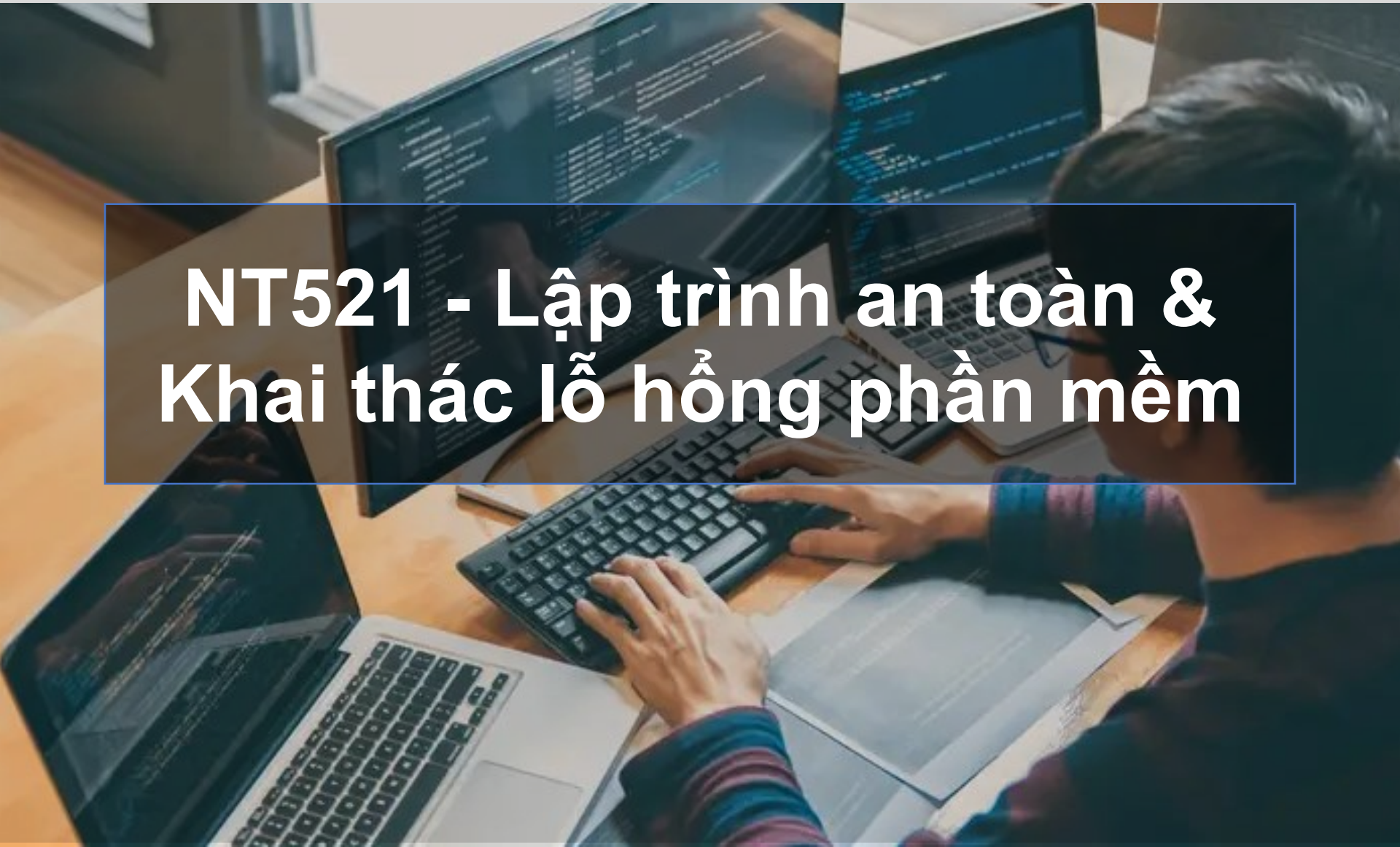
Trường ĐH CNTT – ĐHQG TP. HCM



NT521 - Lập trình an toàn & Khai thác lỗ hổng phần mềm

Off-by-one & Return-to-lib-c

Nội dung

- Tấn công Off-by-one
- Tấn công Return-to-lib-c

Tổng quan

- Khai thác lỗ hổng **tràn bộ đệm - buffer overflow**:
  - Trường hợp lý tưởng:
    - Có khả năng kiểm soát return addr: không dùng stack canary, có thể ghi đè đến vị trí return addr
    - Có thể truyền shellcode vào stack và thực thi: -z execstack
  - Các trường hợp ràng buộc hơn?
    - **Off-by-one**
      - *Giới hạn kích thước buffer có thể bị ghi đè, không đủ để ghi đè return addr*
    - **Return-to-libc**
      - *Shellcode cần viết quá phức tạp hoặc quá dài, stack không cho phép thực thi code*
    - **ROP (Return Oriented Programming)**
      - *Stack không cho phép thực thi*

Tấn công Off-by-one

Tấn công Off-by-one: Vì sao?

- Tấn công **tràn bộ đệm trên stack** đôi khi không cho phép ghi đè trực tiếp địa chỉ trả về.

```
void bar(){
    char buf[256];
    int i;

    for(i = 0; i <= 256; i++)
        buf[i] = getchar();

    // other statements…
}

void foo(){
    bar();
}
```

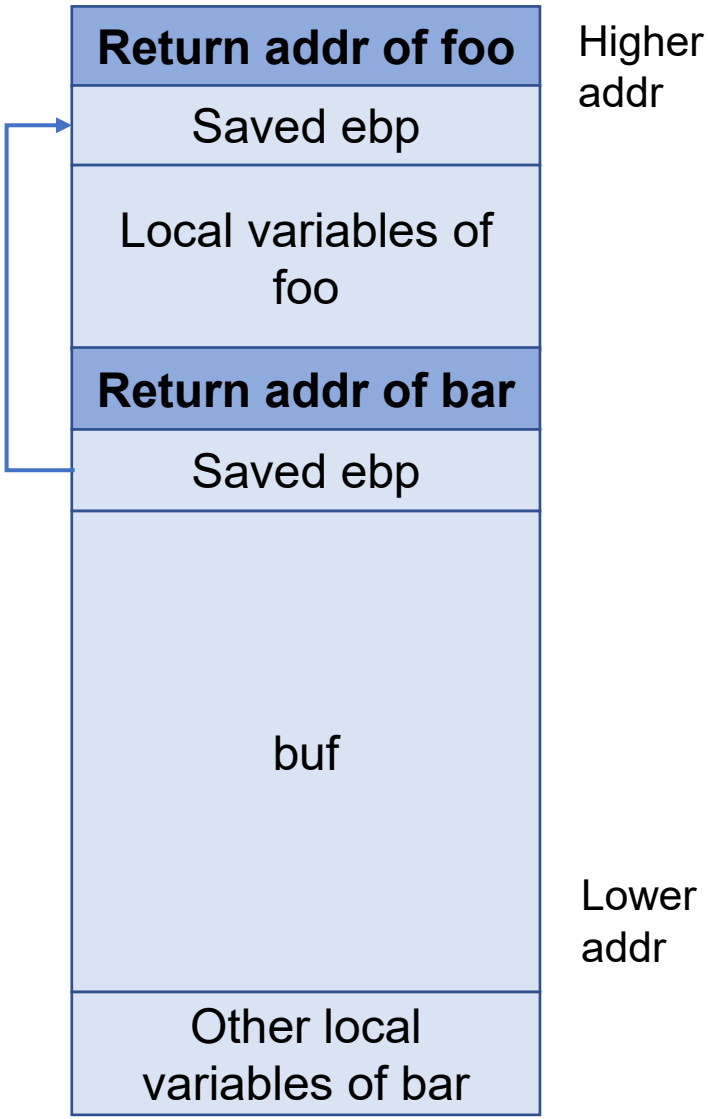
Có vấn đề gì với đoạn code trên?

Dùng <= thay vì < → có thể nhập kí tự thứ 257, nhưng chưa đủ để ghi đè được return addr

→ Cần tấn công kiểu gián tiếp

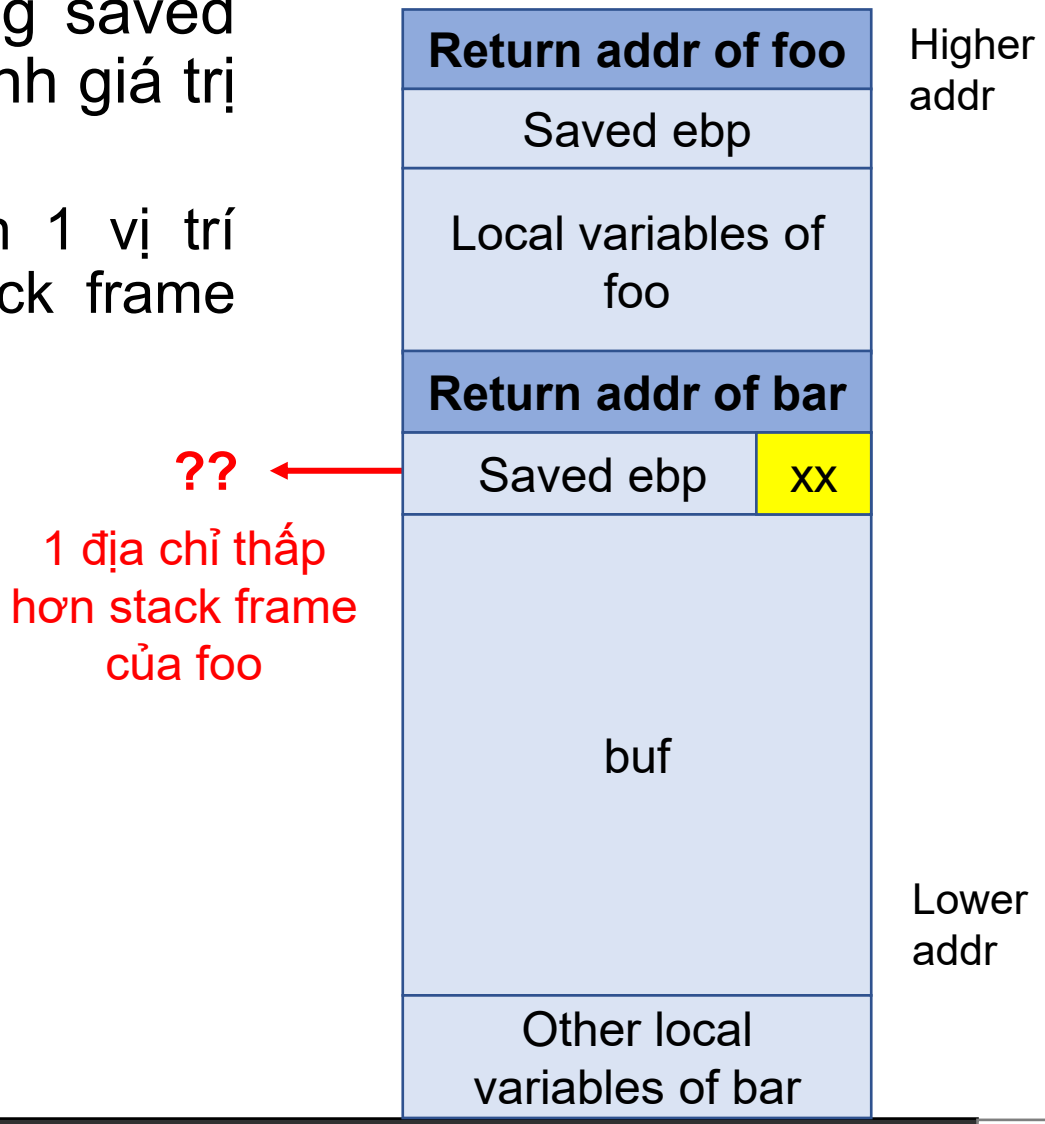
Tấn công Off-by-one: Ví dụ

- Luồng thực thi **foo() → bar()**
- **buf** được thiết kế có độ dài **256 bytes – 256 ký tự.**
- Giả sử buf nằm liền kề vị trí lưu saved ebp.
- Nếu có thêm 1 ký tự được ghi thêm trong stack?
  - Saved ebp của bar sẽ thay đổi
- Nếu saved ebp của bar thay đổi?
  - Khi thực thi xong bar, foo sẽ không tìm được đúng stack frame của mình → foo không tìm được biến cục bộ và **return addr**



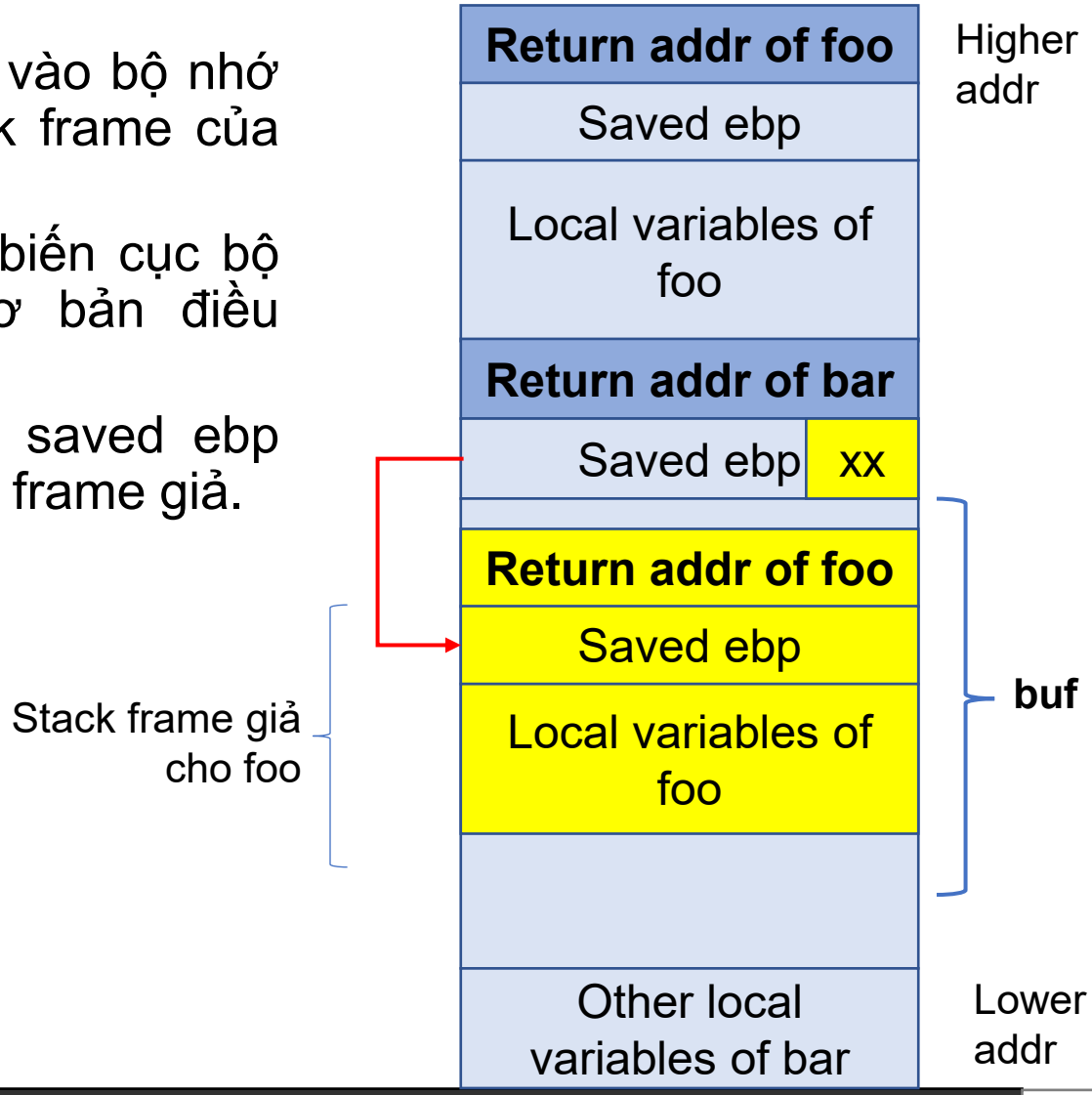
Tấn công Off-by-one: Ví dụ (2)

- Nếu byte thấp nhất trong saved ebp của bar thay đổi thành giá trị **nhỏ hơn,** ví dụ ***0x00***?
  - Saved ebp sẽ trỏ đến 1 vị trí nào đó **thấp hơn** stack frame gốc của foo



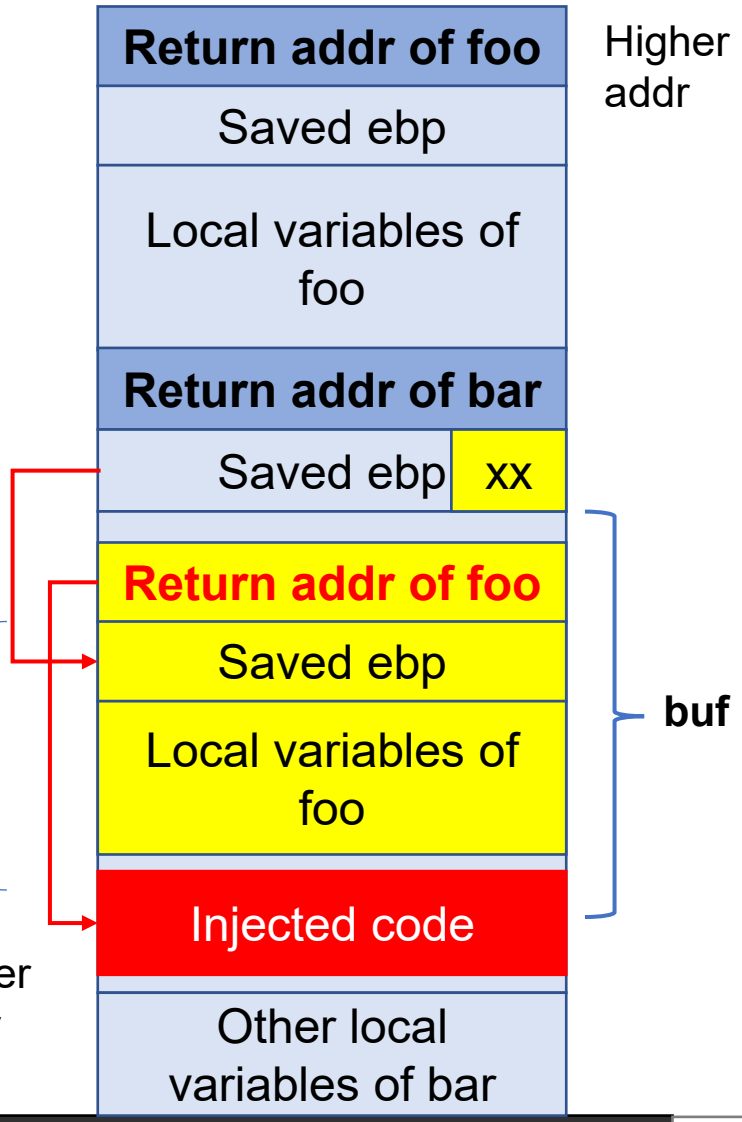
Tấn công Off-by-one: Ví dụ (3)

- Ý tưởng:
  - Ghi 1 stack frame **giả** vào bộ nhớ đệm để giả mạo stack frame của foo.
  - Điều khiển được các biến cục bộ của foo xem như cơ bản điều khiển được foo.
  - Ghi đè byte cuối của saved ebp của bar để trỏ về stack frame giả.



Tấn công Off-by-one: Ví dụ (4)

- Thậm chí tốt hơn nếu attacker có 1 return addr có chủ đích bên cạnh stack frame giả.
  - Có thể điều hướng sau khi thực thi xong foo.
- Có thể khai thác như thế nào?
  - Inject code
  - Gán return address trong stack frame giả của foo thành địa chỉ đoạn code muốn thực thi.



Tấn công Off-by-one: Ví dụ (5)

- Kết luận về ví dụ:
  - Lỗi lập trình trong việc xác định giới hạn
    - Dùng <= thay cho < → gán giá trị cho ô nhớ nằm ngoài mảng
  - Nếu buffer nằm gần saved ebp, tấn công Off-by-one có thể cho phép ghi đè saved ebp.
  - Kẻ tấn công sau đó có thể thay đổi saved ebp để trỏ đến 1 stack frame giả.
  - Với stack frame giả, kẻ tấn công có thể kiểm soát return addr, từ đó kiểm soát luồng thực thi.

Tấn công Off-by-one: Tổng kết

- Một số trường hợp truy xuất con trỏ không chính xác có thể cho phép ghi đè trực tiếp hoặc gián tiếp các cấu trúc/dữ liệu quan trọng, bao gồm return addr.
- Các dạng tấn công vào các lỗ hổng này rất khó ngăn chặn.

Một ví dụ khác có thể bị tấn công off-by-one:

```
void receive(int socket) {
    char buf[MAX];
    int nbytes = recv(socket, buf, sizeof(buf), 0);
    buf[nbytes] = '\0';
    ...
}
```

Tấn công Return-to-libc

Tấn công Return-to-lib-c: Vì sao?

- Tên gọi khác: Tấn công **Arc-injection**
- Ngữ cảnh: việc **truyền code thực thi** (shellcode) đôi khi **khó khăn** trong khai thác lỗ hổng buffer overflow.
  - Shellcode quá dài không để được trong buffer.
  - Shellcode quá phức tạp
  - Trên stack không cho phép thực thi code

→ Tận dụng ***code có sẵn*** trong chương trình, hay cụ thể là trong các **thư viện C (libc)**

Libc cung cấp các macro, các định nghĩa kiểu dữ liệu và các hàm cho các tác vụ phổ biến như: xử lý chuỗi, tính toán toán học, xử lý input/output, cấp phát bộ nhớ,…
Ví dụ: printf(), scanf(), system(),…

- Mục tiêu: Ghi đè **return address** để điều hướng luồng chương trình đến các đoạn code có sẵn trong bộ nhớ.

Tấn công Return-to-lib-c: Ví dụ

- Cho đoạn chương trình ví dụ:

```
int main(int argc, char *argv[]){
    char buf[4];
    strcpy(buf, argv[1]);
    return 0;
}
```

Có vấn đề gì với đoạn code trên?

*Dùng strcpy là 1 hàm sao chép chuỗi nhưng **không có cơ chế kiểm tra** sự phù hợp độ dài giữa nguồn (argv[1]) và đích (buf).*

- **Mục tiêu:** Khai thác lỗ hổng buffer overflow để **mở shell tương tác** mà **không truyền code thực thi**.
  - Giả sử biên dịch với option **-z noexestack** để ngăn thực thi trên code

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
```

Tấn công Return-to-lib-c: Ví dụ (2)

- Cho đoạn chương trình ví dụ:

```
int main(int argc, char *argv[]){
    char buf[4];
    strcpy(buf, argv[1]);
    return 0;
}
```

Ý tưởng

- Có thể mở shell bằng cách gọi hàm `system("/bin/bash")`.
  - system() là 1 hàm libc.
  - Tham số của hàm là command muốn thực thi

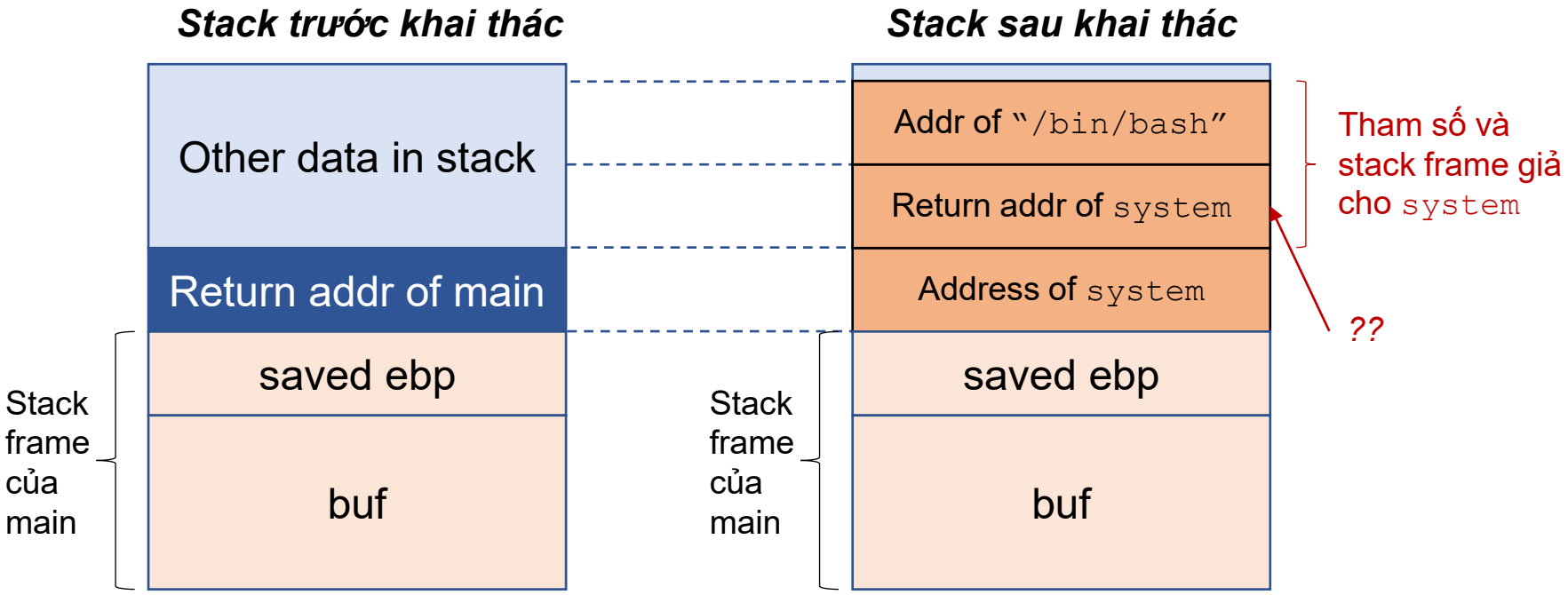
Như thế nào?

- Đổi return address thành địa chỉ của hàm `system`
- Gán tham số đầu tiên thanh địa chỉ chuỗi `"/bin/bash"`

Tấn công Return-to-lib-c: Ví dụ (2)

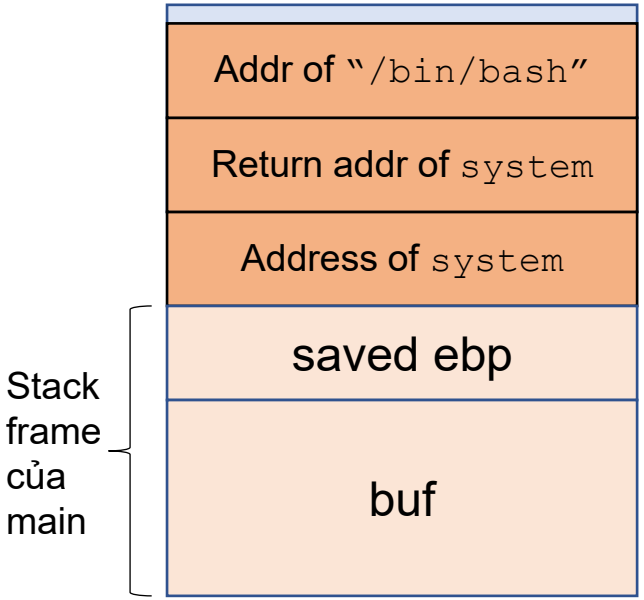
- Cho đoạn chương trình ví dụ:

```
int main(int argc, char *argv[]){
    char buf[4];
    strcpy(buf, argv[1]);
    return 0;
}
```



Tấn công Return-to-lib-c: Ví dụ (3)

- Địa chỉ của hàm system?
  - Debug chương trình với gdb và gõ lệnh **print system**
- Địa chỉ chuỗi "/bin/bash"?
  - Thường có biến môi trường SHELL=/bin/bash
    - gdb: **`x/s *((char **)environ+i)`**
    - *i là thứ tự của biến cần đọc trong danh sách các biến môi trường*
  - Tự chèn thêm biến môi trường với lệnh export
    - Đọc tương tự với gdb



Return-to-lib-c: Gọi nhiều hàm?

- Để thực thi chức năng phức tạp, có thể cần **gọi nhiều hàm libc**

→ Cần "tạo" stack frame cho mỗi hàm được gọi và sử dụng **return addr** để **nối** thành chuỗi gọi hàm

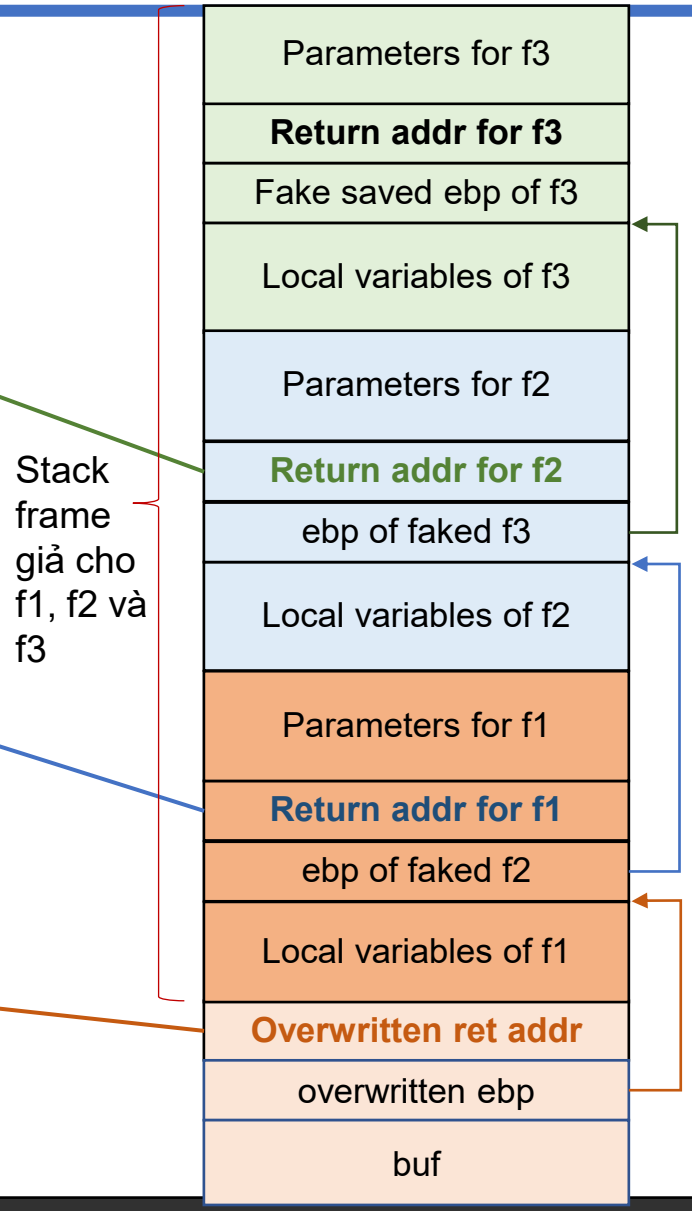
- Giả sử muốn lần lượt thực thi các hàm **f1, f2, f3**
  - Địa chỉ của stack frame giả: **f1 < f2 < f3**.
  - **Return addr của f1** là địa chỉ của **f2**, **return addr của f2** là địa chỉ của **f3** → để thực thi luồng f1 → f2 → f3.
  - Lưu ý: cần gán **return addr** để **bỏ qua phần code set-up stack** của mỗi hàm muốn gọi → để không ảnh hưởng đến stack frame giả.
  - **Saved ebp của f1** cần trỏ về **vị trí ebp** trong stack frame giả của **f2**, tương tự cho **saved ebp của f2** trỏ về vị trí **ebp** trong stack frame giả của **f3**.
  - Nội dung trong stack frame giả phụ thuộc vào hoạt động của hàm **f1, f2, f3**.

Return-to-lib-c: Gọi nhiều hàm? (2)

```
f3:
     <set-up stack code>
     instruction z
     ...
     leave
     ret
```

```
f2:
     <set-up stack code>
     instruction y
     ...
     leave
     ret
```

```
f1:
     <set-up stack code>
     instruction x
     ...
     leave
     ret
```



Biện pháp

- Stack randomization
- Library randomization
- Sử dụng stack canary
- Một số tools: StackGuard, StackShield

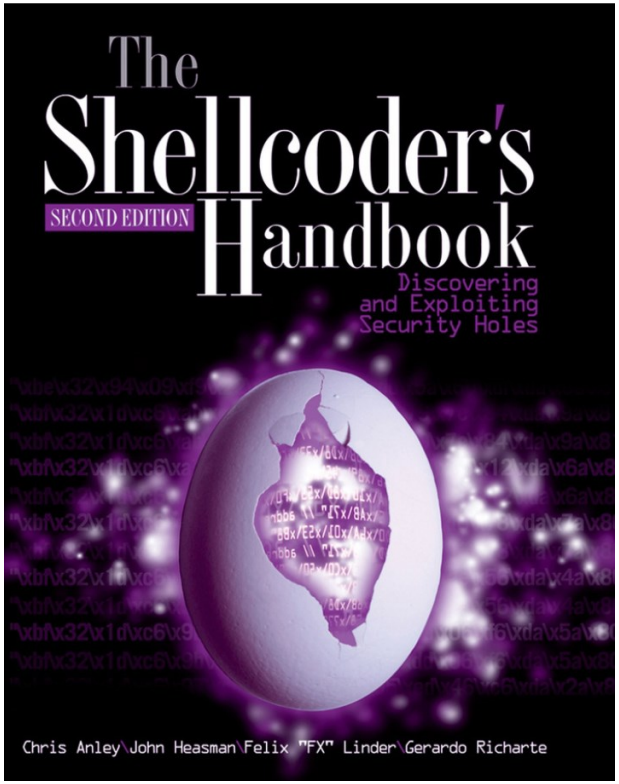
Assignment 4

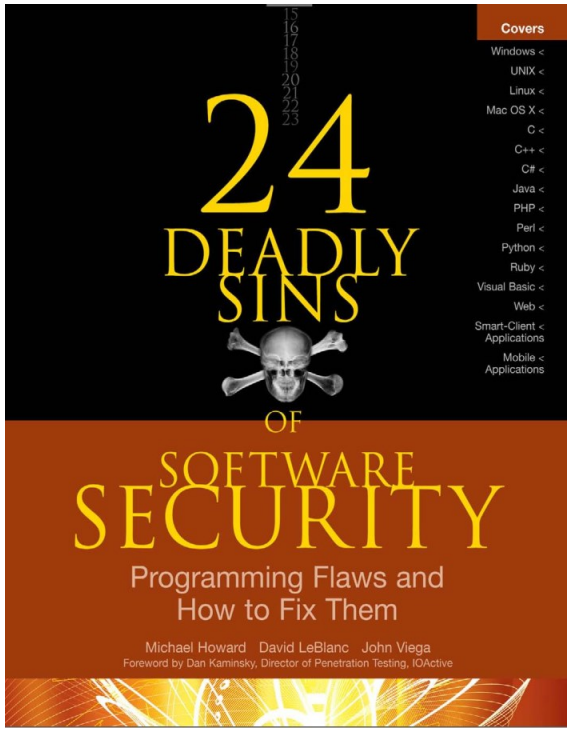
- *Làm **cá nhân***
- *Xem 2 file hướng dẫn chi tiết*
- **4.1 Off-by-one**
  - Lưu ý: máy 64 bit thì compile code thêm option **-m32**.
  - Hiểu cơ chế hoạt động của tấn công.
  - Lý giải được lỗi **segmentation fault** của chương trình.
  - Nộp: hình ảnh chụp giá trị **saved ebp** trên stack của **cpy** hoặc giá trị ebp khi đã quay về main **sau khi nhập chuỗi buf.**
- **4.2 Return-to-lib-c**
  - Thực hành khai thác file
  - Giải thích payload (các giá trị, địa chỉ, padding)
  - Kết quả khai thác

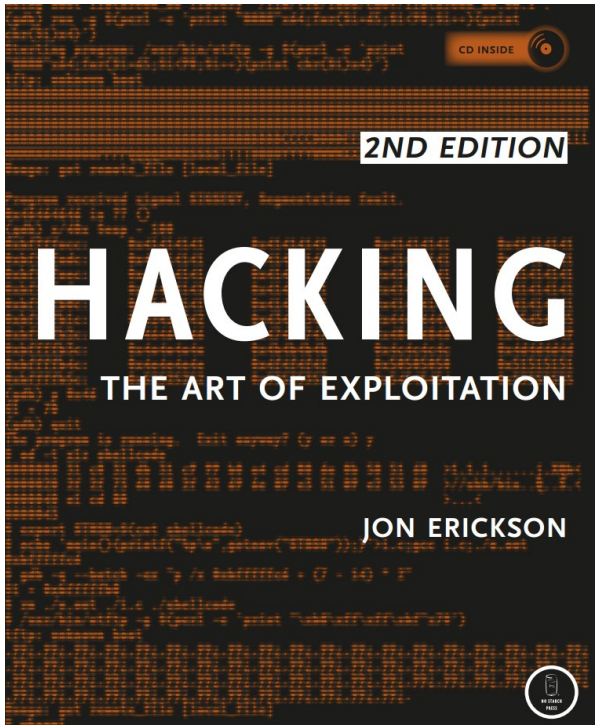
Tài liệu tham khảo: Pwn CTF

- CTF Wiki: https://ctf-wiki.org/pwn/linux/user-mode/environment/
- Modern Binary Exploitation - CSCI 4968: https://github.com/RPISEC/MBE
- NTU Computer Security Fall 2019: https://github.com/yuawn/NTU-Computer-Security
- Nightmare: https://guyinatuxedo.github.io/index.html

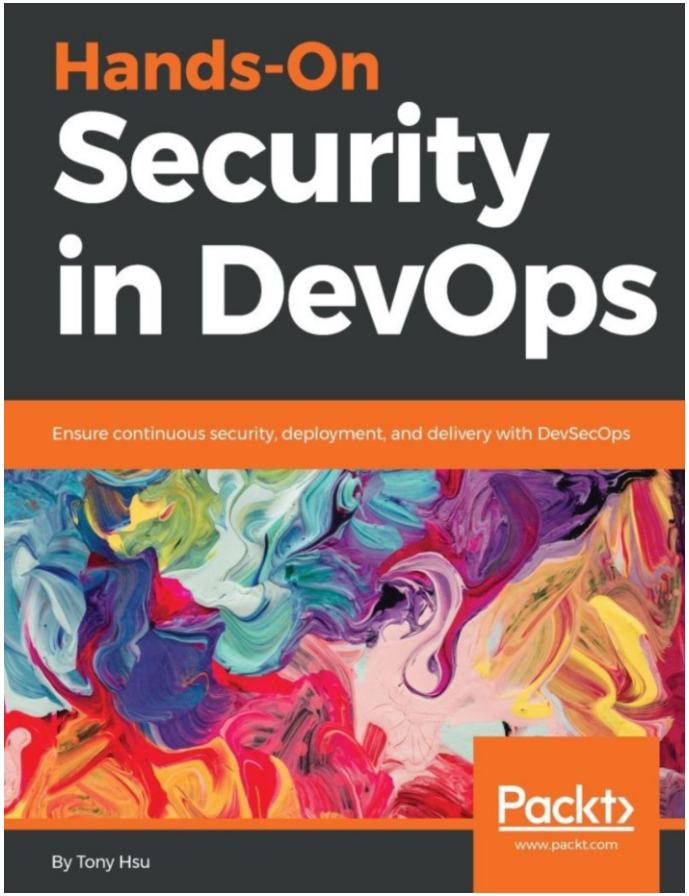
Tài liệu tham khảo

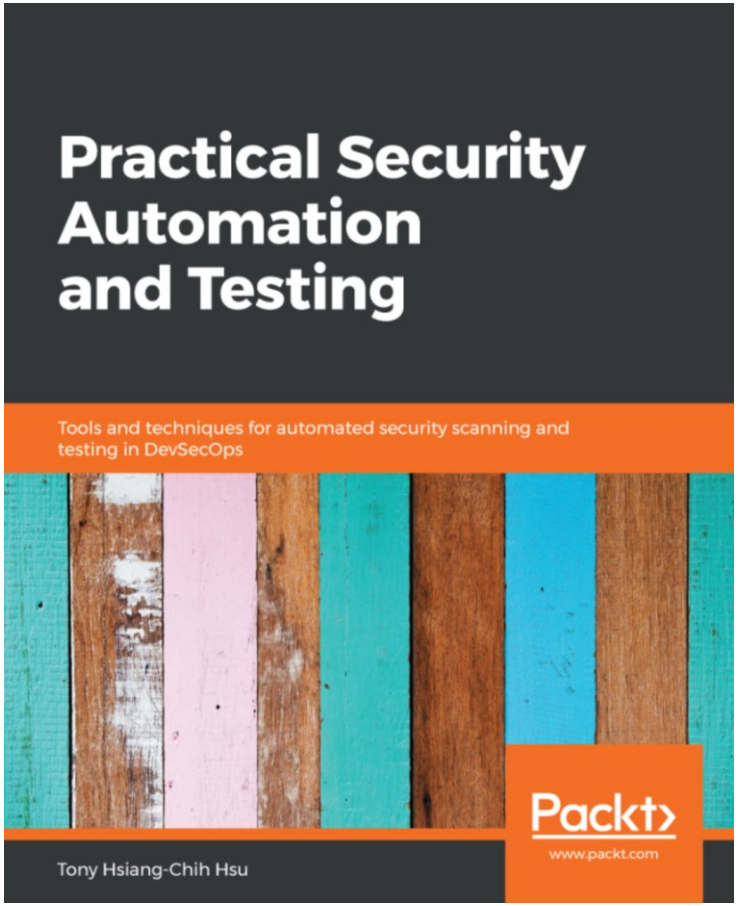




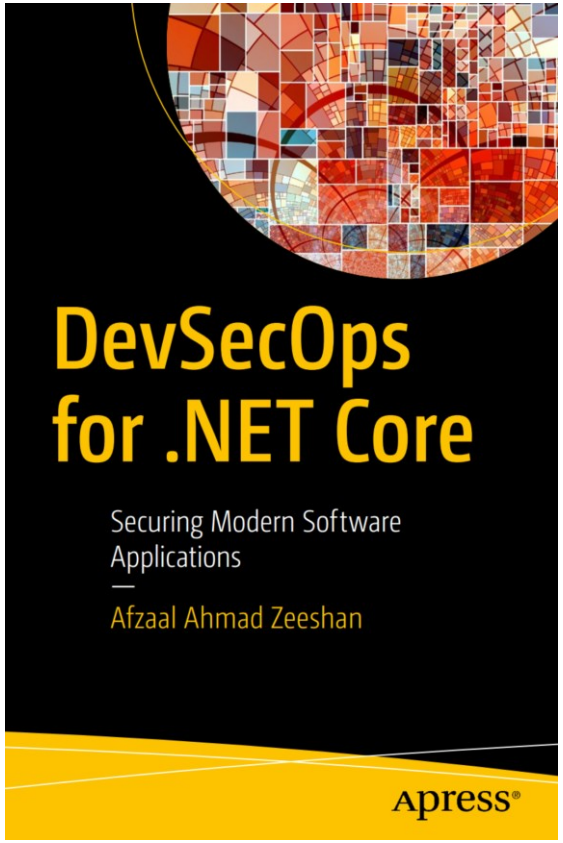


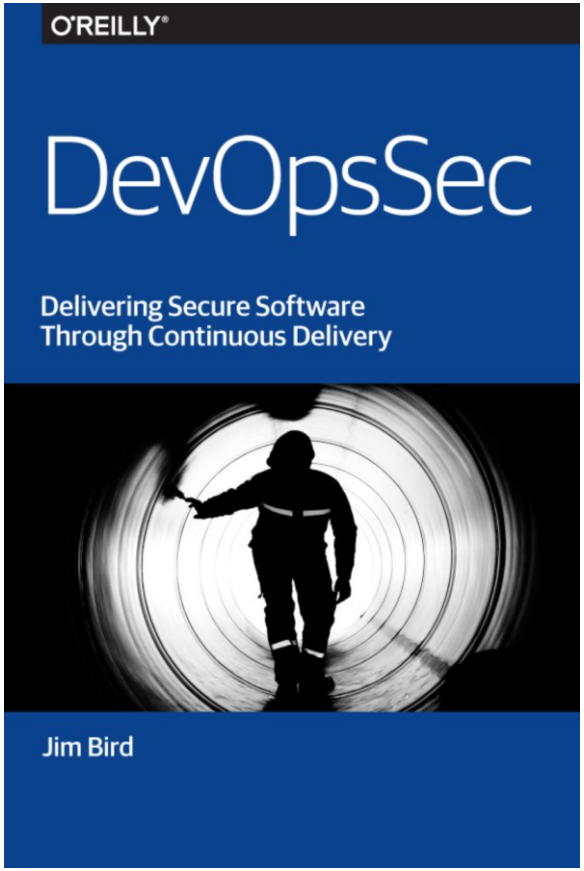
Tài liệu tham khảo



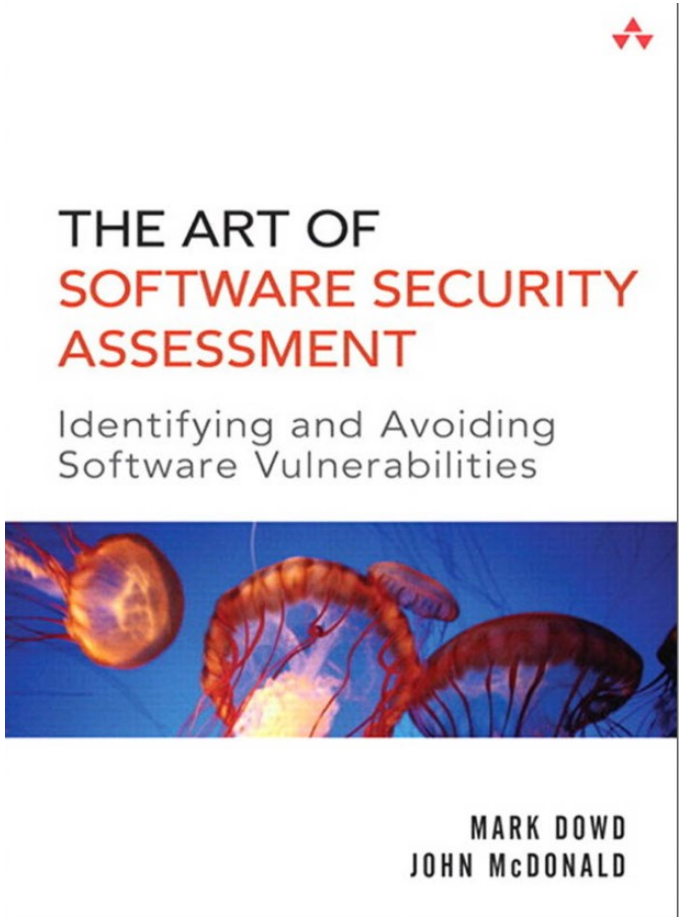


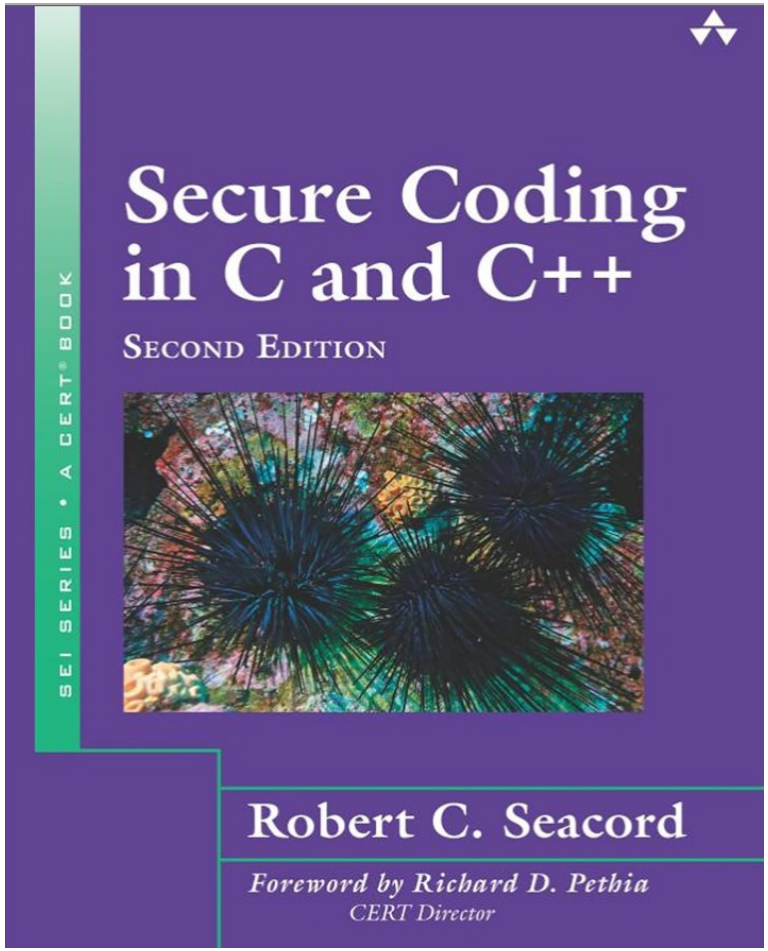
Tài liệu tham khảo



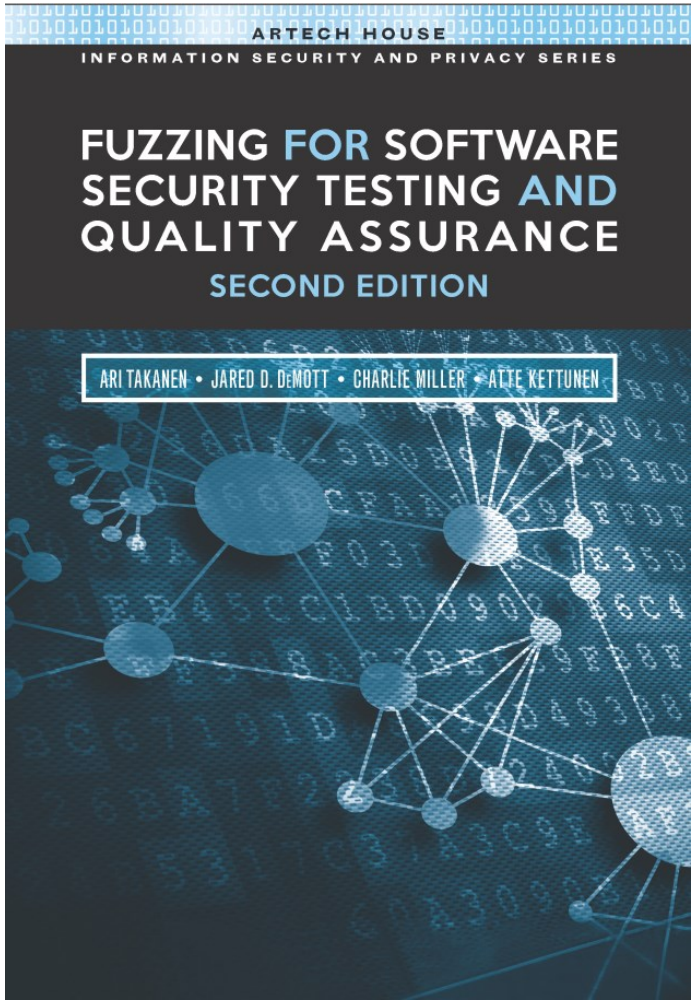


Tài liệu tham khảo

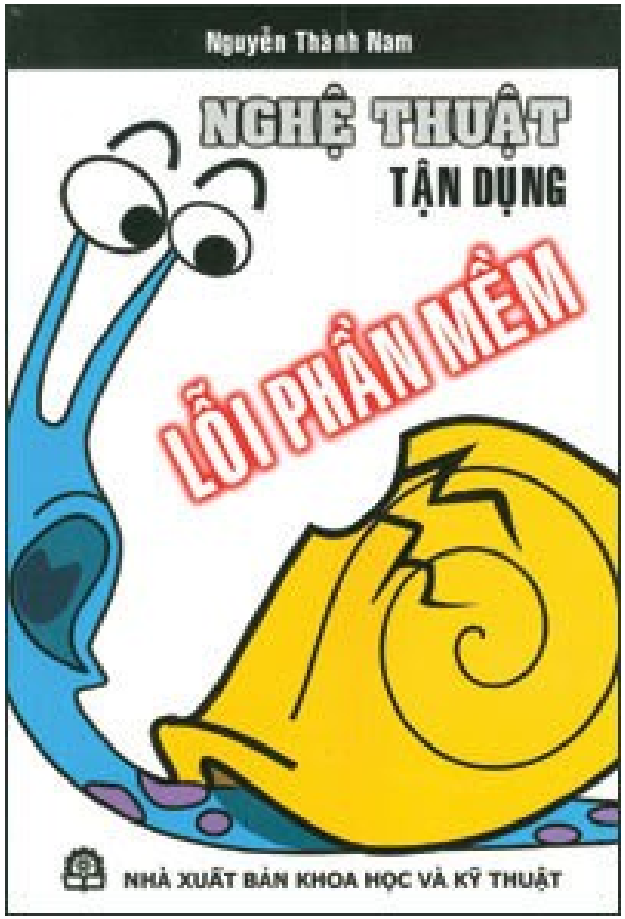




Tài liệu tham khảo



Tài liệu tham khảo



Tài liệu tham khảo

- https://security.berkeley.edu/secure-coding-practice-guidelines
- https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices
- https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf
- https://www.softwaretestinghelp.com/guidelines-for-secure-coding/
- http://security.cs.rpi.edu/courses/binexp-spring2015/
- https://www.ired.team/

Lập trình an toàn & Khai thác lỗ hổng phần mềm

Trường ĐH CNTT TP. HCM