

Introduction to Computer Security Lecture Slides

© 2024 by [Mihai Chiroiu](#) & [Florin Stancu](#)

is licensed under [Attribution-NonCommercial-ShareAlike 4.0
International](#)

Application Security

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

Contents

- Software Vulnerabilities
 - Cause & classification
 - Memory safety bugs + examples
 - Defenses & mitigations

Software – the final frontier

- Access control and crypto are the bricks for building secure blocks
- Protocols/algorithms used to design useful blocks
- Software & hardware implements all of the above
- **Vulnerabilities** – flaws allowing unintended access in a system

Properties of a vulnerability

- Target application / system component
- Cause
- Severity
- Effect: Remote vs Local, e.g.:
 - Remote Code Execution (RCE): enter system via network;
 - Local Privilege Escalation: become root!
- Discovery/exploitation timeline (previously disclosed vs **0-day**)

Vulnerability causes

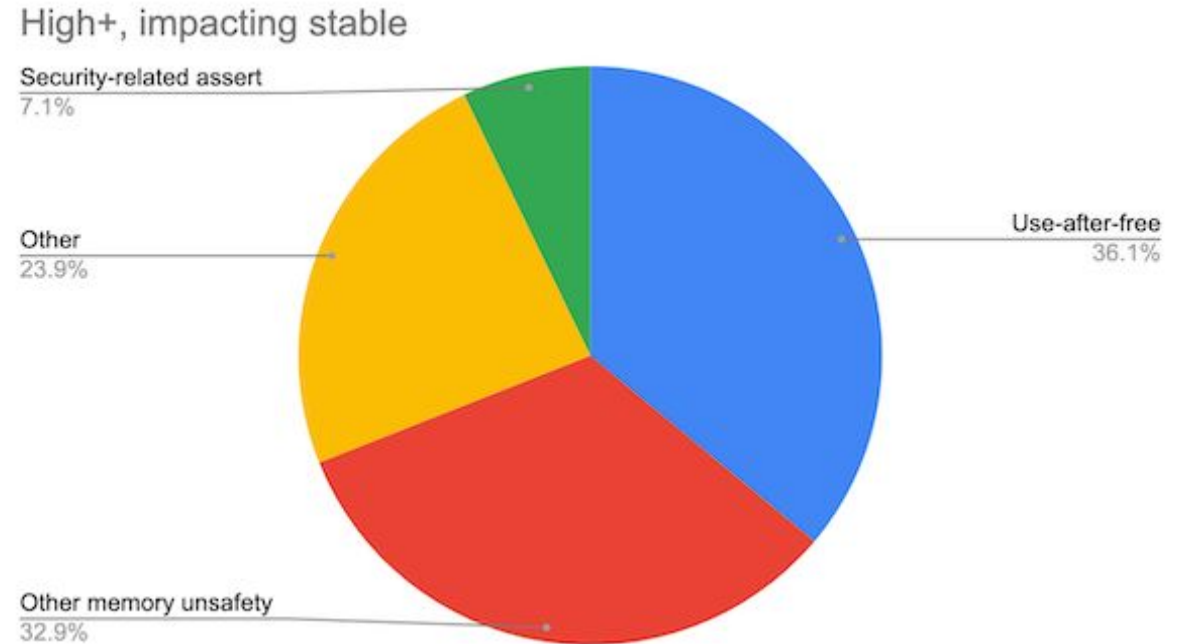
- Access control / business logic bugs
- Code injection
- Input validation (format string attacks, path traversal...)
- **Memory safety**: buffer overflow, dangling pointer, race condition, information leak, use after free etc.
- Weak crypto, side channel attacks...
- UI confusion
- And many more!

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.cloudflare.com/inside-the-log4j2-vulnerability-cve-2021-44228/>

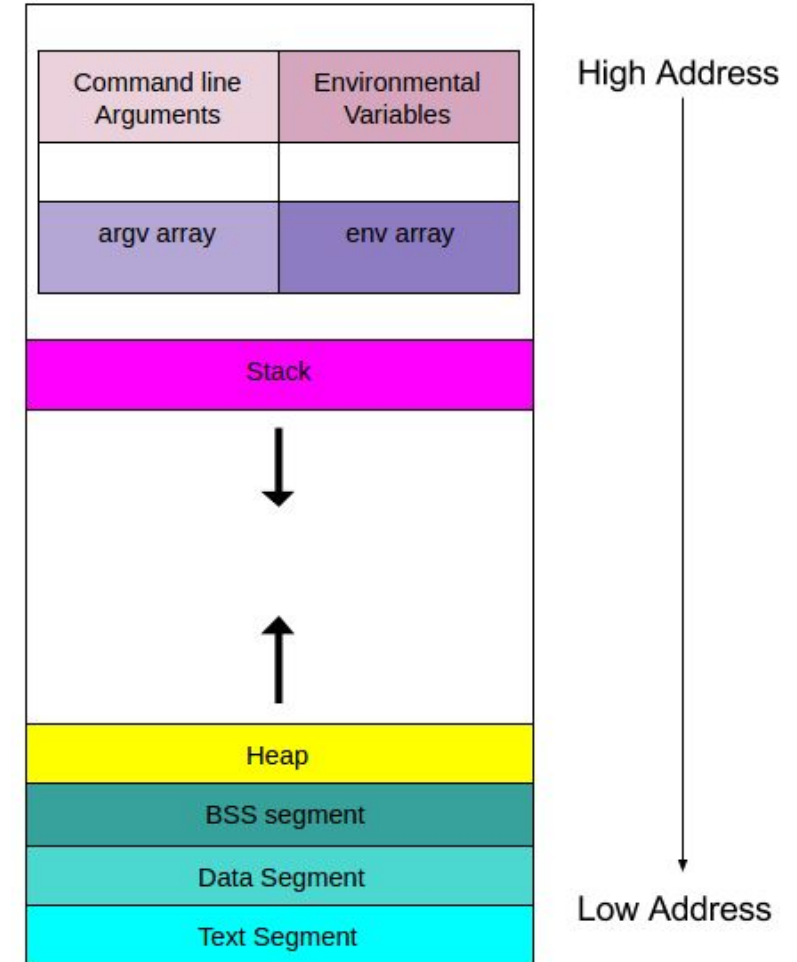
Memory safety

- **Chrome: 70% of all high severity security bugs are memory safety issues [1]**



Intro: address space

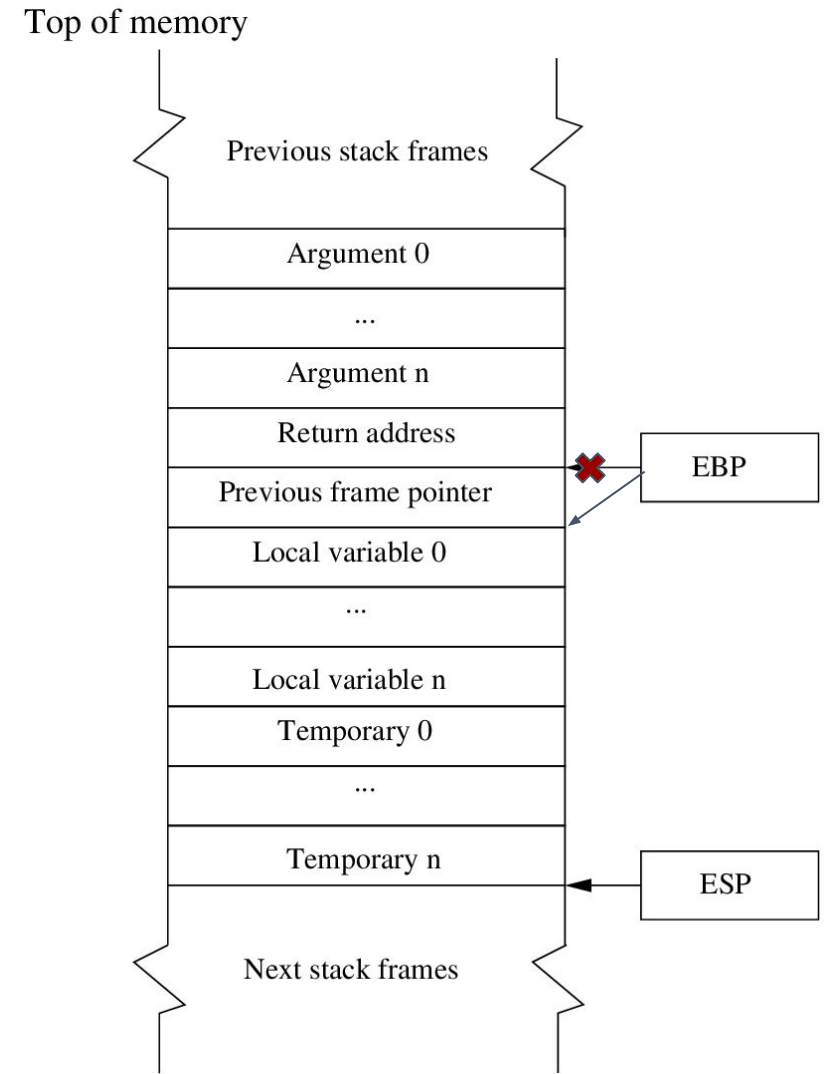
- Userspace processes have virtual memory
- Compiler (linker) / OS decide where each segment goes.
- Address space layout has impact on application's security



Intro: stack frame

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

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

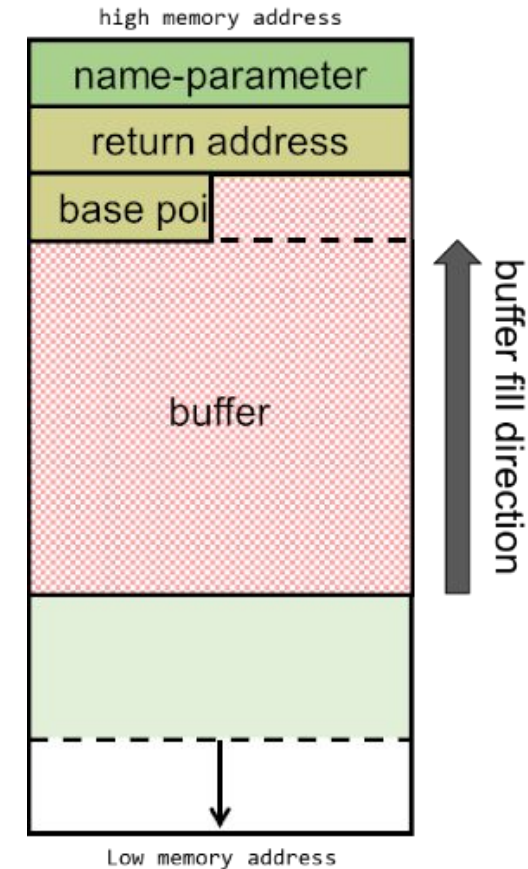


Stack buffer overflow [2]

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

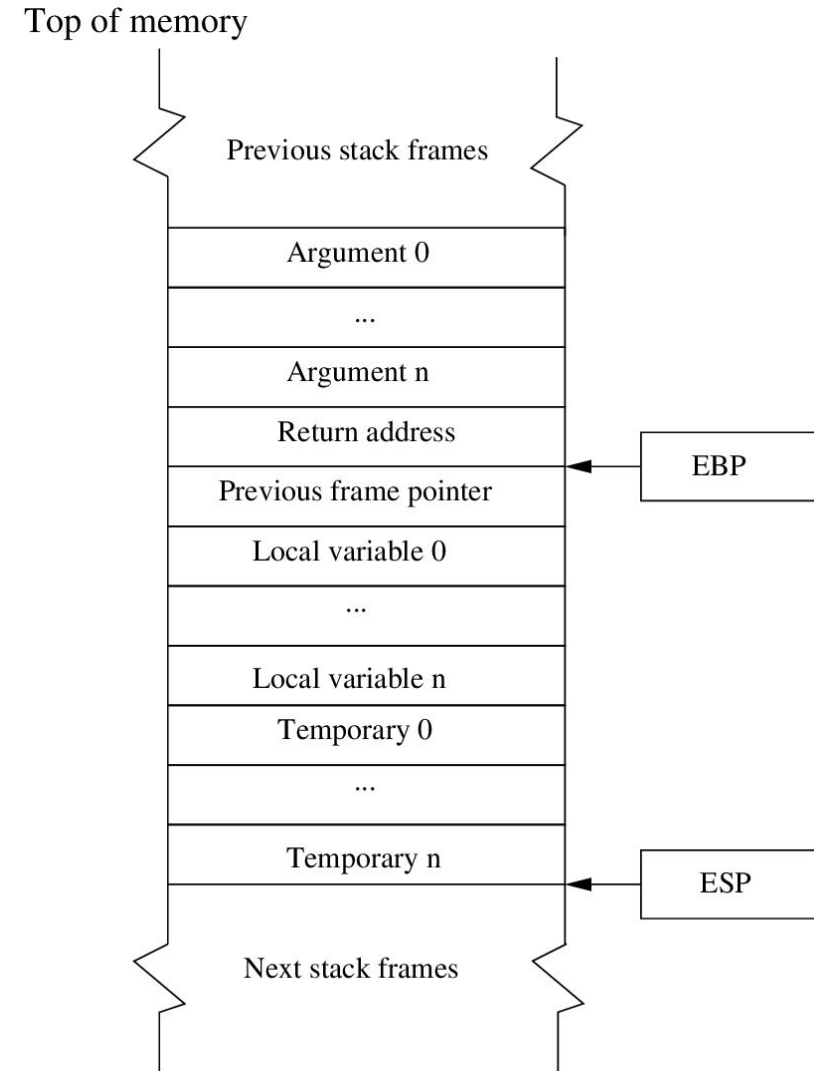
```
char buf[10];  
char *input = "This text is larger than  
expected";
```

```
strcpy(buf, input);
```



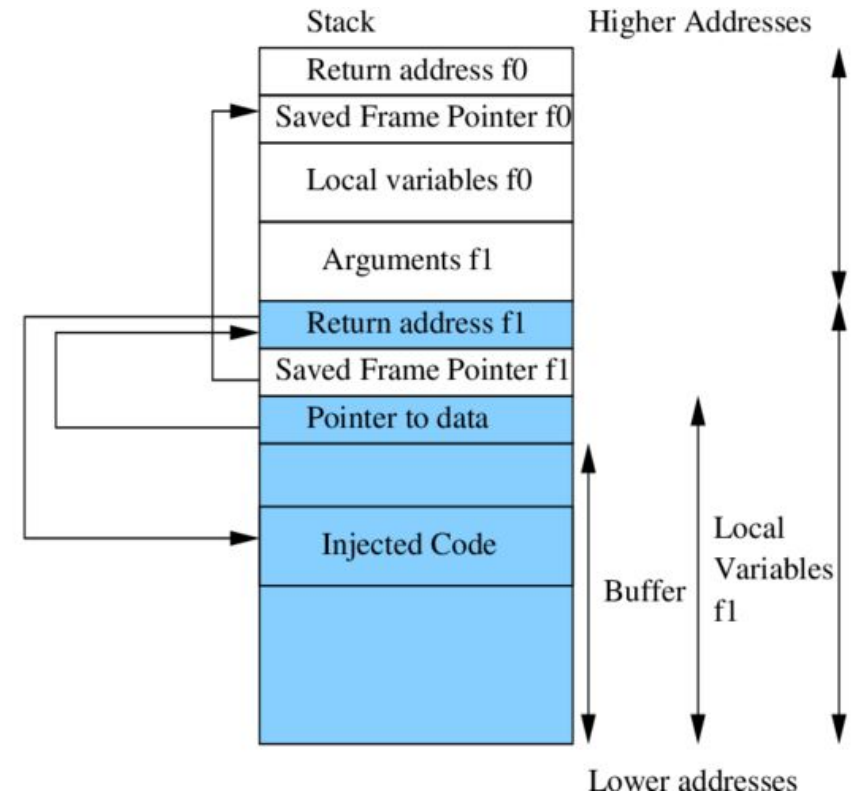
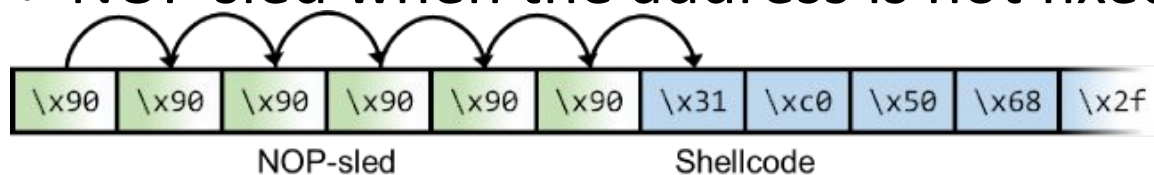
Stack overflow (2)

```
(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)

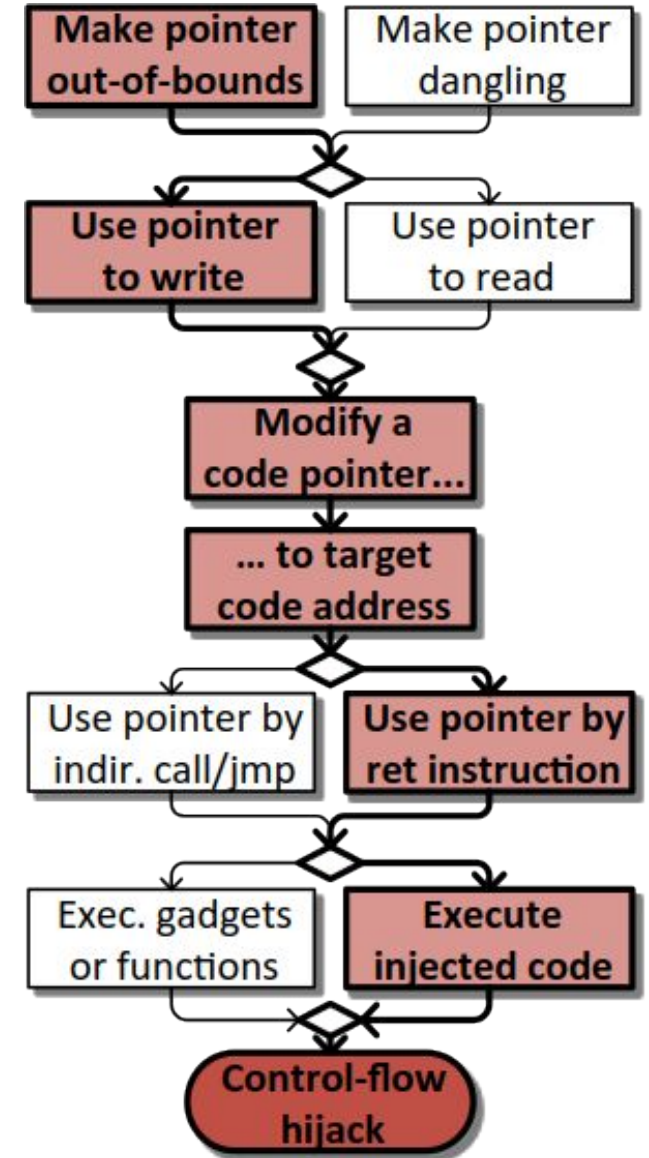
- *ret* instruction will pop the return address from the stack, then jump to it.
- CPU will execute the injected code (shellcode).
- NOP sled when the address is not fixed:



Generic exploit steps [7]

- Find vulnerable input buffer (e.g., stack overflow)
- Find overwritable code pointer offset (e.g., saved EIP)
- Inject/reuse shell code (attacker-defined)
- Corrupt code pointer with attacker value!
- ...
- all your base are belong to us!
(CPU executes malicious instructions)

[7] SoK: Eternal War in Memory

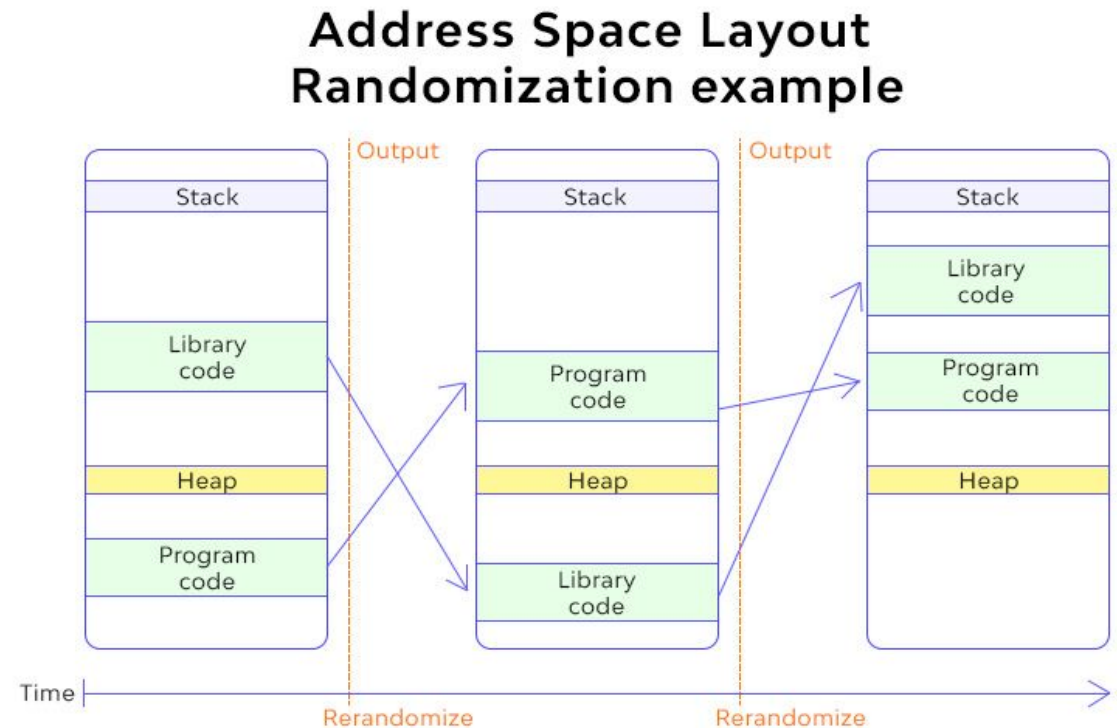


Stack exploit mitigations

- 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 leaks, side channels, data overwrites*
- Shadow Stack

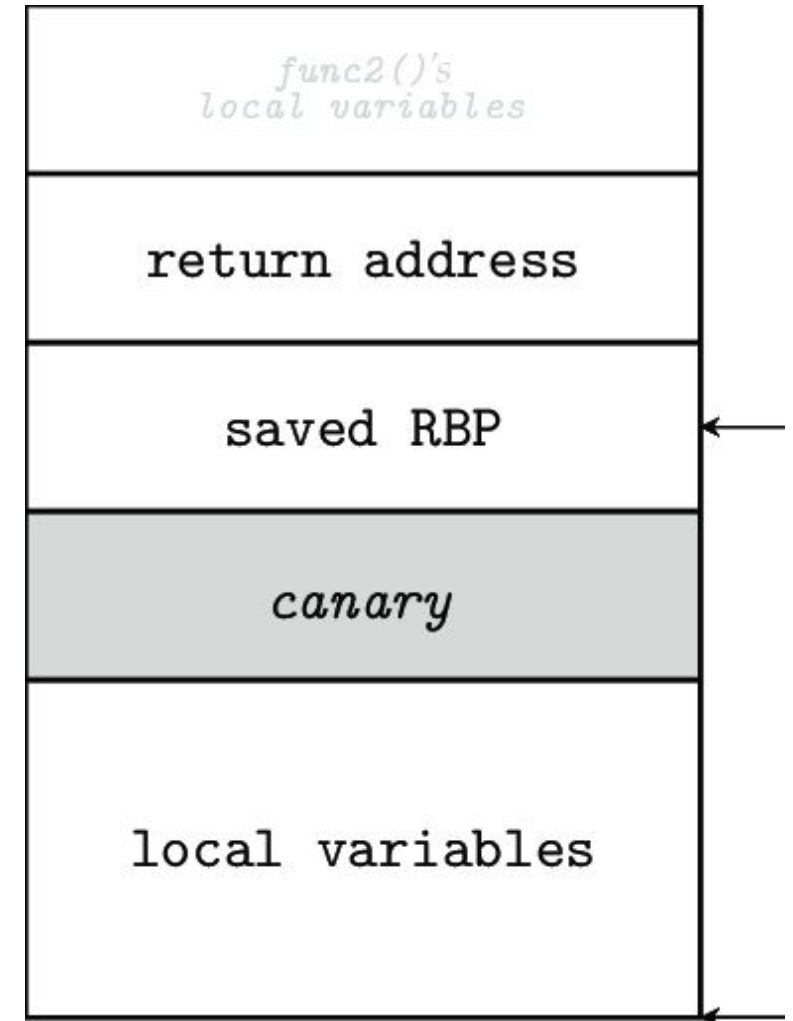
Address Space Layout Randomization (ASLR)

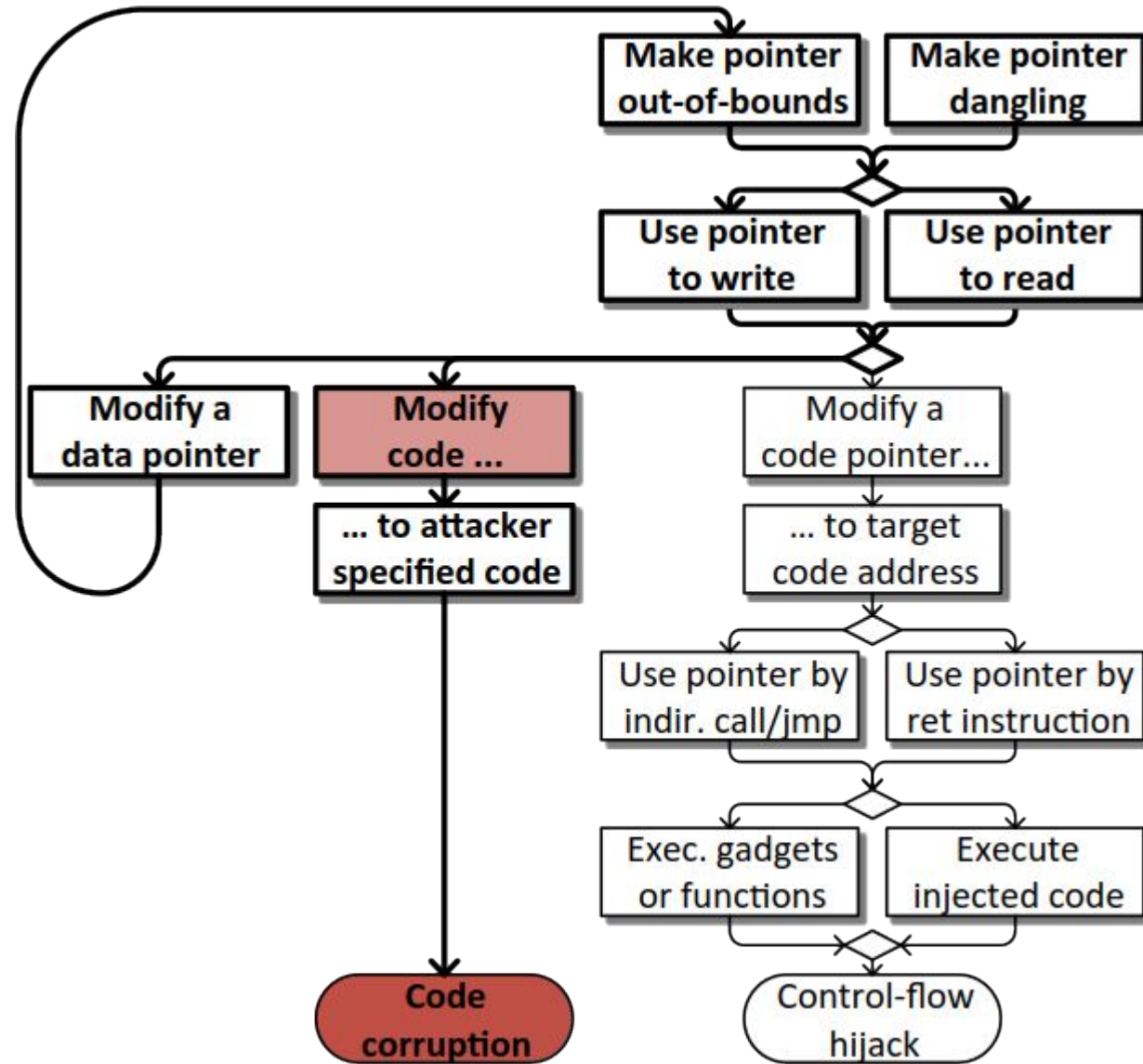
- Implemented by most OSes
- Requires programs be compiled as Position-Independent Code
- Segments can only be randomized at startup!
- KASLR: randomize kernel-space!

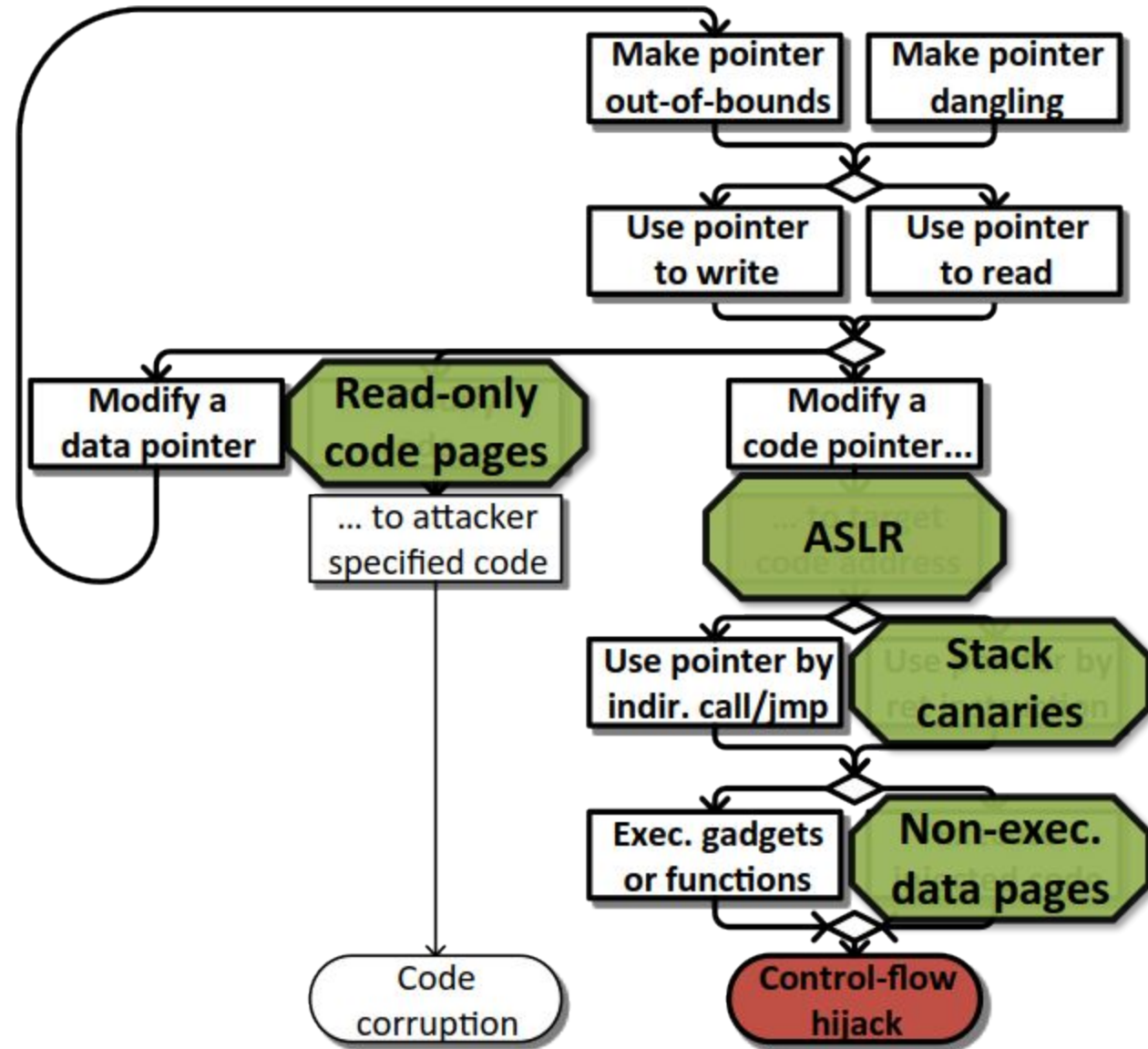


Stack Canaries

- Store random value between Saved EBP + Saved EIP
- Before ret, check this random value
 - If modified, show error & exit
- Weaknesses:
 - memory leaks / side channels
 - canary guessing

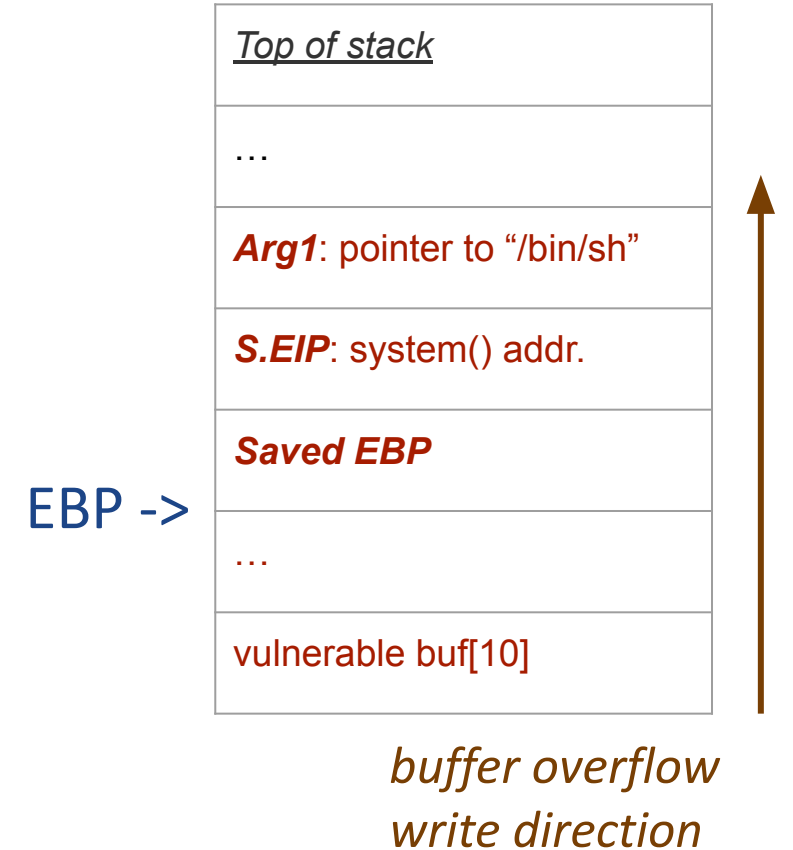




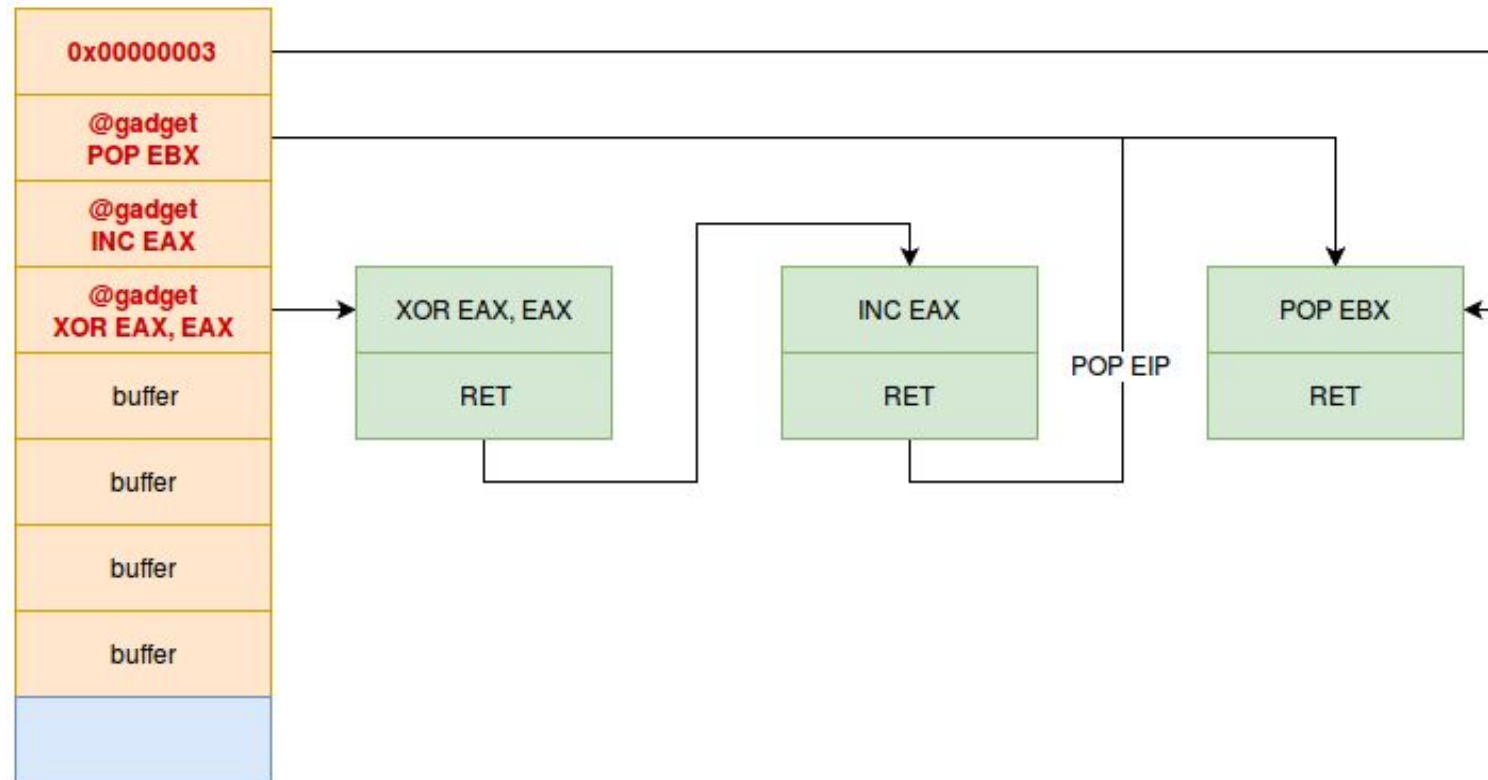


Return to LibC

- Non-executable buffers? no problem!
 - reuse existing functions
- Example: jump to *execve()* / *system()* etc.
 - Reminder: call args from *%EBP - 0x08*!



Return oriented programming



What about exploiting .data?

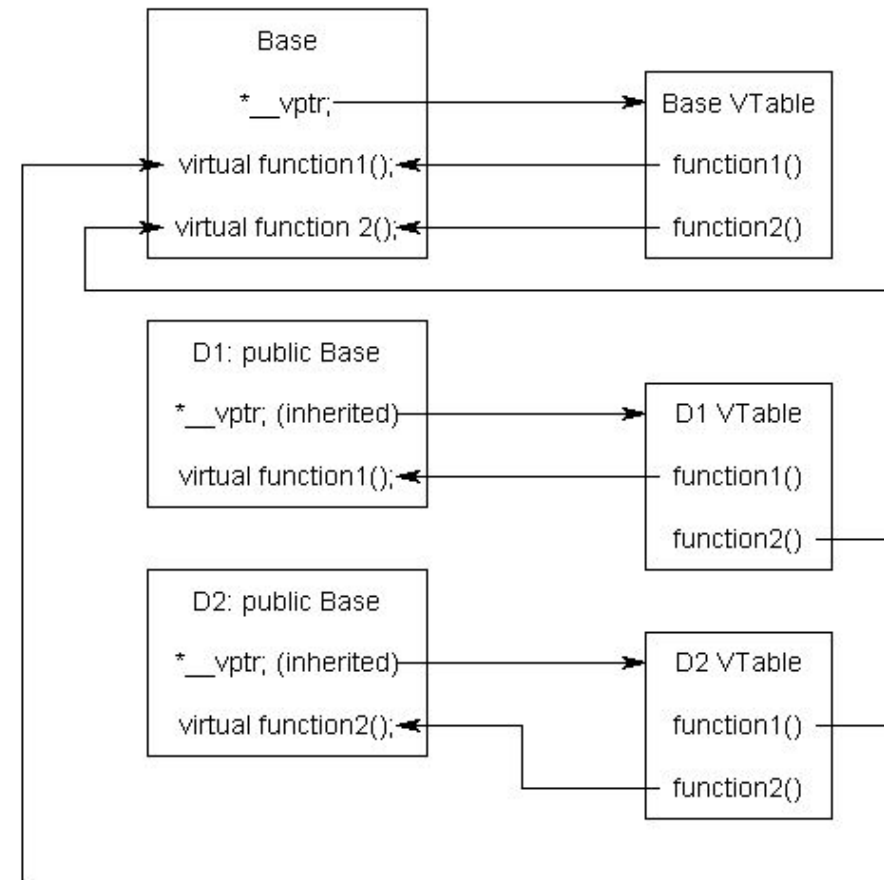
```
struct msg_funcs {  
    (void)(*init)(struct message *msg);  
    (void)(*print)(struct message *msg);  
    (void)(*clear)(struct message *msg);  
}
```

```
struct message {  
    const struct msg_funcs *funcs;  
    int len;  
    char text[255];  
};
```

```
struct message all_messages[10];  
// ...  
int main() {  
    // initialize modules ...  
    for (i=0; i<n; i++) {  
        struct message *msg =  
            &all_messages[i];  
        msg->funcs.init(msg);  
        gets(msg->text);  
    }  
}
```

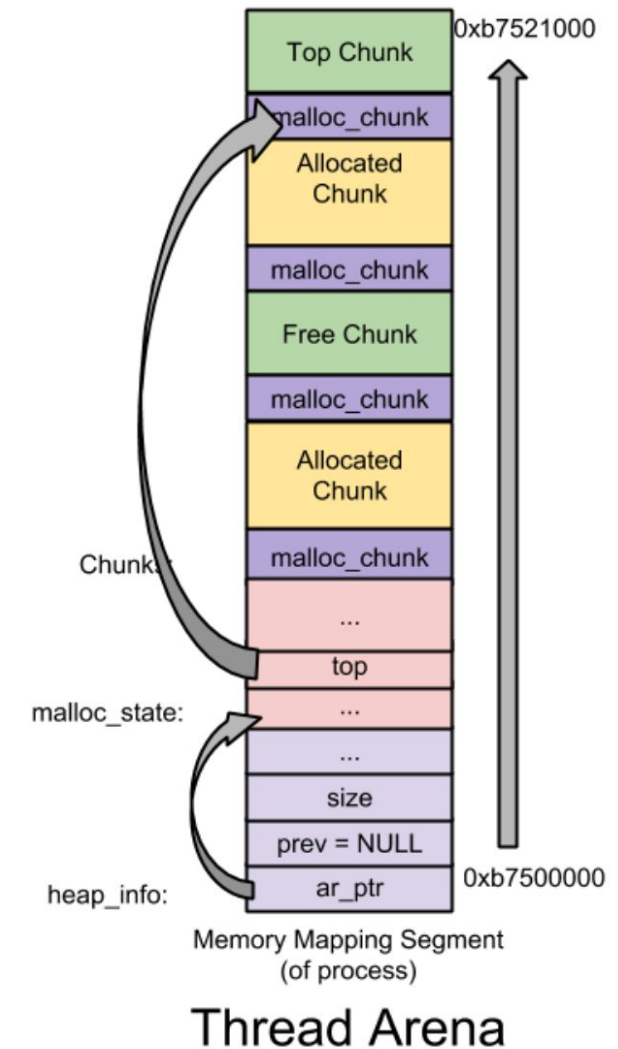
Object Oriented Security

- 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



The Heap

- Dynamic Memory Allocation
 - malloc / new
- Glibc: one master arena, multiple heaps, connected by linked lists
 - Virtual memory allocated by OS (mmap or sbrk)
 - malloc() => returns a **free chunk** of contiguous memory, fills **metadata**
 - free() => clears/resets **chunk** + **metadata**



Dangling Pointer

- Return / store a pointer to an object that will become invalid after a while (e.g., after free)
- Pointer still points to valid memory!
- Example: returning stack-local pointers (function's frame becomes invalid after return)
- Dangerous, especially in OOP languages (e.g.: C++) => attacker can override objects' VTable!

```
char *parse_name(char *input) {  
    char buf[100];  
    // process username  
    return buf;  
}  
  
int main(...) {  
    char *name = parse_name(argv[1]);  
    process_more_data(argv[2]);  
    if (strcmp(name, "admin") == 0)  
        printf("Welcome master!\n");  
}
```

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) { }
```

Size checks vs integer overflows

```
#define HEADER_SIZE 128

uint16_t payload_len = user_payload_size();
uint8_t *buffer = malloc((uint16_t)(payload_len + HEADER_SIZE));

// user gives a valid payload len: 65534
// 65534 + 128 overflows!
// => malloc allocates just 126 bytes...
...
read_input_into_buffer(buffer, len);
```

Format string attacks

- `printf("x=%d, y=%d, z=%d", x, y, z)`
- What if the user controls format string?
 - `printf(user_input)`
 - `"%s"`: read string from address arg.
 - `"%X %X %X..."`: print args as hex
- 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"
[call printf] saves EIP
saved EBP
[printf frame below]

Just in Time + scripting => bytecode injection!

- Defeats $W \oplus X$
- 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

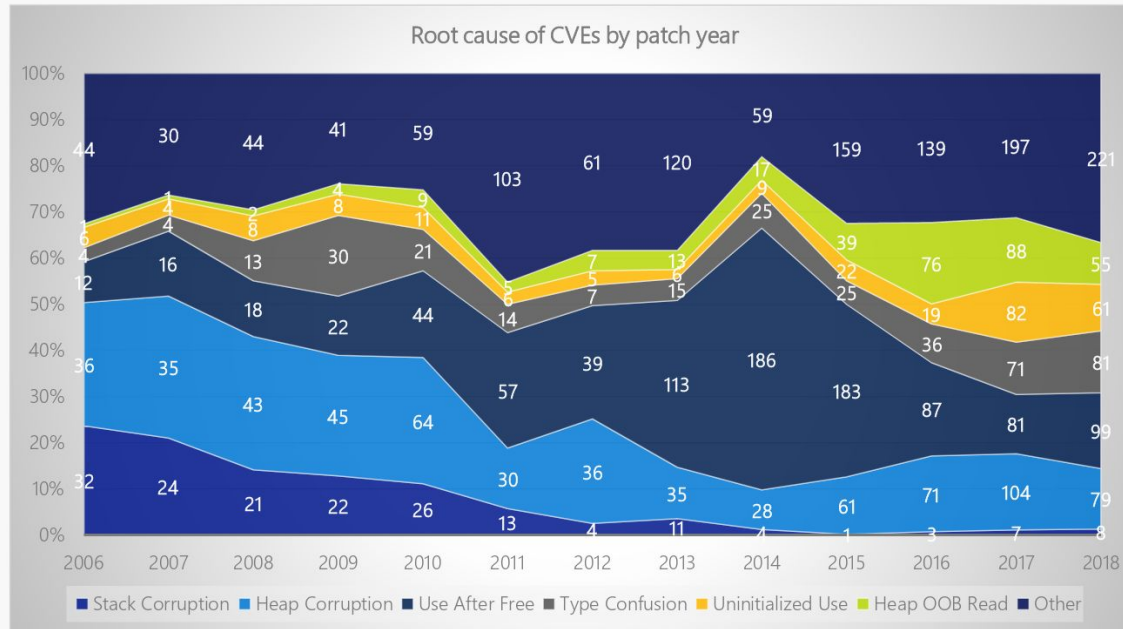
offset pointer with +1 byte:

03: 90 NOP

04: A805 TEST AL, 05

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

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

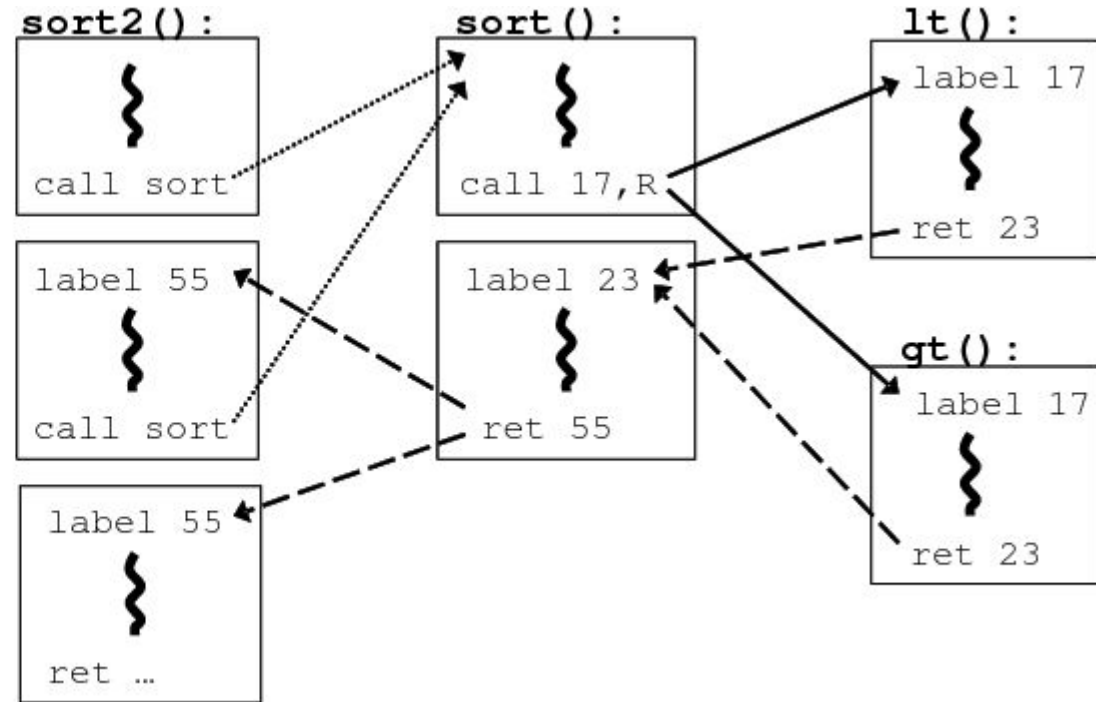
11

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 );  
}
```



Data oriented attacks

- Memory overflows...
non-control flow exploit?

```
int authenticated = 0;
struct user_info *user_ptr =
                                auth_users[last_idx];

int username[100];
gets(username);
// meanwhile: check name & password
if (authenticated) {
    user_ptr->valid = 1;
    strcpy(user_ptr->name, username);
    last_idx++;
}
```

Data oriented attacks (2)

- Memory overflows => non-control exploit
 - Defeat $W \oplus X$, stack canaries, CFI etc. (we don't alter code execution)!
- **Data Oriented Programming**
Gadgets
 - Pointer write access => write to ANY program variable!

```
int authenticated = 0;
struct user_info *user_ptr =
    auth_users[last_idx];

int username[100];
gets(username);
// meanwhile: check name & password
if (authenticated) {
    user_ptr->valid = 1;
    strcpy(user_ptr->name, username);
    last_idx++;
}
```

Data integrity

- Easy: check bounds after each read / write of any variable!
- Softbounds + CETS
 - compile-time transformations for enforcing spatial safety and temporal safety for C
 - Huge overhead!
- Write Integrity Tracking / Data Flow Integrity / Data Space Randomization

Protection mechanism summary

	Policy	Technique	Weakness	Perf.	Comp.
Hijack protection	$W \oplus R$	Page flags	JIT	1x	Good
	Return integrity	Stack cookies	Direct overwrite	1x	Good
	Address space rand.	ASLR	Info-leak.	1.1x	Good
	Control-flow integ.	CFI	Over-approx.	1.4x	Libraries
Generic protection	Memory safety	SB+CETS	None	2-4x	Good
	Data integrity	WIT	Over-approx.,...	1.2x	Libraries
	Data space rand.	DSR	Over-approx.,...	1.3x	Libraries
	Data-flow integrity	DFI	Over-approx.	2-3x	Libraries

Is zero-vulnerabilities software possible?

- **Yep! qmail [6]**
 - Mail Transfer Agent by **David Bernstein**, 1995 (last version: 1.03, 1998)
 - Zero security vulnerabilities so far!
- **Security practices:**
 - Keep It Simple Stupid (KISS)
 - Unix Philosophy (modular development, each component KISS)
 - Separate functions into multiple unprivileged binaries
 - Don't parse! Pass uniform/binary messages between programs!
 - Write careful code, avoid libc (gets, printf, malloc/free etc.)!

Memory safety defense?

- Scripting (Python, JS etc.) / Java / C#?
 - not if you need performance (e.g., games, system level stuff)...
- Rust / GoLang / Zig (etc.)
 - still able to write “unsafe” code
(e.g., syscalls / hardware interfaces / code optimizations)
- Stronger typing systems:
 - Pure-functional programming (e.g., Haskell)
 - ATS (write both code + mathematical proofs!) => HARDEST
- Backwards compatibility...
 - no money to rewrite everything from scratch
 - use secure coding practices, static analysis tools etc.
 - hardware pointer/boundary checks (Intel MPX, ARM PAC)

References

- [1] <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/>
- [2] <https://security-summer-school.github.io/binary/>
- [3] <https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf>
- [4] <https://sourceware.org/glibc/wiki/MallocInternals>
- [5] <https://infosecwriteups.com/use-after-free-13544be5a921>
- [6] <https://blog.acolyer.org/2018/01/17/some-thoughts-on-security-after-ten-years-of-qmail-1-0/>
- [7] SoK: Eternal War in Memory <https://www.ieee-security.org/TC/SP2013/papers/4977a048.pdf>
- [8] <https://people.scs.carleton.ca/~paulv/toolsjewels.html>