Introduction to Computer Security Lecture Slides

Application Security

Asst. Prof. Mihai Chiroiu

- “My software never has bugs. It just develops random features.”
- “I have one more bug left”
- “You’re holding it wrong!”

Contents

- Computer Vulnerabilities
  - Cause & classification
  - Memory safety
  - Common mitigations
- State of the Art
  - Eternal War in Memory (paper presentation)

Software – the final frontier

- Access control and crypto are the bricks for building blocks
- Protocols/algorithms used to design useful blocks
- Software implements all of the above

Properties of a vulnerability

- Target application / system component
- Cause
- Severity
- Effect: Remote vs Local:
    - Remote Code Execution (RCE): enter system via network;
    - Local Privilege Escalation: become root!
- Disclosure timeline (previously discovered vs **0-day**)

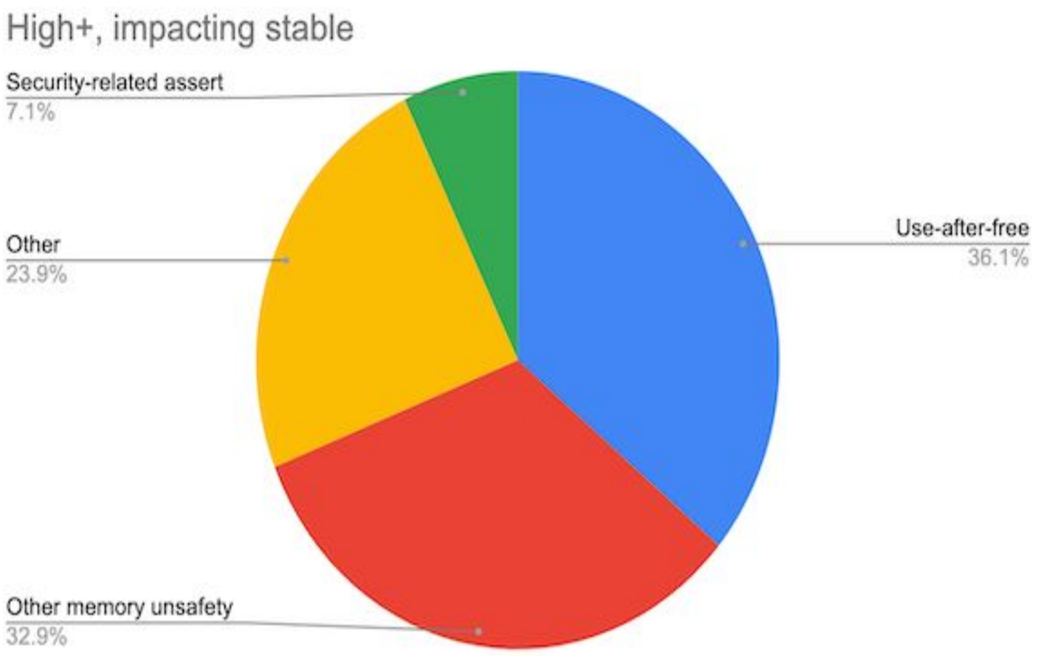
Vulnerability causes

- Access control / business logic bugs
- Code injection
- Input validation (format string attacks, path traversal…)
- **Memory safety**: buffer overflow, dangling pointer, race condition, information leak, use after free etc.
- Side channel attacks
- UI confusion
- And many more!

Memory safety

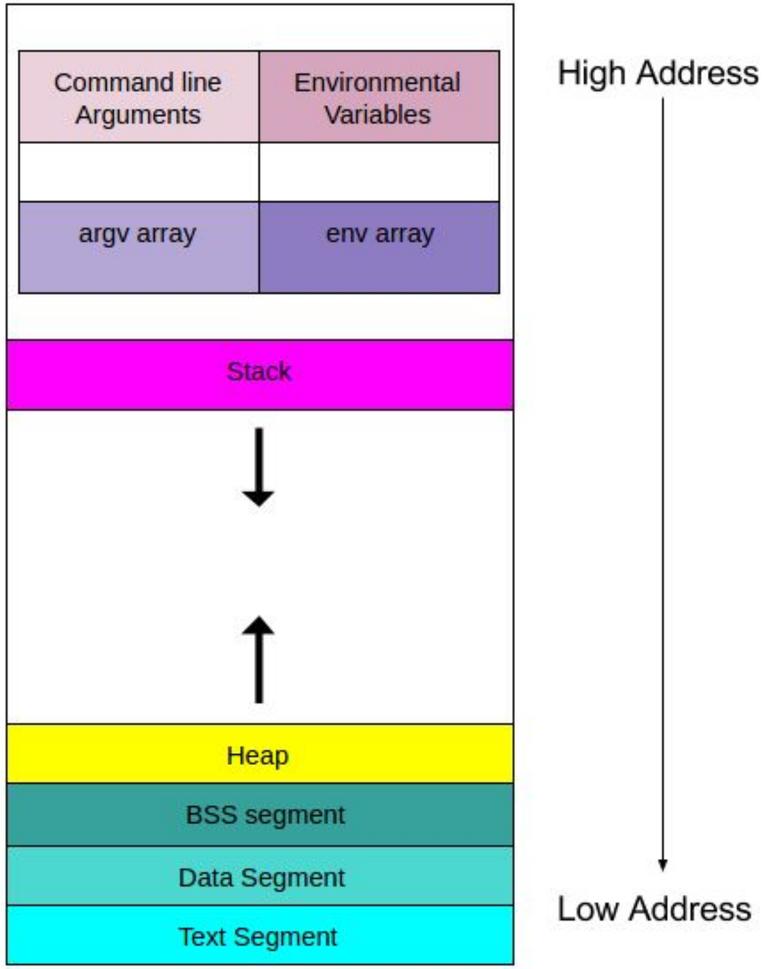
- **Chrome: 70% of all security bugs are memory safety issues**

https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/



Intro: address space

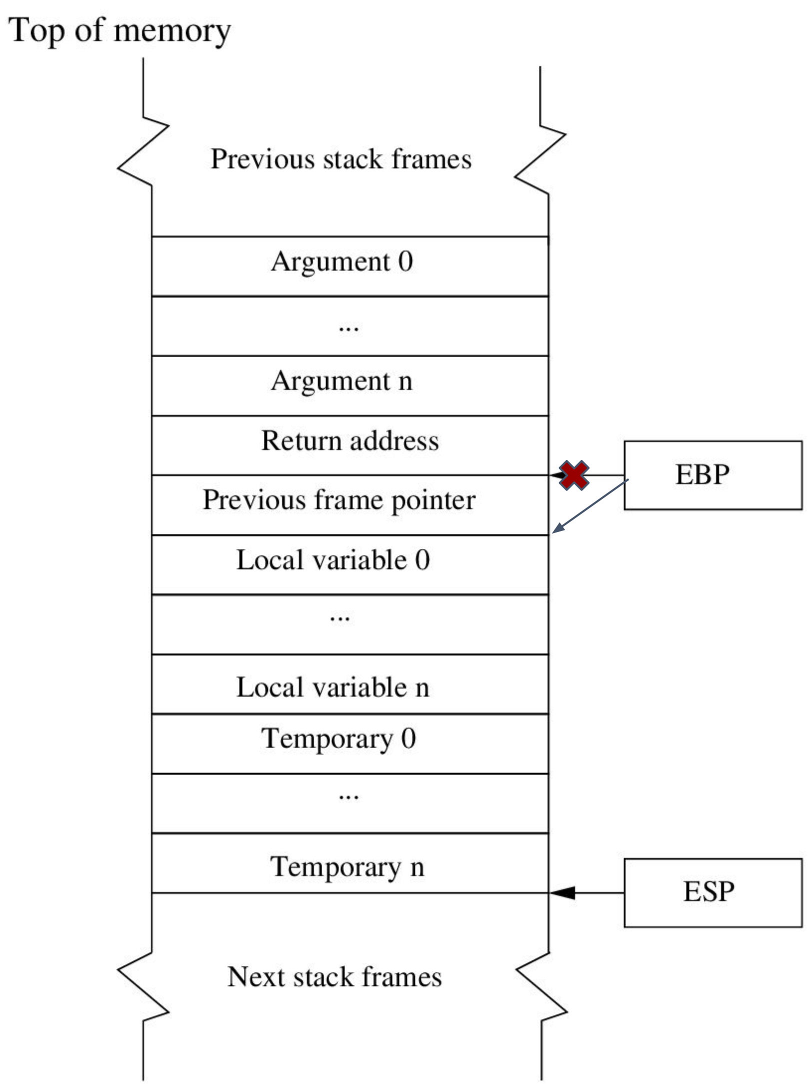
- Userspace processes have virtual memory
- Compiler (linker) + OS decide where each segment goes.
- Address space layout has impact on application's security



Intro: stack frame

- Stack: function arguments, saves CPU state (saved program counter, prev. frame, registers) and local variables:

```c
int f(int x) {
    int n;
    int buf[10];
    // ...
}
int main() {
    f(); // asm call f() <-- saves PC
}
```

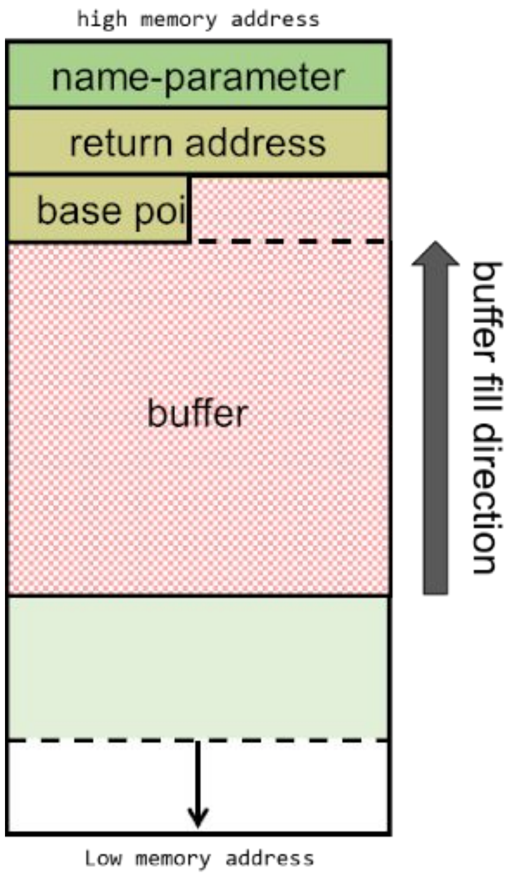


Stack buffer overflow

- Happens when a buffer's is written after its allocated size.

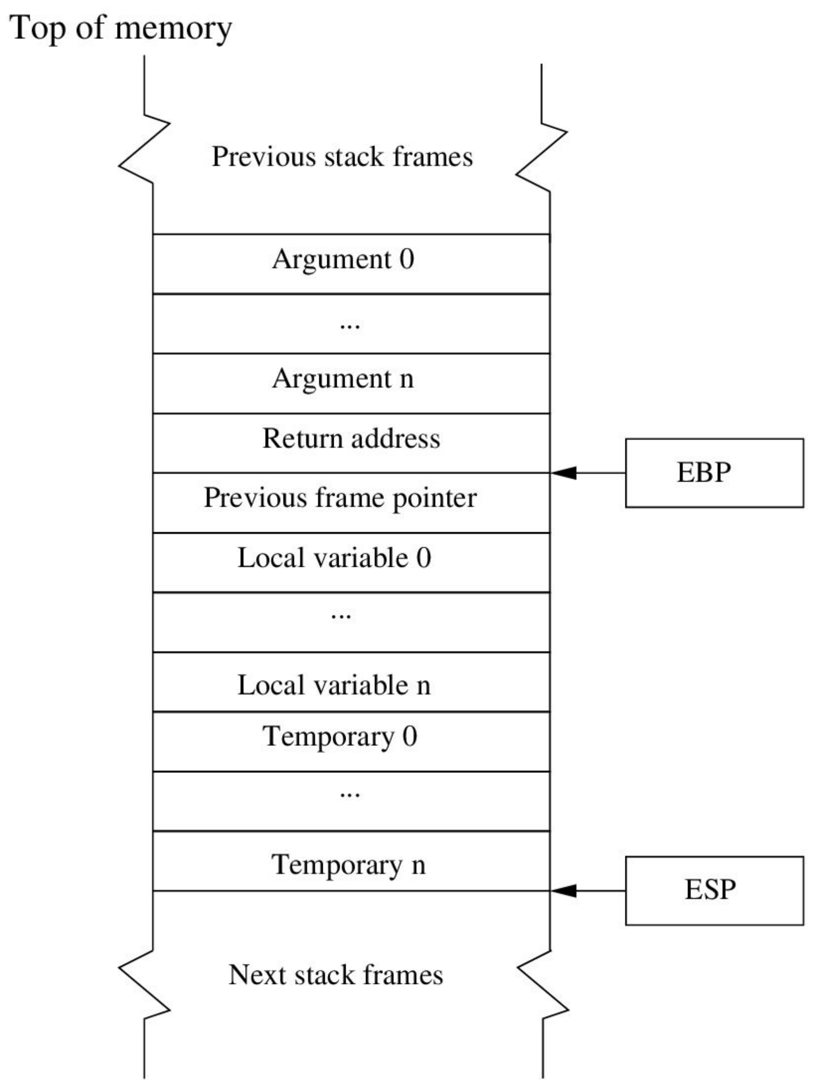
```
char buf[10];
char *input = "This text is larger than expected";

strcpy(buf, input);
```



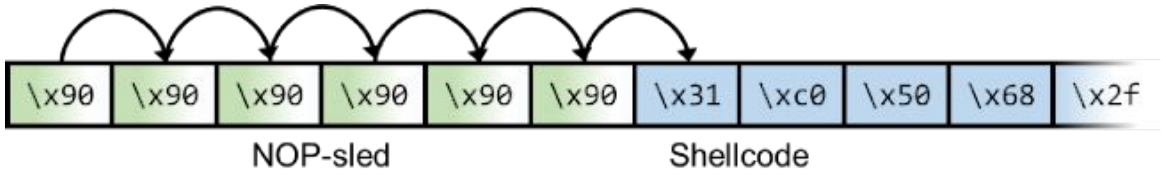
Stack overflow (2)

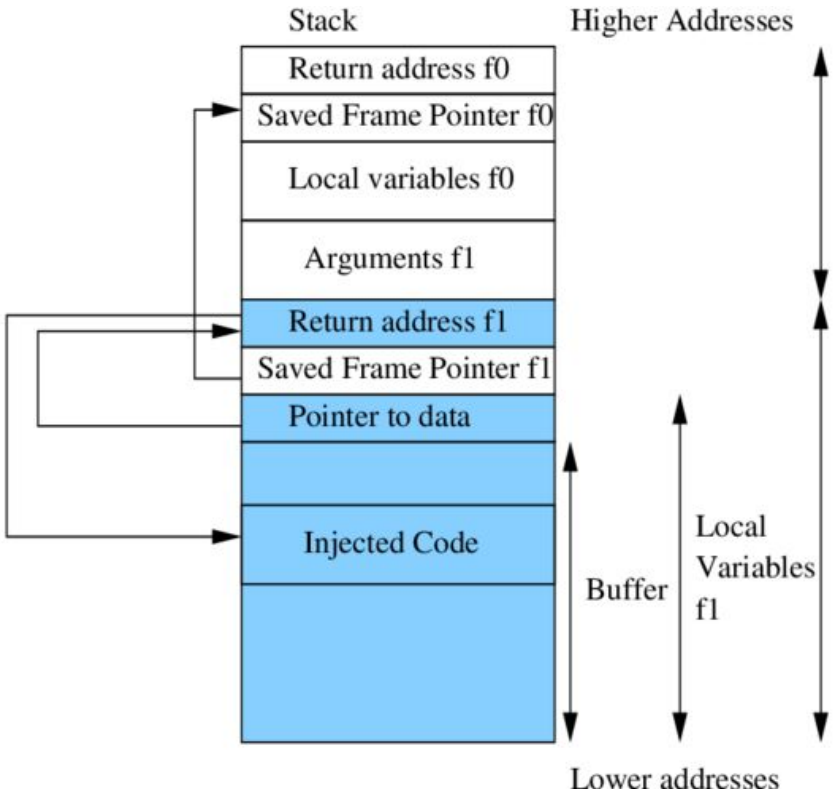
```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```



Stack overflow (3)

- ret instruction will pop the return address from the stack, then jump to it.
- CPU will execute the injected code (shellcode).
- NOP sled when the address is not fixed:





Format string attacks

```
printf("x=%d, y=%d, z=%d", x, y, z)
```

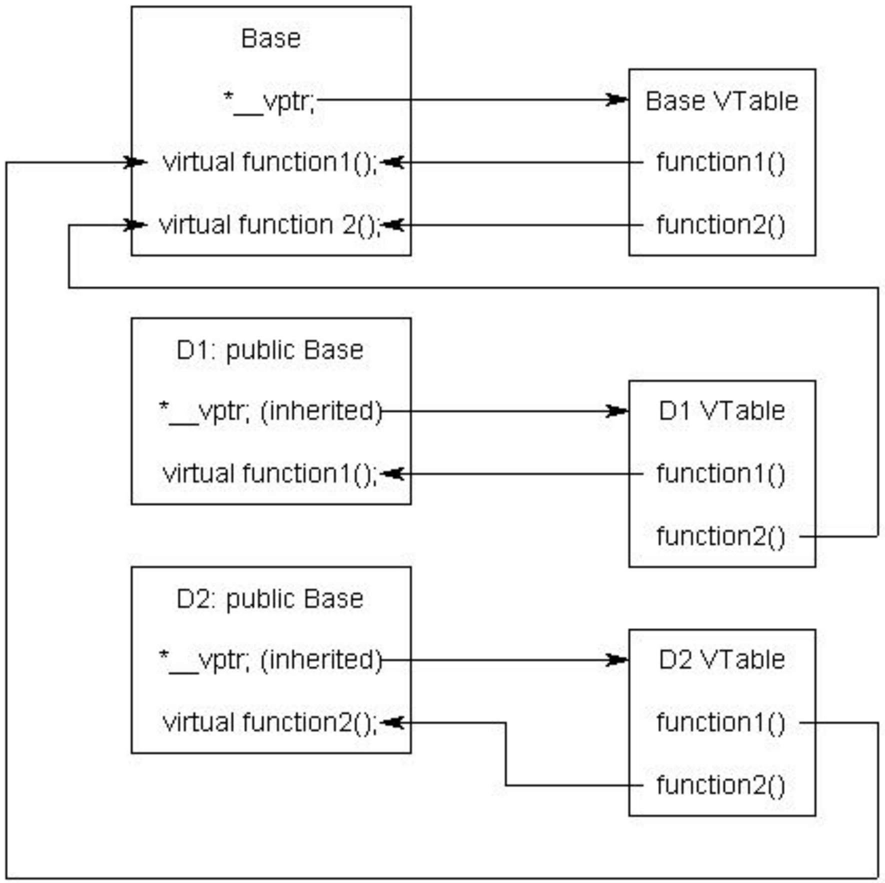
- What if the user controls format string?
  - %s: read from custom memory address
- Read-only vulnerability? Nope…

"%n": consume next argument as address (pointer) and store the number of bytes written so far into it.

z
y
x
“fmt string”
printf: RIP
printf: RBP

What about heap?

- C++ (and other OOP languages) use virtual method tables for implementing polymorphism
- Attacker replaces VTable pointers to controlled memory
- When an object method is called, the function pointer is loaded from the attacker's VTable



What about .data?

```
struct module {
   char private_data[1024];
   (void)(*callback)();
};
struct module enabled_modules;
...
main() {
    struct module *mod = ...;
    // meanwhile: buffer overflow on module->private_data
    mod->callback();
}
```

Use after free

- Free the memory of an object (not needed anymore)
- Next, application allocates new object with attacker-controlled data
- Another section of the application uses the released object (still has an old pointer stored in a variable)
- Are scripting languages safe?
    - ➢ *nope*

```
// Adobe Flash exploit (ActionScript)

ps = PSDK.pSDK;
ps.release();
ms = new MediaResource("jack",
                0x54336677, null);
try{
   ps.createDefaultContentFactory();
} catch (e:Error) { }
```

Just in Time + scripting => bytecode!

- Defeats R⊕X
- JIT Spraying:

```
VAL = (VAL + 0xA8909090)|0;
VAL = (VAL + 0xA8909090)|0;
```

=> just in time compiles it into:

```
00: 05909090A8    ADD EAX, 0xA8909090
05: 05909090A8    ADD EAX, 0xA8909090
```

offset pointer with +1 byte:

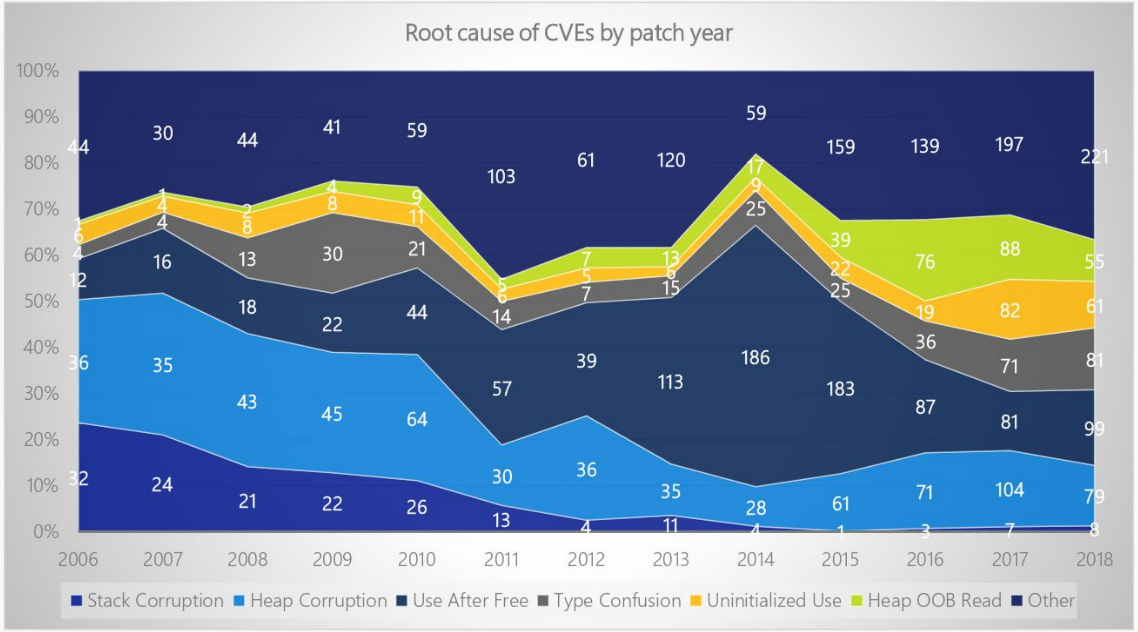
```
03: 90     NOP
04: A805   TEST AL, 05
```

Size checks vs integer overflows

```
#define HEADER_SIZE 128
uint16_t len = read_input_size();
uint8_t *buffer = malloc(len + HEADER_SIZE);
// user gives a valid len = 65535
// malloc allocates just 127 bytes...
...
read_input_into_buffer(buffer, len);
```

***Microsoft:** BlueHatIL - Trends, challenge, and shifts in software vulnerability mitigation*

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016: #1: heap out-of-bounds | #2: use after free | #3: type confusion | #4: uninitialized use

11

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Real World Examples

- EternalBlue - SMB Protocol Vulnerability (CVE-2017-0144)
  https://research.checkpoint.com/2017/eternalblue-everything-know
- Microsoft Exchange RCE Vulnerability (CVE-2021-26857)
  https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers
- Flash Player (CVE-2018-15982)
  https://securityaffairs.co/wordpress/78712/hacking/cve-2018-15982-flash-zero-day.html
- Log4J (CVE-2021-44228):
  https://blog.checkpoint.com/2021/12/11/protecting-against-cve-2021-44228-apache-log4j2-versions-2-14-1/

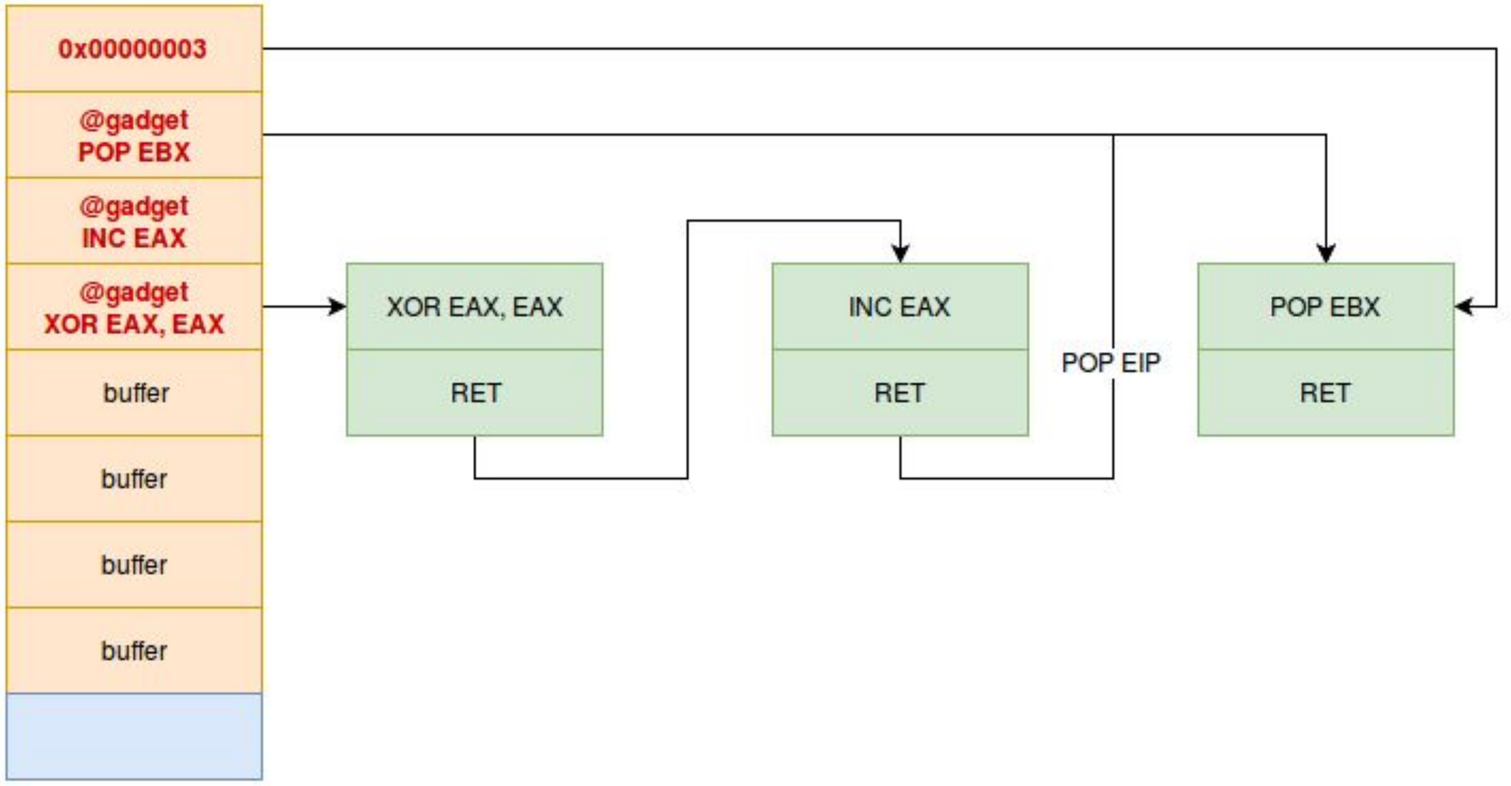
Memory bugs mitigation

- DEP (data execution prevention) / No-execute (NX) bit
  - *defeated by ROP (return oriented programming)*
- Address space layout randomization
  - defeated by memory leaks
- Stack Canaries
  - *defeated by memory leakage, side channels etc.*
- Control flow integrity

SoK: Eternal War in Memory

https://www.ieee-security.org/TC/SP2013/papers/4977a048.pdf

Return oriented programming



Control Flow Integrity

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```