Các kịch bản lỗ hổng khác (Beyond Buffer Overflow)

1. Format String Vulnerability

- **Nguyên nhân**: dùng `printf(user_input)` thay vì `printf("%s", user_input)`.
- Attacker chèn `%x` để leak stack, `%n` để ghi số byte đã in vào địa chỉ tùy ý.
- **Khai thác**:
    - Leak password/key từ stack.
    - Overwrite GOT/DTORS với `%n` ⇒ redirect code execution.

Ví dụ:
```
printf(argv[1]); // input: "%x %x %x %n"
```

2. Integer Overflow / Signedness Errors

- **Integer Overflow**: số nguyên khi vượt quá max/min bị wrap.
    - Ví dụ: `size = user_input_len * sizeof(struct)` ⇒ nếu tràn ⇒ cấp phát buffer nhỏ nhưng copy nhiều dữ liệu ⇒ overflow.
- **Signedness Error**: signed ↔ unsigned gây nhầm lẫn.
    - `if (len < 128)` pass, nhưng nếu len = -1 ⇒ convert sang unsigned = 4,294,967,295 ⇒ huge copy.

3. Off-by-One Errors

- Chỉ tràn **1 byte** nhưng vẫn nguy hiểm.
- Có thể ghi đè byte thấp của **saved EBP** ⇒ khi `leave; ret` chạy, `ESP` bị attacker điều khiển ⇒ redirect đến shellcode.

4. Uninitialized Variables

- Stack frame không được reset ⇒ biến rác có thể chứa data cũ (attacker-controlled).
- Nếu biến này dùng trong `sprintf` hoặc làm pointer ⇒ attacker có thể điều khiển write location.

Ví dụ:
```
char *msg; // uninitialized
sprintf(buf, msg); // nếu msg chứa addr do attacker inject trên stack ⇒ overflow
```

5. Double Free / Use After Free

- **Double Free**: gọi `free(ptr)` 2 lần ⇒ free list bị hỏng ⇒ attacker kiểm soát pointer trong allocator.

- **Use After Free**: dùng pointer sau khi free ⇒ attacker thay nội dung chunk đó bằng shellcode/data khác.
- Rất phổ biến trong browser, PDF, JS engines.

Thực tế lỗ hổng phần mềm

- **IMAP server**: classic `strcpy`.
- **ProFTP**: attacker điều khiển biến length ⇒ overflow.
- **Apache**: off-by-one trong `memcpy`.
- **Linux Kernel**: integer overflow trong `vmalloc()`.
- **FreeBSD Kernel**: signedness bug trong `copyout`.
- **Snort IDS**: bounds check sai khi reassemble packets ⇒ overflow.

👉 Các bug hiện đại = subtle: **off-by-one, integer overflow, signedness, bounds check errors**.
Strcpy-based overflow gần như tuyệt chủng trong phần mềm lớn, nhưng vẫn còn trong code custom.

Tìm lỗ hổng

1. Closed Source

- **Fuzzing**: gửi dữ liệu random, tìm crash ⇒ candidate exploit.
- **Reverse Engineering**: đọc assembly, phân tích binary.
- **Binary Diffing**: so sánh patch cũ/mới để tìm “silent security fix”.

2. Open Source

- **Manual Inspection**: hiệu quả nhất để tìm subtle bugs.
- **Automated Tools** (splint, Coverity, static analyzers): nhanh nhưng nhiều false positives.

3. Tips

- Tập trung vào **user input path** (điểm vào input → hàm xử lý).
- Đặc biệt chú ý: **loops, bounds checks, integer arithmetic**.
- Đọc comments & bug history (`// FIXME`, “not safe”, “crash here”) ⇒ hint lỗ hổng.

Công nghệ chống khai thác (Exploit Mitigation)

1. DEP / NX Bit

- **Data Execution Prevention**: stack/heap không thực thi được.

- Vẫn bypass được bằng **ret2libc** (return-to-libc).
    - Thay vì inject shellcode, attacker gọi `system("/bin/sh")` từ libc.

2. ASLR (Address Space Layout Randomization)

- Random hóa địa chỉ **stack, heap, libc, mmap**.
- Làm khó đoán địa chỉ return address hoặc GOT entry.
- Nhưng thường không hoàn toàn random ⇒ attacker có thể brute-force hoặc info leak.

3. ASLR + DEP

- Kết hợp chặn nhiều attack.
- Nhưng tồn tại bypass ⇒ **ROP (Return-Oriented Programming)**.
    - Attacker dùng gadget có sẵn trong binary/libc để ghép thành “shellcode không cần inject”.

Key Insight

- Các lỗ hổng ngày nay **không còn obvious** (kiểu `strcpy`) mà toàn subtle: signedness, off-by-one, integer overflow.
- Fuzzing + manual code review = combo mạnh nhất để tìm lỗ hổng.
- DEP + ASLR = khá cứng, nhưng attacker chơi ROP/ret2libc vẫn vượt được.
- **Mọi crash = cơ hội, nhưng không phải mọi crash đều exploit được**.