ web Captive Portal chạy trên thiết bị tấn công.
6. Xây dựng máy chủ web Captive Portal: Tạo một máy chủ web đơn giản để phục vụ file HTML của trang đăng nhập giả và xử lý các yêu cầu POST chứa thông tin đăng nhập, ghi thông tin này vào file.
7. Phát triển công cụ Rust hỗ trợ: Xây dựng một công cụ CLI Rust để nhận SSID mục tiêu làm input và tự động tạo file script shell (evil\_twin.sh) chứa các lệnh cấu hình hostapd, dnsmasq, iptables đã điền sẵn SSID.
8. Thực thi tấn công: Chạy script shell cấu hình mạng và máy chủ web Captive Portal trên thiết bị tấn công.
9. Thu thập và phân tích log: Sử dụng công cụ Rust khác (tùy chọn) để đọc và phân tích file chứa thông tin đăng nhập đã thu thập.

2.1.4. Các công cụ và Crate Rust cần dùng

* Công cụ hệ thống: hostapd, dnsmasq, iptables (hoặc nftables), hệ điều hành Linux, card WiFi hỗ trợ AP mode.
* Máy chủ web và script thu thập log.
* Crate Rust cần dùng cho công cụ CLI hỗ trợ:
  + clap: Để xử lý các tham số dòng lệnh, nhận SSID mục tiêu.
  + std::fs: Để làm việc với file, ghi nội dung script shell vào file.
  + std::fmt / format!: Để định dạng chuỗi, tạo nội dung script shell từ template và SSID.
  + std::result::Result, anyhow: Để xử lý lỗi một cách hiệu quả.
  + (Tùy chọn cho công cụ phân tích log): Các crate xử lý chuỗi nâng cao, có thể là regex (regex).

2.1.6. Kế hoạch thực hiện trong 5 tuần (Xây dựng ứng dụng web Captive Portal bằng Rust)  
Nếu tập trung vào việc xây dựng một máy chủ web đơn giản bằng Rust cho phần Captive Portal kế hoạch 5 tuần có thể như sau:  
Tuần | Mục tiêu | Kết quả | Công việc chính  
----|---------|--------|---------------  
1 | Tìm hiểu Web Framework Rust | Hiểu cơ bản về 1 web framework (ví dụ: warp, axum), cài đặt môi trường | Đọc tài liệu về warp/axum, tạo project Rust mới, cài đặt framework, chạy ví dụ "Hello World", tìm hiểu cách xử lý request GET cơ bản.  
2 | Phục vụ file tĩnh và Form HTML | Webserver trả về file HTML, hiểu cách xử lý request POST | Tìm hiểu cách framework phục vụ file tĩnh (warp::fs::file(), axum::routing::get\_service), tạo file HTML trang đăng nhập giả, cấu hình route GET / để trả về HTML đó. Tìm hiểu cách framework xử lý request POST và lấy request body.  
3 | Thu thập dữ liệu Form | Nhận và parse dữ liệu username/password từ request POST | Tìm hiểu cách parse dữ liệu từ request body (ví dụ: warp::body::form(), axum::form::Form hoặc dùng serde\_urlencoded), trích xuất username và password. In thông tin ra console để debug.  
4 | Lưu trữ Credentials và Redirect | Credentials được lưu vào file, người dùng bị chuyển hướng | Tìm hiểu std::fs để làm việc với file, mở file (append mode), ghi username/password vào file. Tìm hiểu cách trả về response redirect (status 302, header Location). Kiểm tra luồng hoạt động từ trình duyệt.  
5 | Hoàn thiện và Đóng gói | Webserver ổn định cơ bản, có log, có thể đóng gói | Thêm logging (log, env\_logger), xử lý các trường hợp lỗi cơ bản, tối ưu kích thước binary (nếu cần), đóng gói binary cuối cùng, viết hướng dẫn sử dụng/triển khai.

**2.2. Đồ án 2: Simple Remote Keylogger(for a Single OS)**

2.2.1. Mục tiêu  
Xây dựng một keylogger cơ bản chỉ hoạt động trên một hệ điều hành cụ thể (Windows hoặc Linux/macOS) và có khả năng gửi dữ liệu phím bấm thu thập được đến một điểm cuối từ xa.

2.2.2. Mô tả chung  
Đây là đồ án tập trung vào việc tương tác cấp thấp với hệ điều hành để bắt sự kiện nhấn phím. Chương trình (Agent) sẽ chạy trên máy nạn nhân, ghi lại các phím bấm (chỉ ký tự cơ bản trong phạm vi đơn giản) và gửi dữ liệu này định kỳ qua mạng đến một máy chủ thu thập đơn giản.

2.2.3. Các bước thực hiện chính

1. Chọn Hệ điều hành mục tiêu: Xác định rõ OS nào sẽ là mục tiêu chính của keylogger (ví dụ: Windows).
2. Tìm hiểu API bắt phím của OS: Nghiên cứu cách bắt các sự kiện nhấn phím toàn cục trên OS đã chọn. Điều này thường liên quan đến API hệ thống cấp thấp (WinAPI trên Windows, X11/Wayland/Input Subsystem trên Linux, I/O Kit/Event Taps trên macOS).
3. Tương tác với API OS bằng Rust: Sử dụng Foreign Function Interface (FFI) với libc (cho Linux) hoặc tìm các crate Rust có sẵn cung cấp binding/wrapper an toàn hơn cho việc bắt phím trên OS mục tiêu.
4. Triển khai logic bắt phím: Viết code Rust để nhận các sự kiện phím bấm từ API OS, chuyển đổi chúng thành ký tự có thể đọc được.
5. Đệm và Thu thập log: Lưu trữ các ký tự đã bắt được vào bộ nhớ đệm (buffer) hoặc một danh sách.
6. Gửi log định kỳ: Sử dụng một luồng (thread) hoặc tác vụ bất đồng bộ (async task) khác để định kỳ gửi nội dung bộ đệm log đến một địa chỉ từ xa qua mạng.
7. Chuẩn bị điểm cuối thu thập (Server): Một máy chủ đơn giản (không cần viết bằng Rust nếu muốn giữ phạm vi nhỏ) để nhận dữ liệu gửi đến qua HTTP POST và ghi vào file.
8. Đóng gói Binary: Biên dịch và đóng gói binary Rust cho hệ điều hành mục tiêu.

2.2.4. Các công cụ và Crate Rust cần dùng

* Tương tác với OS: Cần tìm crate Rust hoặc sử dụng FFI (libc) để gọi các hàm API hệ điều hành liên quan đến keyboard hooking. Ví dụ trên Windows có thể dùng winapi hoặc các crate cụ thể hơn.
* Networking Client: reqwest hoặc ureq để gửi dữ liệu qua HTTP.
* Quản lý thời gian: std::thread::sleep hoặc tokio::time::sleep để gửi log định kỳ.
* Xử lý dữ liệu: Các hàm xử lý chuỗi, byte array.
* Xử lý lỗi: Result, anyhow.
* Logging: log, env\_logger để ghi log hoạt động của keylogger.

2.2.5. Phạm vi Đồ án

* Tập trung vào việc bắt phím thành công trên một OS duy nhất và gửi dữ liệu thô (chưa định dạng) đến một điểm cuối từ xa đơn giản (ví dụ: HTTP endpoint ghi vào file text). Các khía cạnh về persistence, stealth, xử lý phím đặc biệt, mã hóa dữ liệu có thể bị bỏ qua hoặc rất cơ bản. Cần tương tác cấp thấp với OS.
* Mở rộng phạm vi bao gồm: hỗ trợ đa nền tảng (cross-platform), bắt phím nâng cao (xử lý phím đặc biệt, window title), buffer log hiệu quả hơn, mã hóa dữ liệu log trước khi gửi (sử dụng AEAD), triển khai persistence cho nhiều OS, áp dụng các kỹ thuật né tránh phát hiện cơ bản, xây dựng C&C server đơn giản bằng Rust để nhận/hiển thị log.

**2.3. Đồ án 3: Secure File Encryption Tool**

2.3.1. Mục tiêu  
Xây dựng một công cụ dòng lệnh (CLI) bằng Rust để mã hóa và giải mã các file một cách an toàn và đáng tin cậy, tuân thủ các nguyên tắc mật mã hiện đại.

2.3.2. Mô tả chung  
Đồ án này tập trung vào việc sử dụng đúng và an toàn các primitive mật mã để bảo vệ tính bảo mật và toàn vẹn của file. Công cụ sẽ nhận file, mật khẩu (hoặc key) và tùy chọn mã hóa/giải mã làm đầu vào, sau đó thực hiện tác vụ tương ứng.

2.3.3. Các bước thực hiện chính

1. Thiết kế giao diện CLI: Sử dụng crate clap để định nghĩa các lệnh (encrypt, decrypt) và các tùy chọn (input file, output file, password/key).
2. Tìm hiểu và Sử dụng KDF: Lựa chọn và sử dụng một crate cung cấp hàm KDF an toàn (ví dụ: pbkdf2 hoặc argon2). Triển khai logic để tạo key mã hóa mạnh từ mật khẩu người dùng, kết hợp với một salt ngẫu nhiên và duy nhất cho mỗi lần mã hóa. Salt cần được lưu cùng với ciphertext.
3. Tìm hiểu và Sử dụng AEAD: Lựa chọn và sử dụng một crate cung cấp thuật toán AEAD hiện đại (ví dụ: chacha20poly1305 hoặc aes-gcm). Hiểu cách sử dụng thuật toán này, bao gồm việc tạo một Nonce (IV) ngẫu nhiên và duy nhất cho mỗi lần mã hóa với cùng một key.
4. Triển khai Mã hóa file: Đọc nội dung file đầu vào, sử dụng key KDF và AEAD đã chọn với Nonce mới tạo để mã hóa. Kết quả sẽ là ciphertext cùng với authentication tag. Lưu Salt, Nonce và Ciphertext (thường theo một định dạng cụ thể, ví dụ: Salt || Nonce || Ciphertext || Tag) vào file đầu ra.
5. Triển khai Giải mã file: Đọc nội dung file đầu vào, trích xuất Salt, Nonce, Ciphertext, và Tag. Sử dụng mật khẩu người dùng và Salt để tái tạo key KDF. Sử dụng key này, Nonce, Ciphertext, và Tag để gọi hàm giải mã của thuật toán AEAD. Hàm này sẽ tự động kiểm tra Tag để xác minh tính toàn vẹn. Nếu Tag hợp lệ, nó trả về plaintext; nếu không, báo lỗi. Ghi plaintext vào file đầu ra.
6. Xử lý lỗi: Triển khai xử lý lỗi chi tiết cho các trường hợp như file không tồn tại, mật khẩu sai, file mã hóa bị hỏng, lỗi ghi file, v.v.
7. Xóa dữ liệu nhạy cảm khỏi bộ nhớ: Sử dụng crate zeroize để xóa key và các dữ liệu nhạy cảm khác khỏi bộ nhớ sau khi sử dụng.

2.3.4. Các công cụ và Crate Rust cần dùng

* CLI: clap.
* KDF: pbkdf2, argon2.
* AEAD: chacha20poly1305, aes-gcm.
* Ngẫu nhiên: rand để tạo Salt và Nonce.
* Xóa dữ liệu nhạy cảm: zeroize.
* Làm việc với file: std::fs.
* Xử lý lỗi: Result, anyhow.
* Xử lý byte array và chuỗi.

2.3.5. Phạm vi Đồ án

* Triển khai chức năng mã hóa/giải mã file sử dụng một KDF và một thuật toán AEAD cố định đã chọn. Hỗ trợ file có kích thước vừa phải (có thể đọc toàn bộ vào bộ nhớ). Giao diện CLI đơn giản. Tuân thủ các nguyên tắc cơ bản về KDF, Nonce/IV và AEAD.
* Mở rộng phạm vi bao gồm: hỗ trợ file kích thước lớn (streaming encryption/decryption), hỗ trợ nhiều thuật toán KDF và AEAD khác nhau (có thể lựa chọn qua CLI), thêm các tính năng quản lý key (ví dụ: lưu key đã tạo ra file riêng và mã hóa key đó bằng mật khẩu), định dạng file đầu ra chi tiết hơn, benchmark hiệu năng các thuật toán khác nhau, giao diện CLI nâng cao với nhiều tùy chọn.