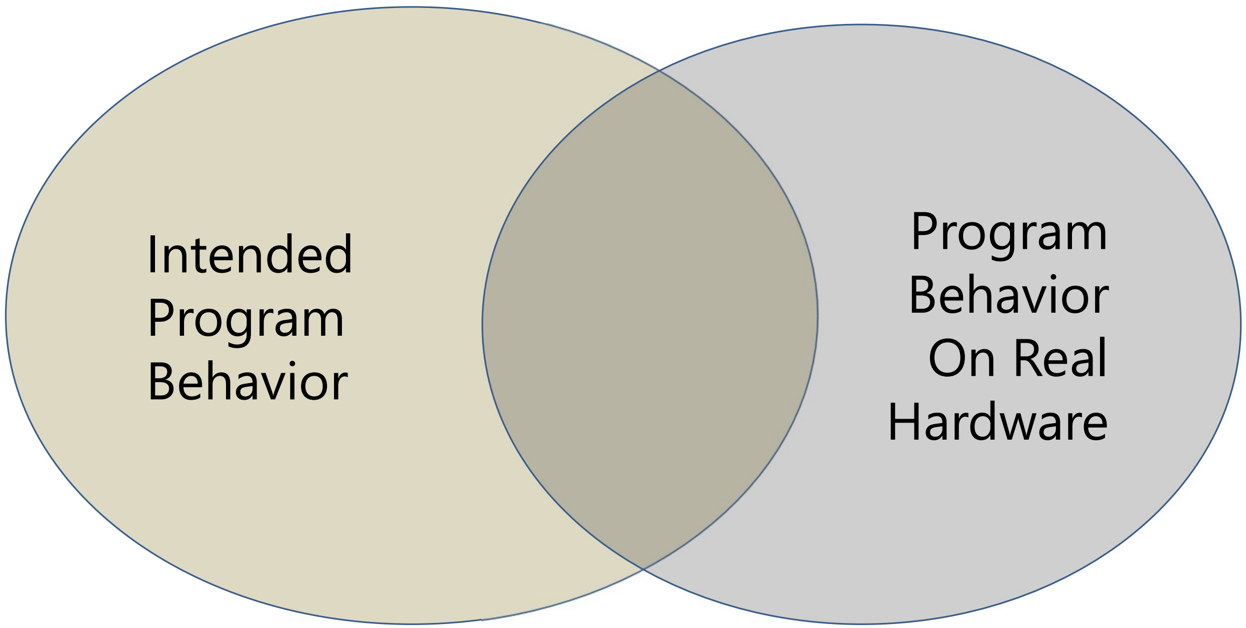
From Vulnerabilities To

Memory Exploits

Prateek Saxena

Recap

- Vulnerabilities
  - Memory Errors
  - Type Errors
- C/C++: weak type safety, no memory safety
- Hardware does <u>not</u> give memory & type safety



Reminder: Hands-on Exercises

- "Self-Study" Tutorials on LumiNUS
  - Install the VM
  - Follow the step-by-step tutorial in the notes
  - It will be helpful for Coding Assgt. 2 (TBA)

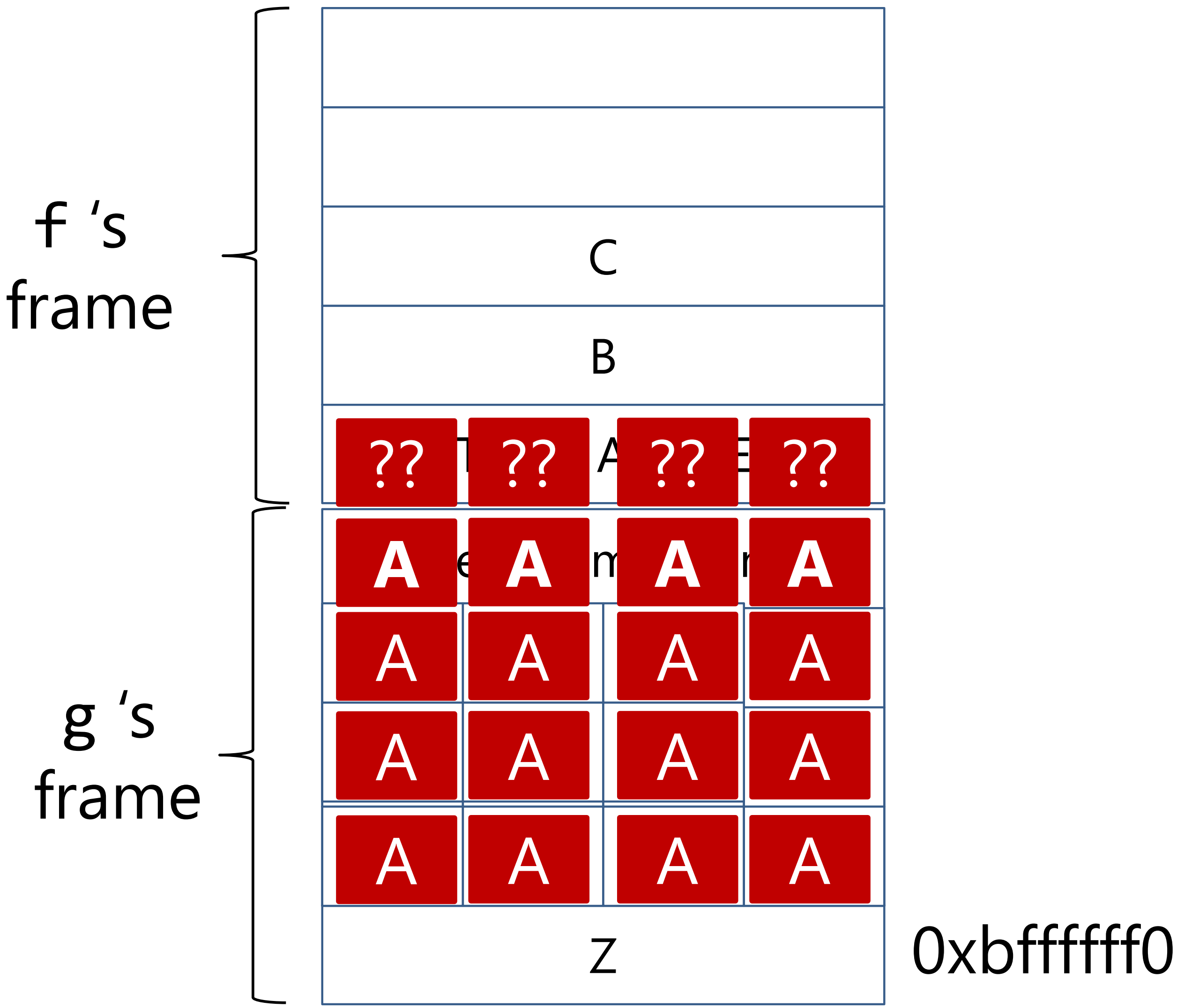
Control-flow Hijacking Exploits: Code Injection

- Control-oriented a.k.a control-flow hijacking
- Outcome 1: Code Injection
  - ***Definition***: *A memory exploit that hijacks control to <u>jump to attacker's data payload</u>*

Code Injection Example

```
int f() {
…
   g (x, y);
}

int g(int x, int y) {
   char buf[50];
   scanf("%s", buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



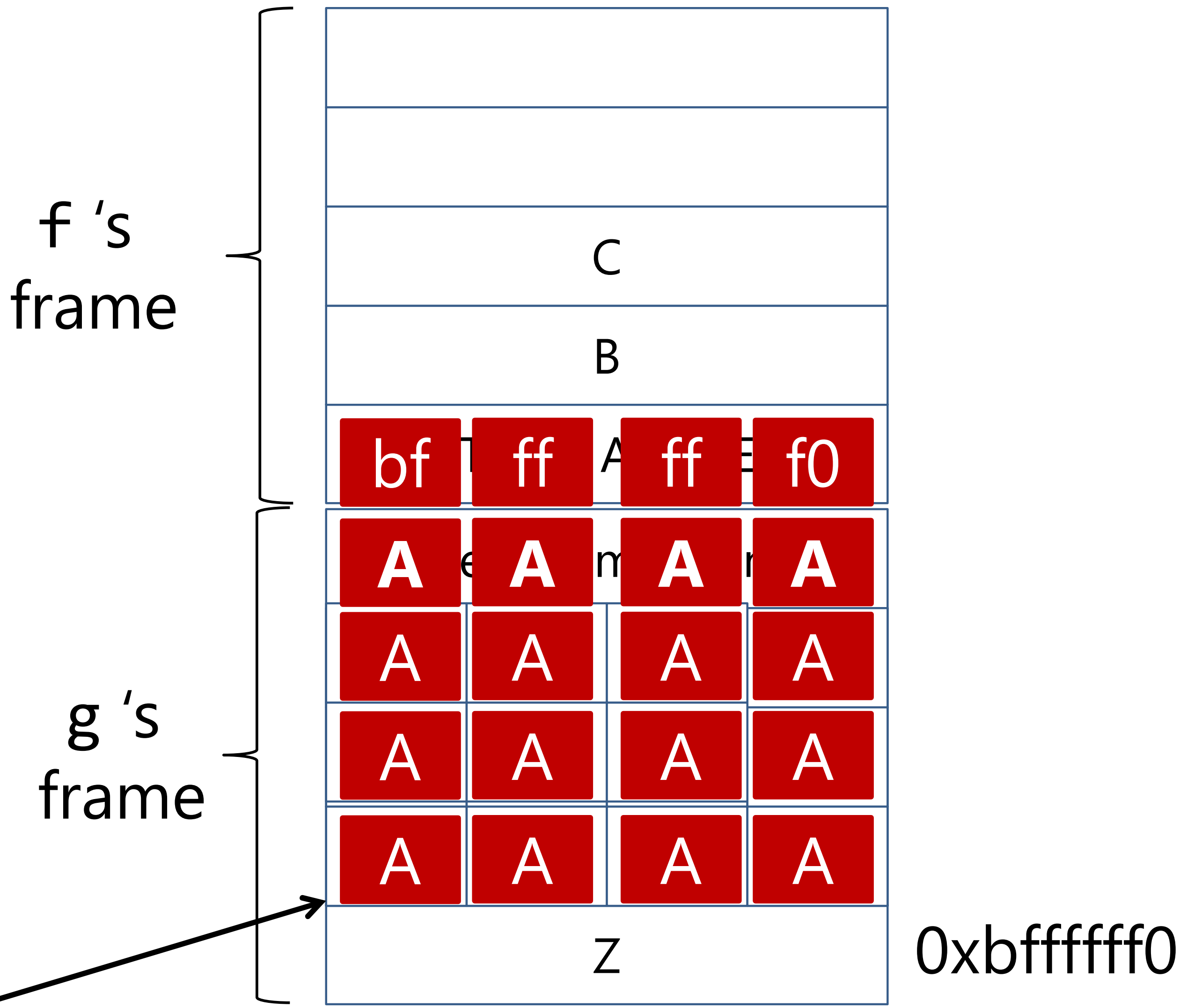
What should be Values for ??

Code Injection Example

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf("%s", buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```

Will return inside attacker's buffer



What should be Values for A?

Control-oriented Exploits: Code Injection Example

- Example: Payload

```
int main(int argc,
   char*argv[])
   {
       char *sh;
       char *args[2];

       sh = "/bin/bash";
       args[0] = sh;
       args[1] = NULL;
       execve(sh, args, NULL);
   }
```

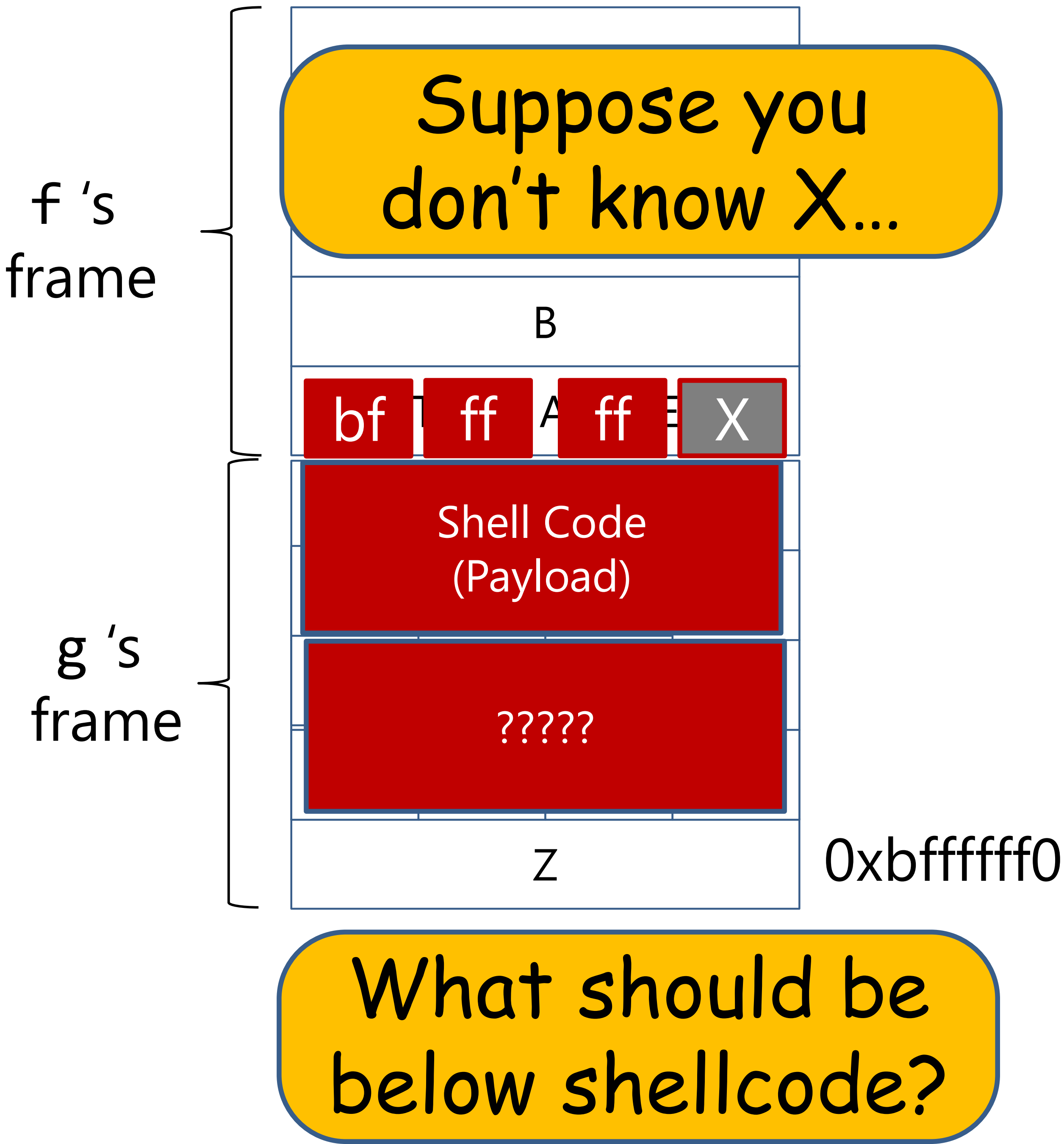
- Shell Code

```
90 90 eb 1a 5e 31 c0 88
  46 07 8d 1e 89 5e 08
  89 46 0c b0 0b 89 f3
  8d 4e 08 8d 56 0c cd
  80 e8 e1 ff ff ff 2f
  62 69 6e 2f 73 68 20
  20 20 20 20 20
```

Code Injection Example

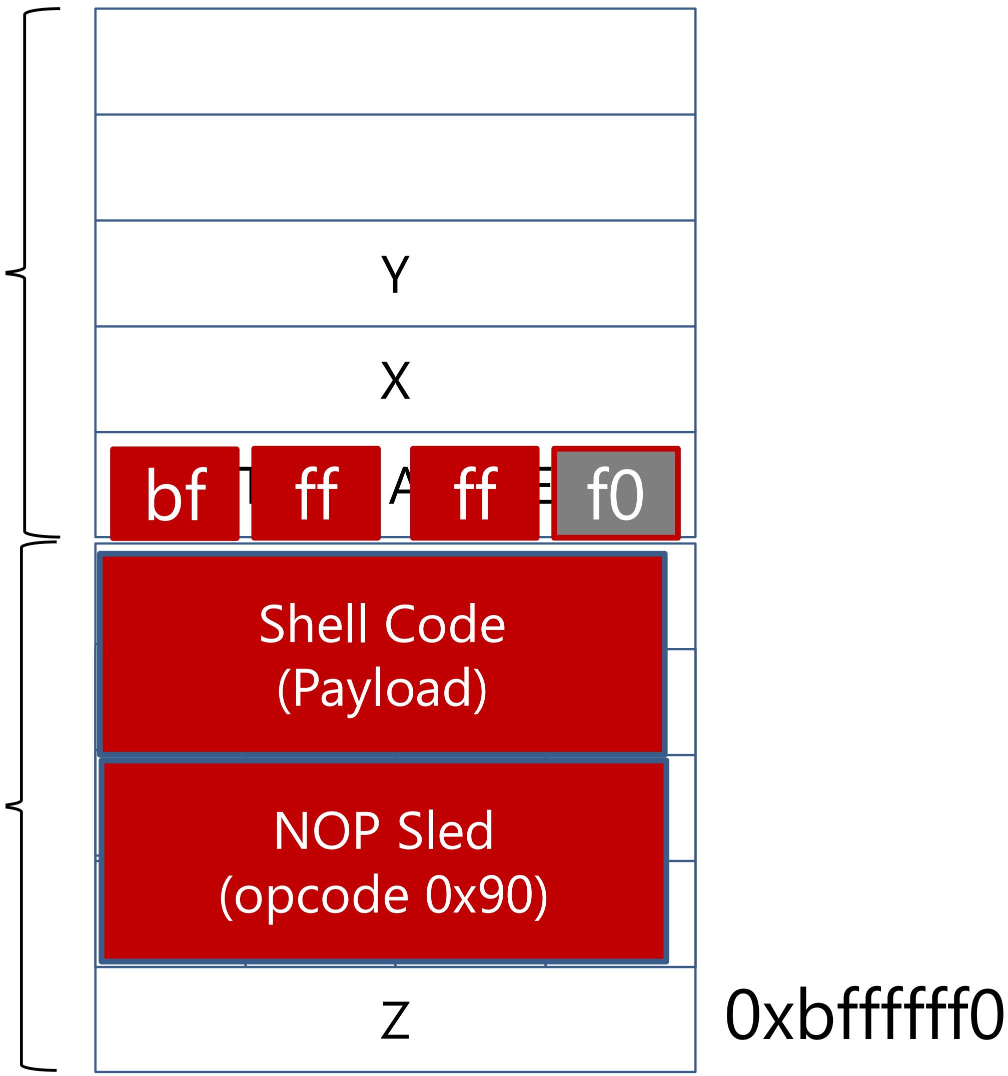
```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp


…
call scanf
…
pop ebp
ret
```



Code Injection Example

- Instruction NOP, No Operation.
  - Tell CPU to do nothing and fetch the next instruction
- Including a large block of NOP instructions in the injected code as *landing area*
- Execution will reach shell code as long as return address pointing to somewhere in the NOP sled



Adv: You can jump anywhere in the NOP sled

Control-flow Hijacking: Code Injection

- Control-oriented a.k.a control-flow hijacking
- Outcome 1: Code Injection
  - ***Definition***: *A memory exploit that hijacks control to <u>jump to attacker's data payload</u>*

Code Injection: Requirements

- Req 1: Write Attack Payload in memory
- Req 2: Have Attack Payload Be Executable
- Req 3: Divert control-flow to payload

More Control-flow Hijacking: Code Reuse

Control-oriented Exploits (II): Code Reuse

- Outcome 2: Code Reuse
  - ***Definition***: *A memory exploit that hijacks control to jump to attacker's controlled code address*
- Requirements for Code Reuse
  - ~~Req 1: Write Attack Payload in memory~~
  - Req 2: Have Attack Payload Be Executable
  - Req 3: Divert control-flow to payload

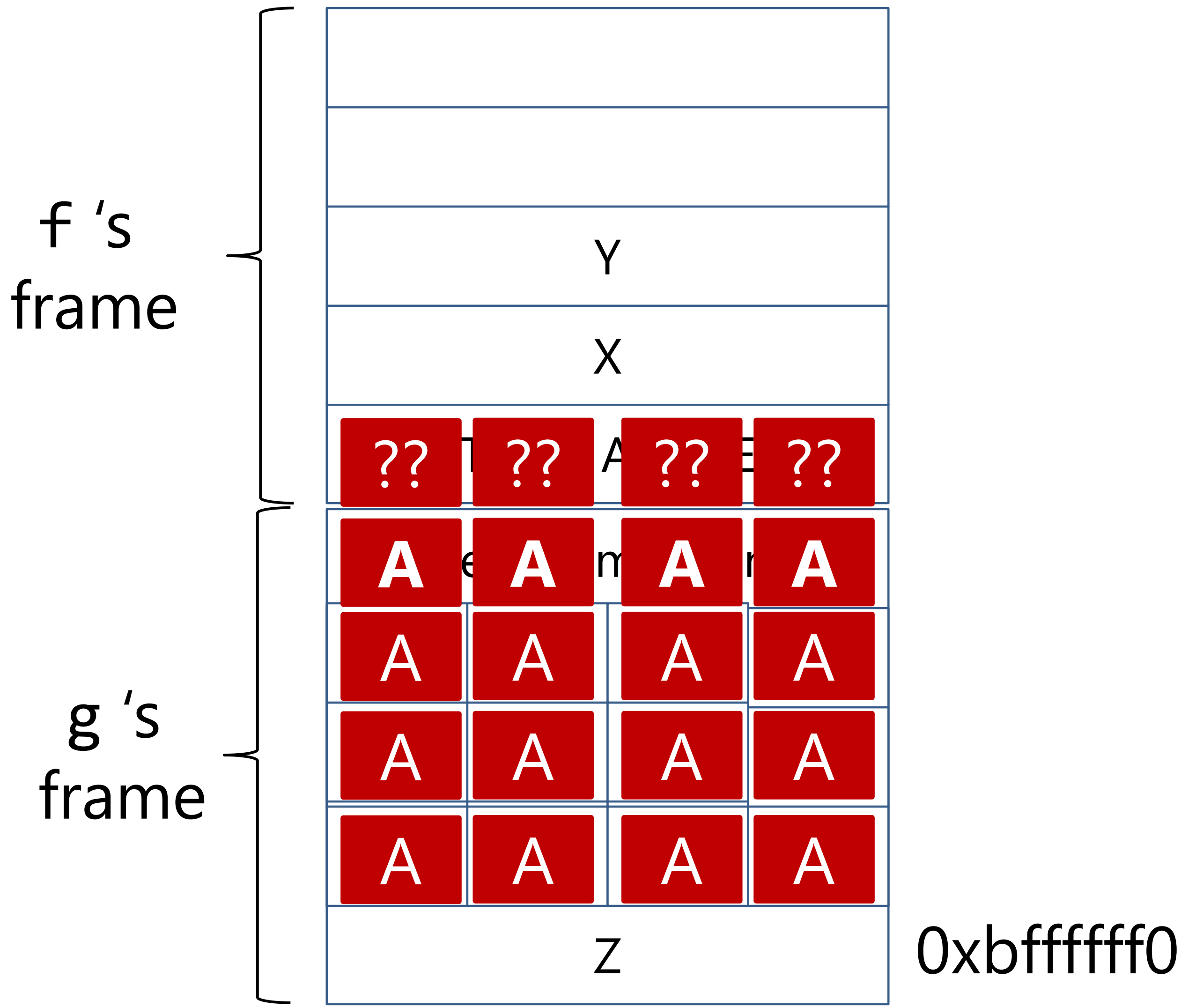
- Insight: Re-use the existing code as payload

Code Reuse:
The Idea

- Attacker hijacks control flow
- Jumps back to the code segment
- Example: Return-to-libc

Code Reuse Attack: Return-to-libc

```
.g

push ebp


…
call scanf
…
pop ebp
ret
```

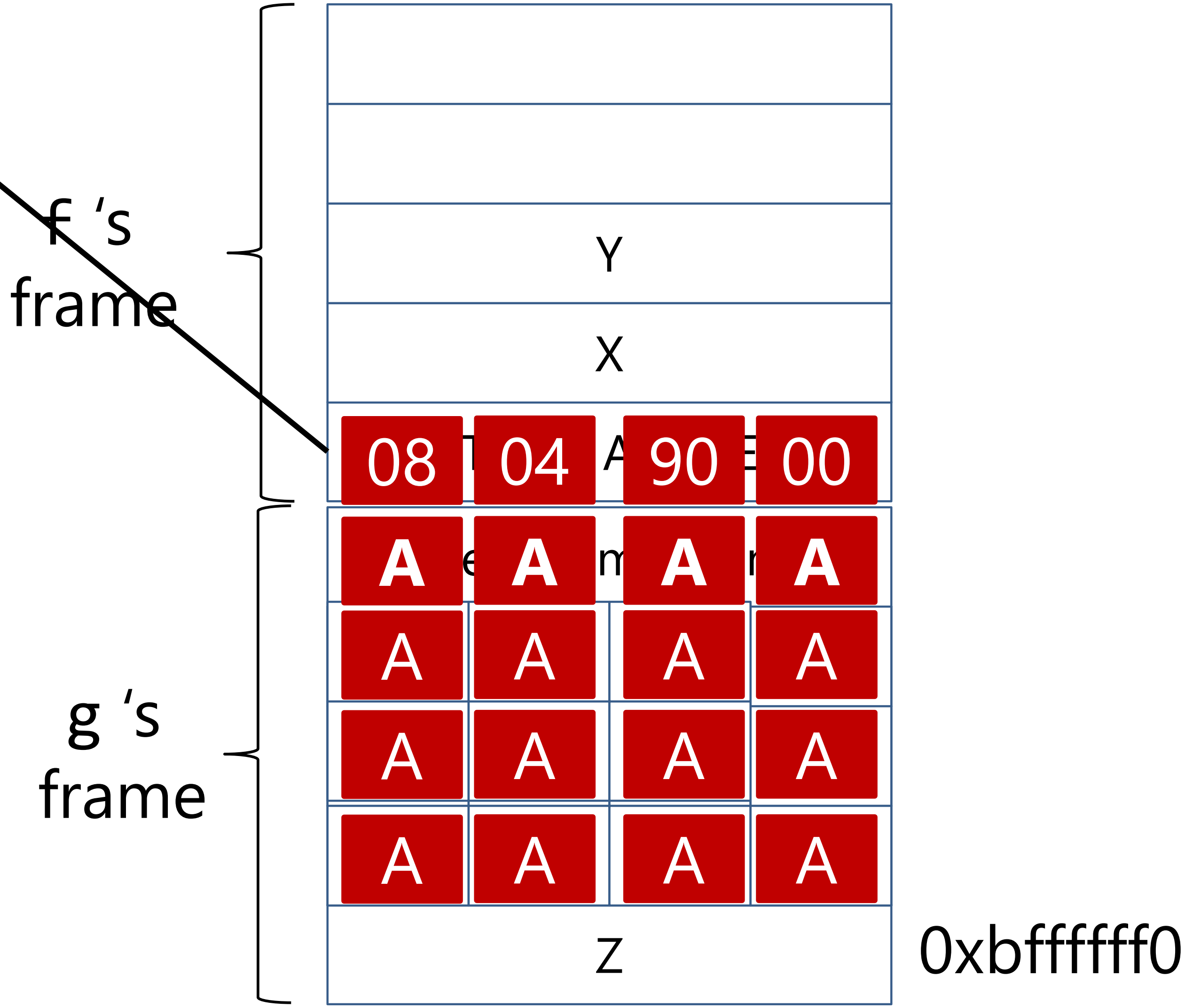


What should you do?

Code Reuse Attack: Return-to-libc

```
.execv <0x8049000>

push ebp
… // read parameter1
… // read parameter2
pop ebp
ret
```

```
.g

push ebp


…
call scanf
…
pop ebp
ret
```

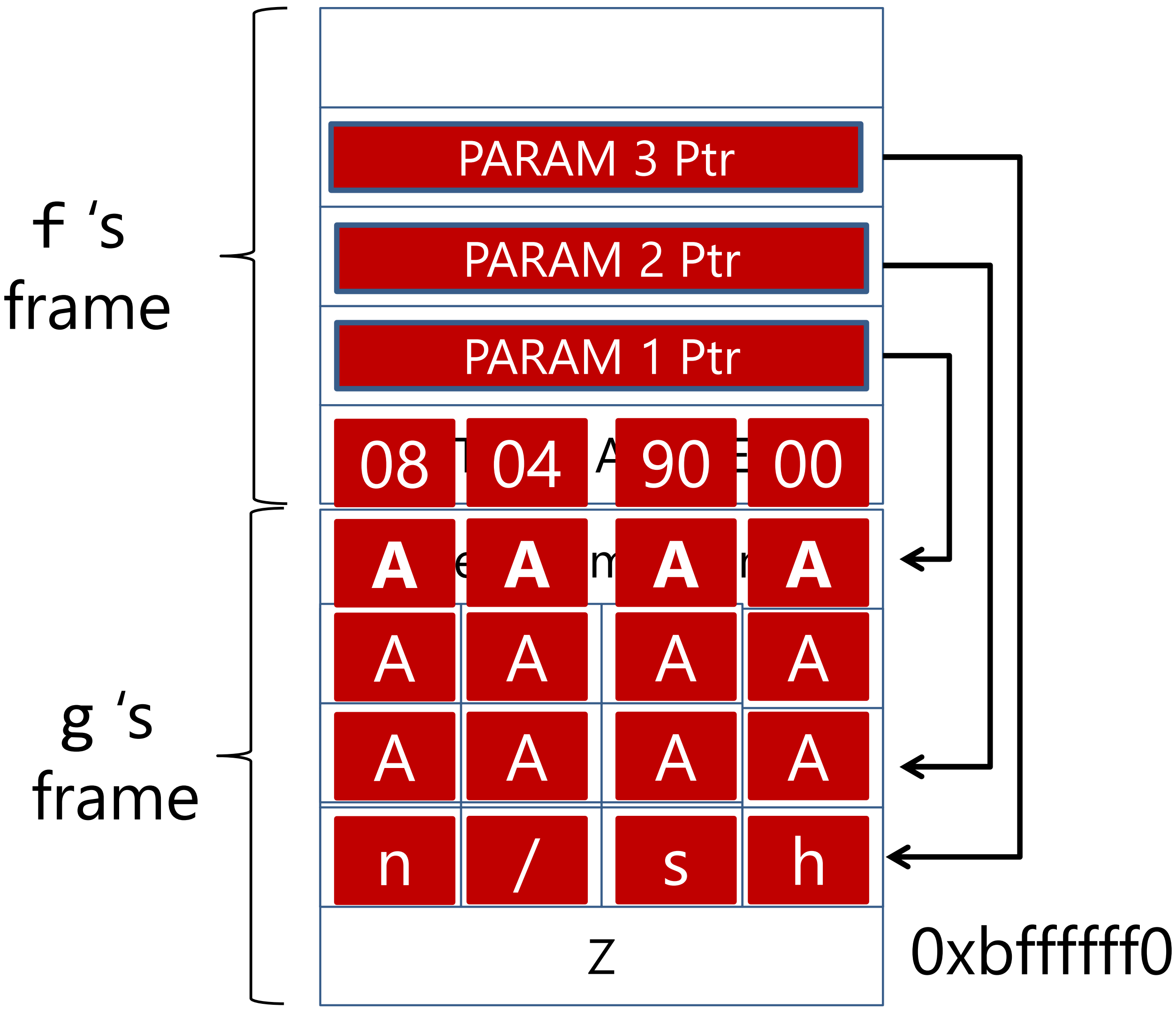


Change return address to code seg.

Code Reuse Attack: Return-to-libc

```
.execv <0x8049000>

push ebp
… // read parameter1
… // read parameter2
… // read parameter3
pop ebp
ret
```

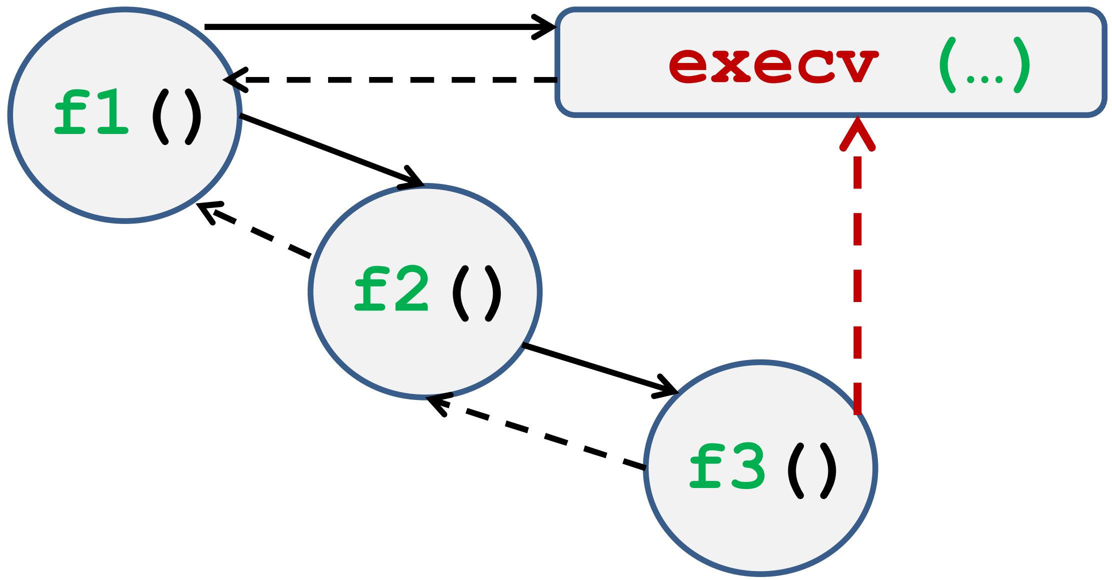
```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



Setup arguments to execv on stack

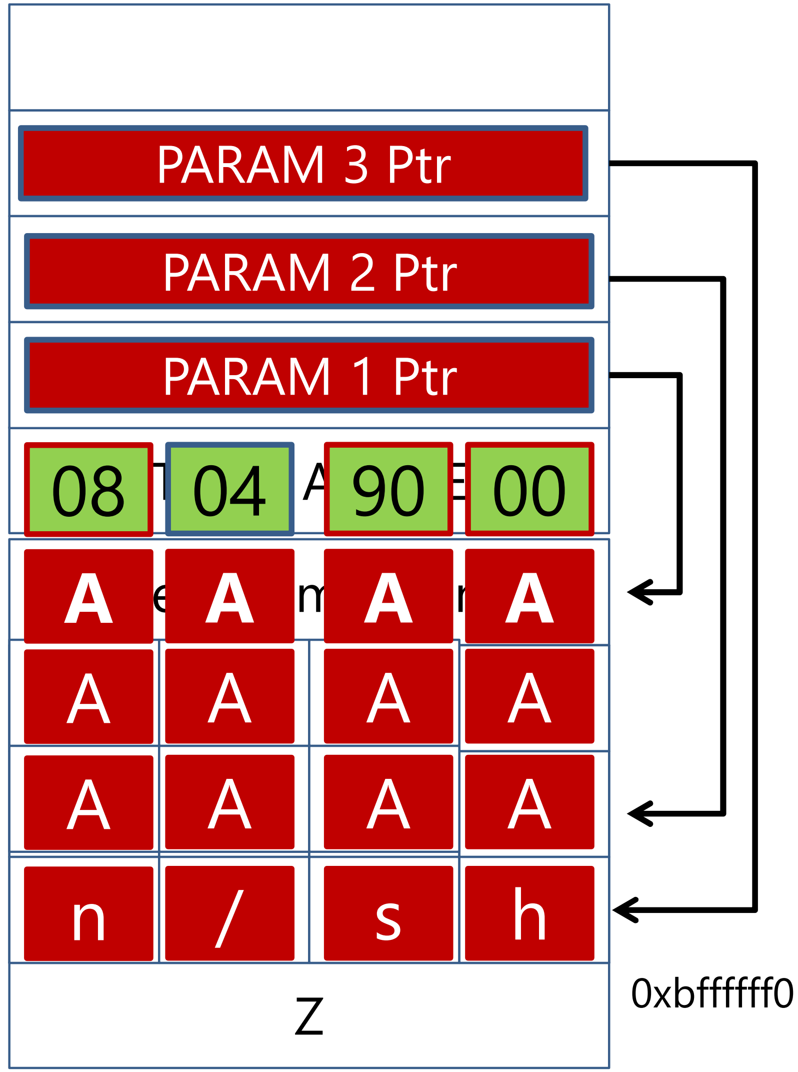
Return-to-Libc

- Introduce new Control Edges



Control Flow Graph

Call Edges
Ret Edges



Break!

Code Reuse (II): ROP

Control-oriented Exploits (II): Code Reuse

- Outcome 2: Code Reuse
  - ***Definition***: *A memory exploit that hijacks control to <u>jump to attacker's controlled code address</u>*
- Requirements for Code Reuse
  - ~~Req 1: Write Attack Payload in memory~~
  - Req 2: Have Attack Payload Be Executable
  - Req 3: Divert control-flow to payload
- Insight: Re-use the existing code as payload

Code Reuse Attacks (II): Return-oriented Programming (ROP)

- Key Observation:

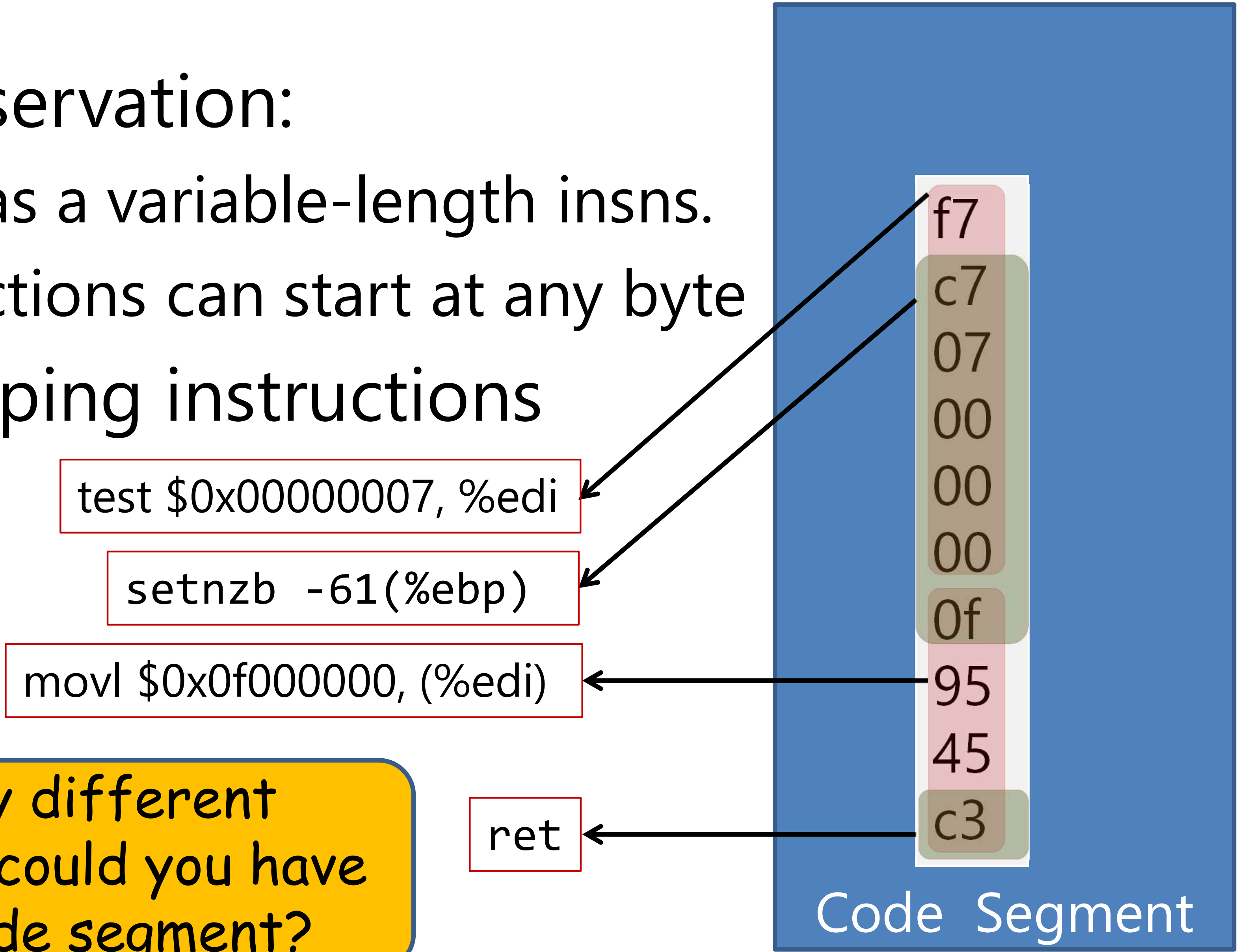
`f7 c7 07 00 00 00`	test \$0x00000007, %edi
`0f 95 45 c3`	setnzb -61(%ebp)
`c7 07 00 00 00 0f`	movl \$0x0f000000, (%edi)
`95`	xchg %ebp, %eax
`45`	inc %ebp
`c3`	ret

What is the similarity between these machine code snippets?

Return-oriented programming

Code Reuse Attacks (II): Return-oriented Programming

- Key Observation:
  - X86 has a variable-length insns.
  - Instructions can start at any byte
- Overlapping instructions



How many different instructions could you have in large code segment?

ROP Gadgets

ROP Gadget: Instruction sequence which end a control transfer

```
return; // C3
```

- Add 4 to ESP
- Jump to *(ES

Return Address

ESP

Control flow

```
sodsd / lodsd
```

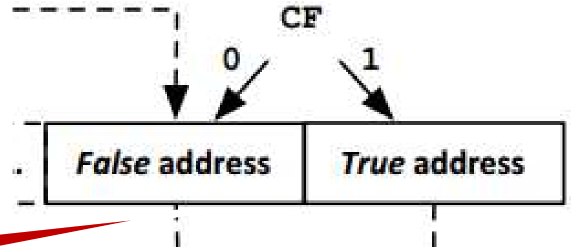
```
A-B // Set CF
… // Use CF to
decide jump address
```

M -> Reg,
Reg -> M

```
pop ebp; // 5D
```

- Set EBP to *(ESP)
- Add 4 to ESP

M -> Reg, Artithmetic



Comparison

Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming [WOOT'12]

ROP: An Example

Let's say the attacker's goal is: `*(0x4a304120) = *(0x4a304120) + 0x00032400`

Attacker find the following **gadgets** in code memory:

```
8331c034    pop eax
            ret

831e5201    pop ebx
            ret

832a34b9    mov eax, [eax]
            ret

8324f321    add eax, ebx
            ret

83294a3a    pop ecx
            ret

8320d265    mov [ecx], eax
            ret
```

Exploit Payload (Stack)

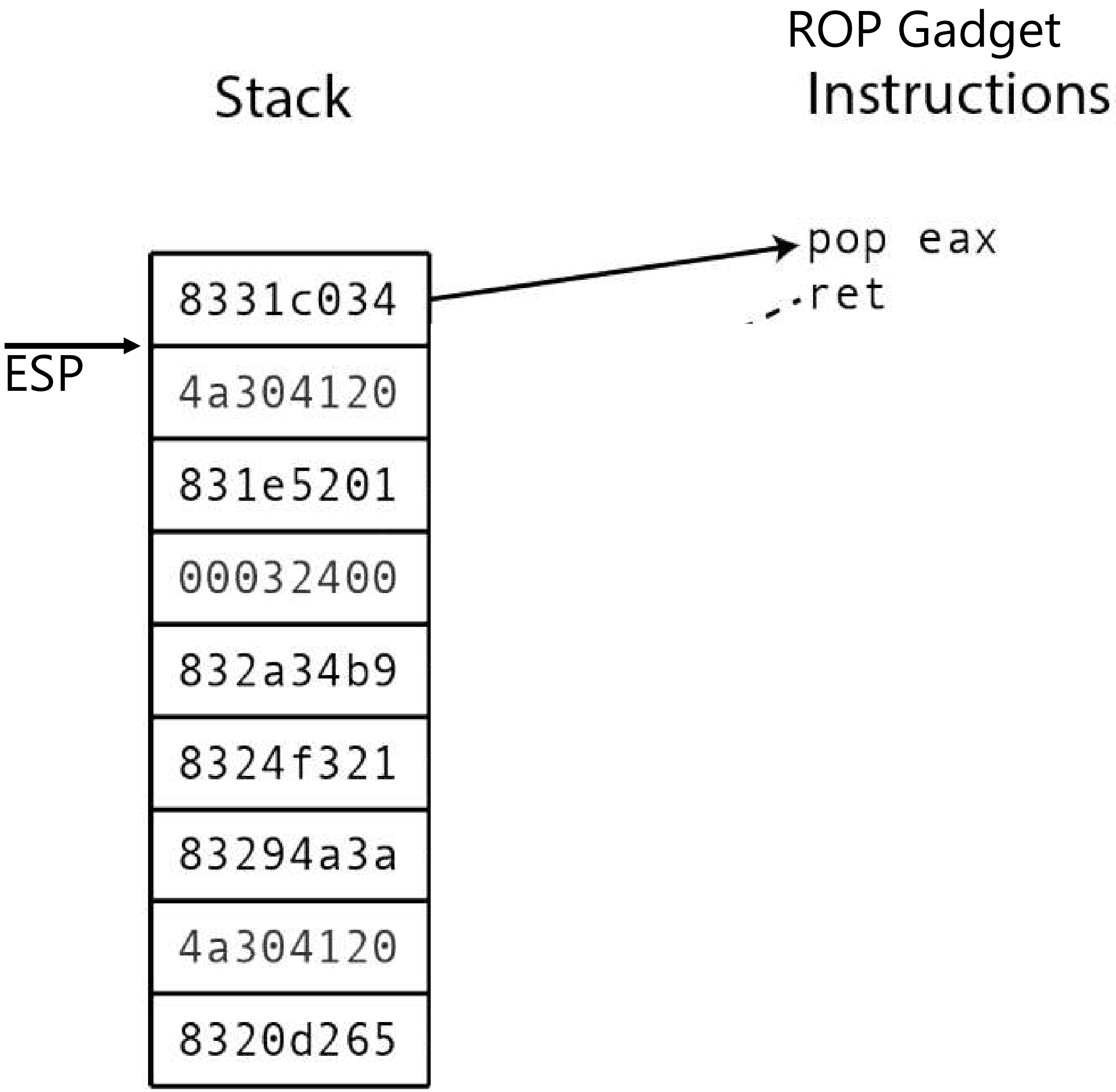
ESP Before Ret →

8331c034
4a304120
831e5201
00032400
832a34b9
8324f321
83294a3a
4a304120
8320d265

Items Popped Downwards

ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

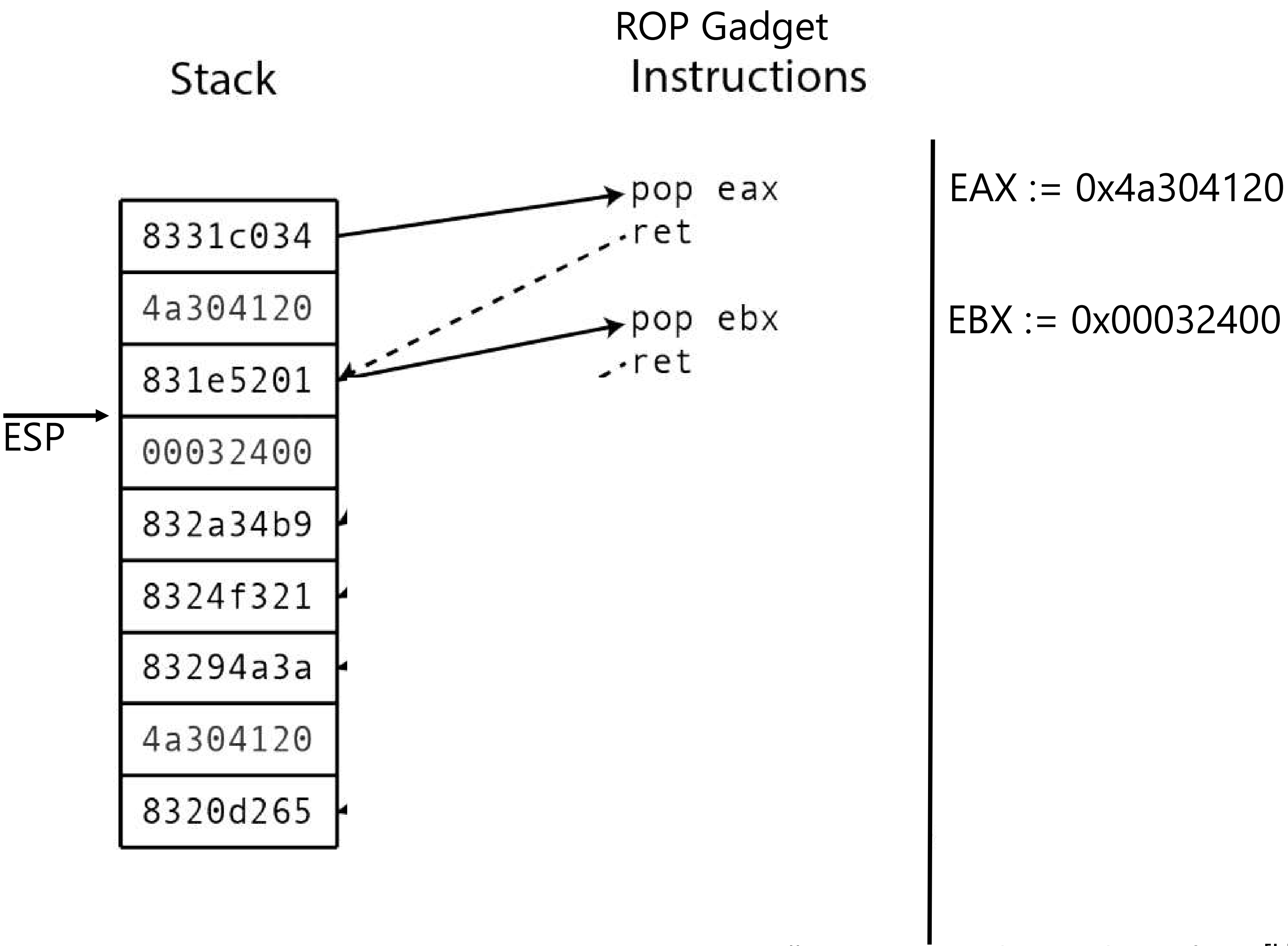
ROP: An Example



EAX := 0x4a304120

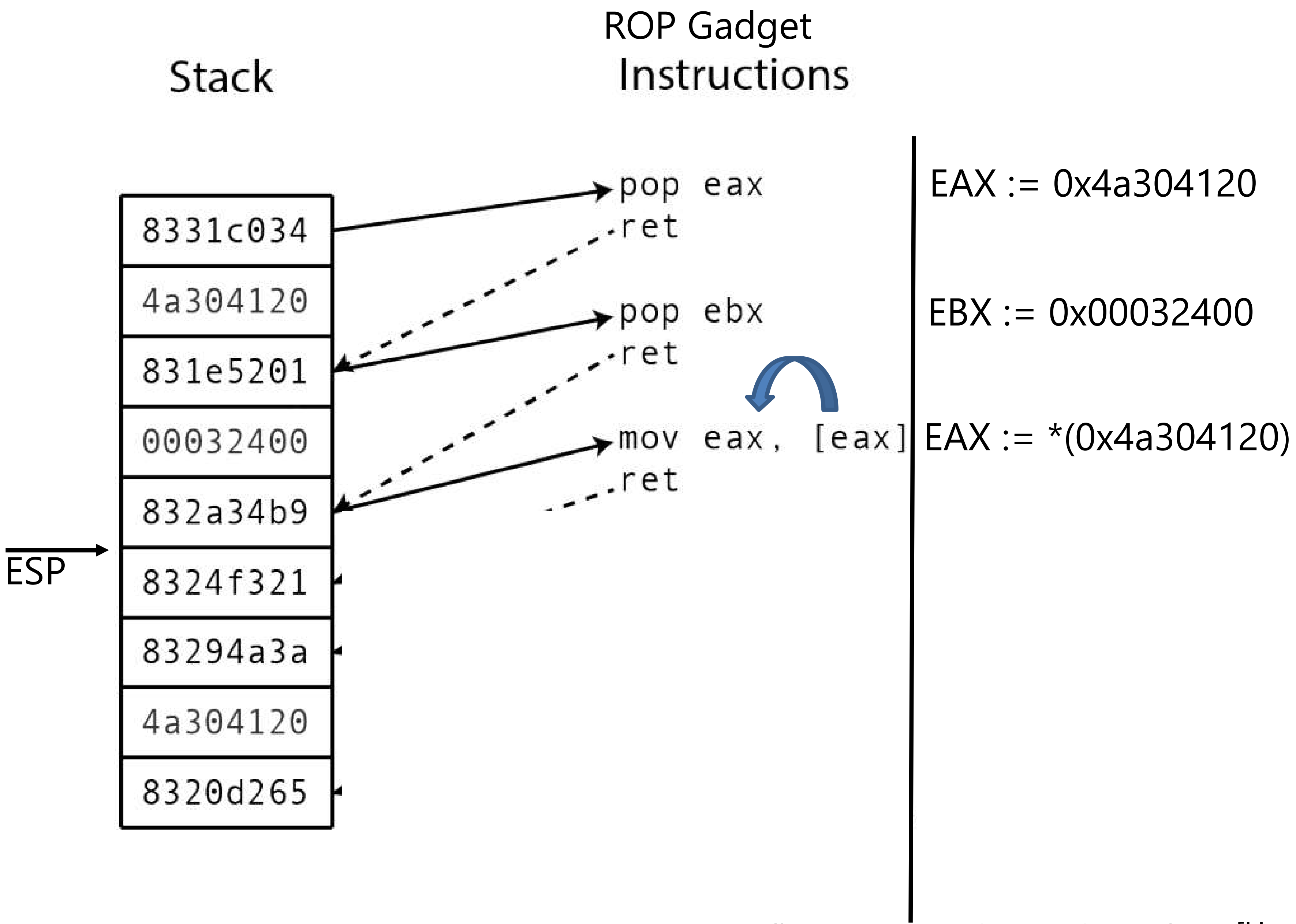
ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

ROP: An Example



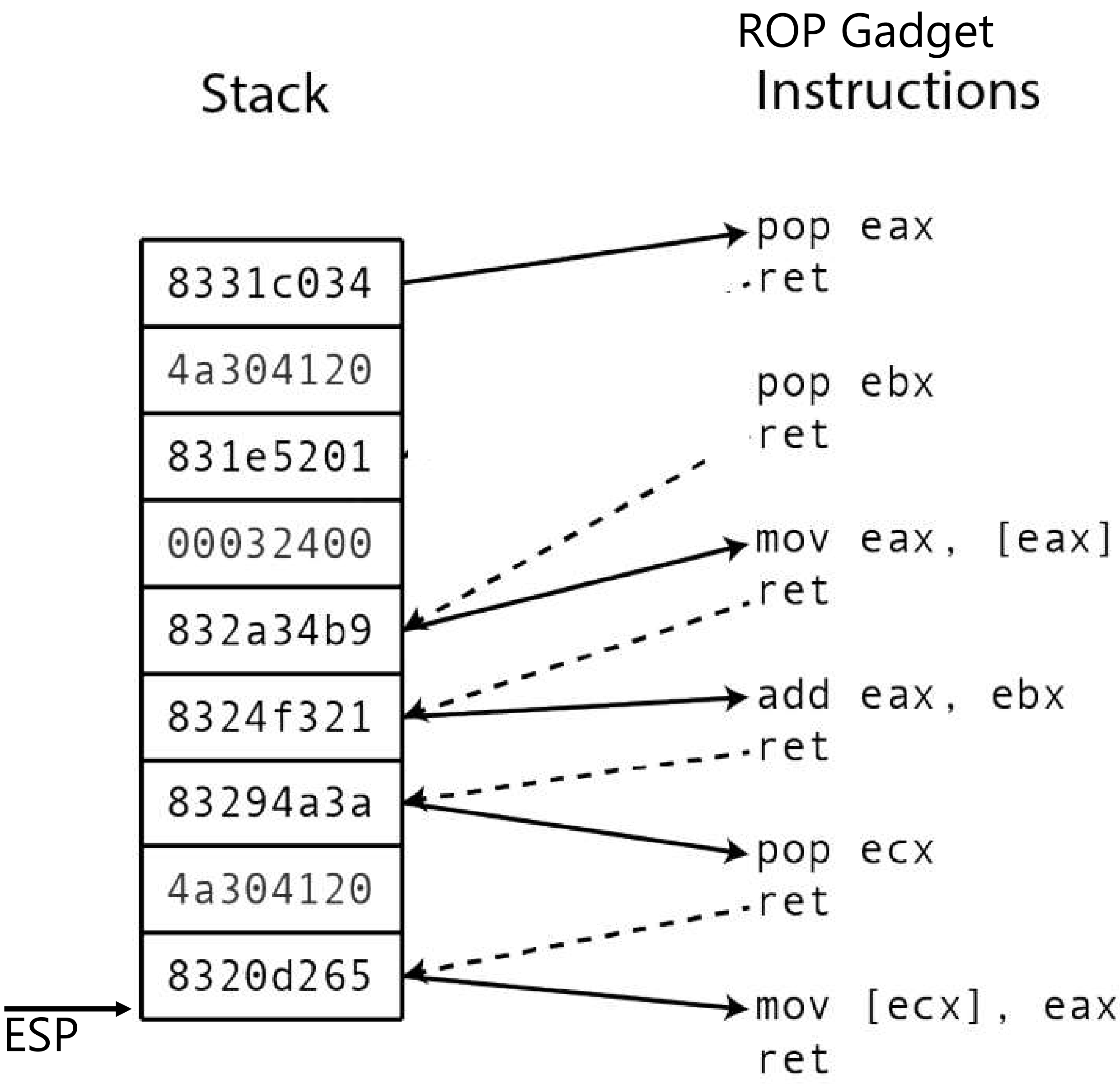
ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

ROP: An Example



ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

ROP: An Example



EAX := 0x4a304120

EBX := 0x00032400

EAX := *(0x4a304120)

EAX := *(0x4a304120)+ 0x32400

ECX := 0x4a304120

*(ECX) := EAX

The net effect is *(0x4a304120) = *(0x4a304120) + 0x00032400

ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

Recap: The ROP Attack Procedure

- Attacker's goal: Execute a particular exploit code of its choice
- The attackers pre-identifies certain useful instruction patterns called "ROP gadgets" in the executable section of the program
- Procedure:
  - The attacker corrupts the stack in a particular way (see later)
  - Hijack control – Cause the program to Jump to address A
  - At address A, we have a "ROP gadget" code:
    1. Instruction (I) at address A executes the first instruction of the payload
    2. The next instruction after (I) is a **ret** instruction
    3. The **ret** instruction will jump to address B, by reading B from the stack.
  - The step 3 above will cause the program to repeat Steps 1-3 (with a new value B instead of A) recursively.

ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

How many ROP Gadgets In Programs?

Binary	PSHAPE	rp++	ropper	ROPgadget
firefox$_W$	6,709	6,182	5,445	6,259
iexplore$_W$	928	888	836	888
chrome$_W$	64,372	58,890	52,991	59,969
mshtml$_W$	1,329,705	1,239,403	1,099,466	1,242,616
jfxwebkit$_W$	1,172,718	1,076,350	960,091	1,086,061
chromium$_L$	5,358,283	5,159,712	4,579,388	5,130,856
apache2$_L$	24,164	22,722	18,061	22,875
openssl$_L$	6,978	6,829	5,377	6,845
nginx$_L$	26,314	25,700	21,081	25,245

(a) Number of extracted gadgets

Function	PSHAPE	ropper	ROPgadget
$W_{VirtualProtect}$	2/4	-	-
$L_{mprotect}$	4/4	1/4	1/4
L_{mmap}	3/3	-	-

(b) Number of gadget chains

Table 2: (a) Number of gadgets found by each tool on the given binaries, as determined by our evaluation. (b) It is possible to build chains to `mprotect` for all four Linux binaries, line mprotect shows how many of those chains each tool creates. For `mmap`, only three of the Linux binaries have the necessary gadgets to build a chain and this line shows how many of those each tool can create. Chains to `VirtualProtect` exist in four out of the five Windows binaries, this line shows how many of them each tool creates. A dash indicates that the tool does not support calling a function that requires the tool to initialize the required number of arguments. In (a) and (b), L denotes Linux and W, Windows.

[PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution]

Summarizing Code Reuse Attacks

- Outcome 2: Code Reuse
  - ***Definition****: A memory exploit that hijacks control to <u>jump to attacker's controlled* ***code address***</u>
- Requirements for Code Reuse
  - Req 1: Have Attack Payload Be Executable
  - Req 2: Divert control-flow to payload

Beyond Control-flow Hijacking: Data-oriented Attacks

Data-oriented Attacks

- Requirements for Data-oriented attacks
  - ~~Req 1: Write Attack Payload in memory~~
  - ~~Req 2: Have Attack Payload Be Executable~~
  - ~~Req 3: Divert control-flow to payload~~

- Insight: Simply manipulate **non-control data**

Data-Oriented Exploits

- State-of-the-art: Corrupt security-critical data
  - leave control flow as the same
  - Exhibit "significant" damage

```
// set root privilege
seteuid(0);
......
// set normal user privilege
seteuid(pw->pw_uid);
// execute user's command
```

Wu-ftpd *setuid* operation*

```
//0x1D4, 0x1E4 or 0x1F4 in JScript 9,
//0x188 or 0x184 in JScript 5.8,
safemode = *(DWORD *)(jsobj + 0x188);
if( safemode & 0xB == 0 ) {
      Turn_on_God_Mode();
}
```

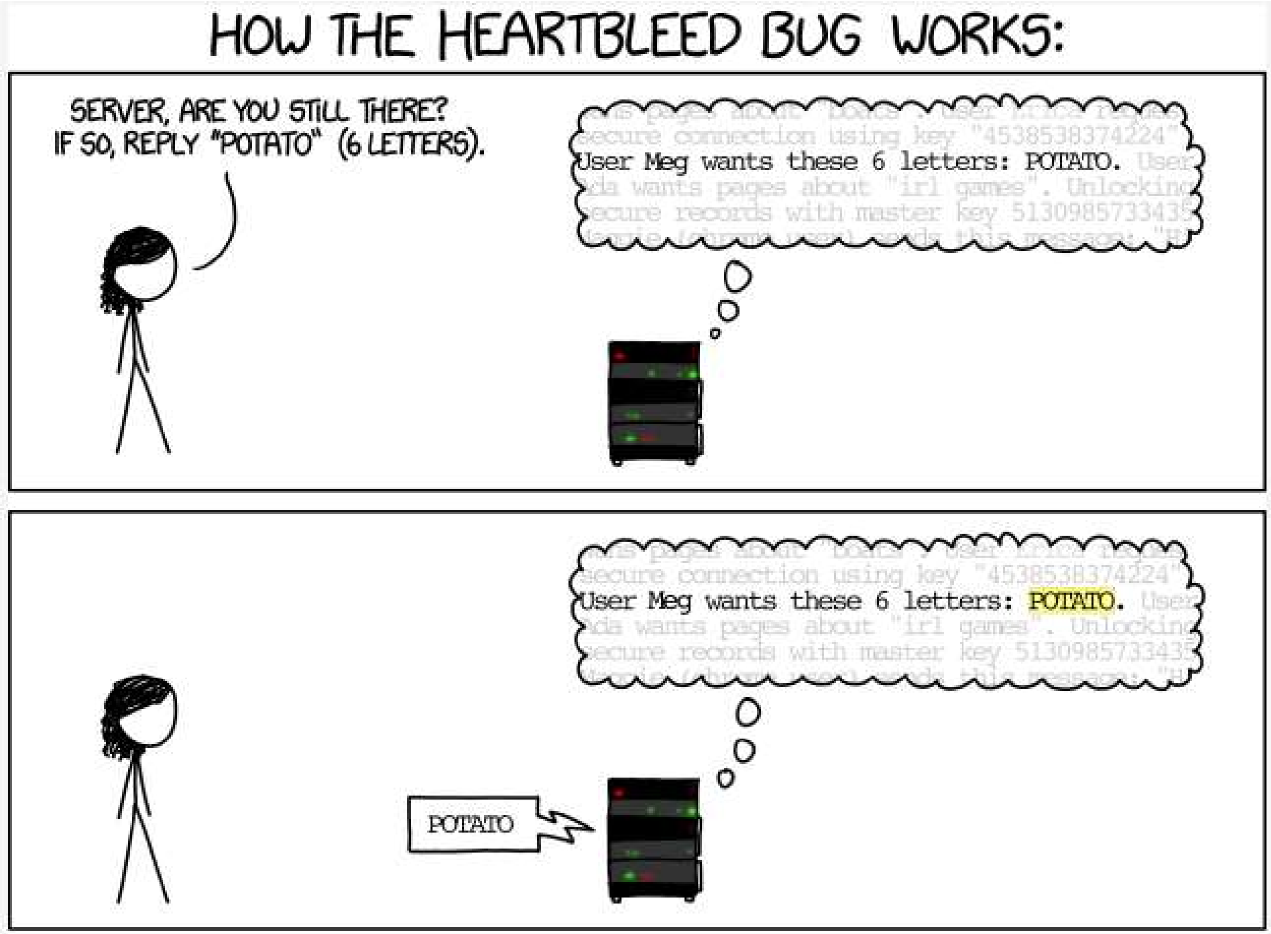
IE *SafeMode* Bypass+

+ Yang Yu. Write Once, Pwn Anywhere. In Black Hat USA 2014

* Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In USENIX 2005.

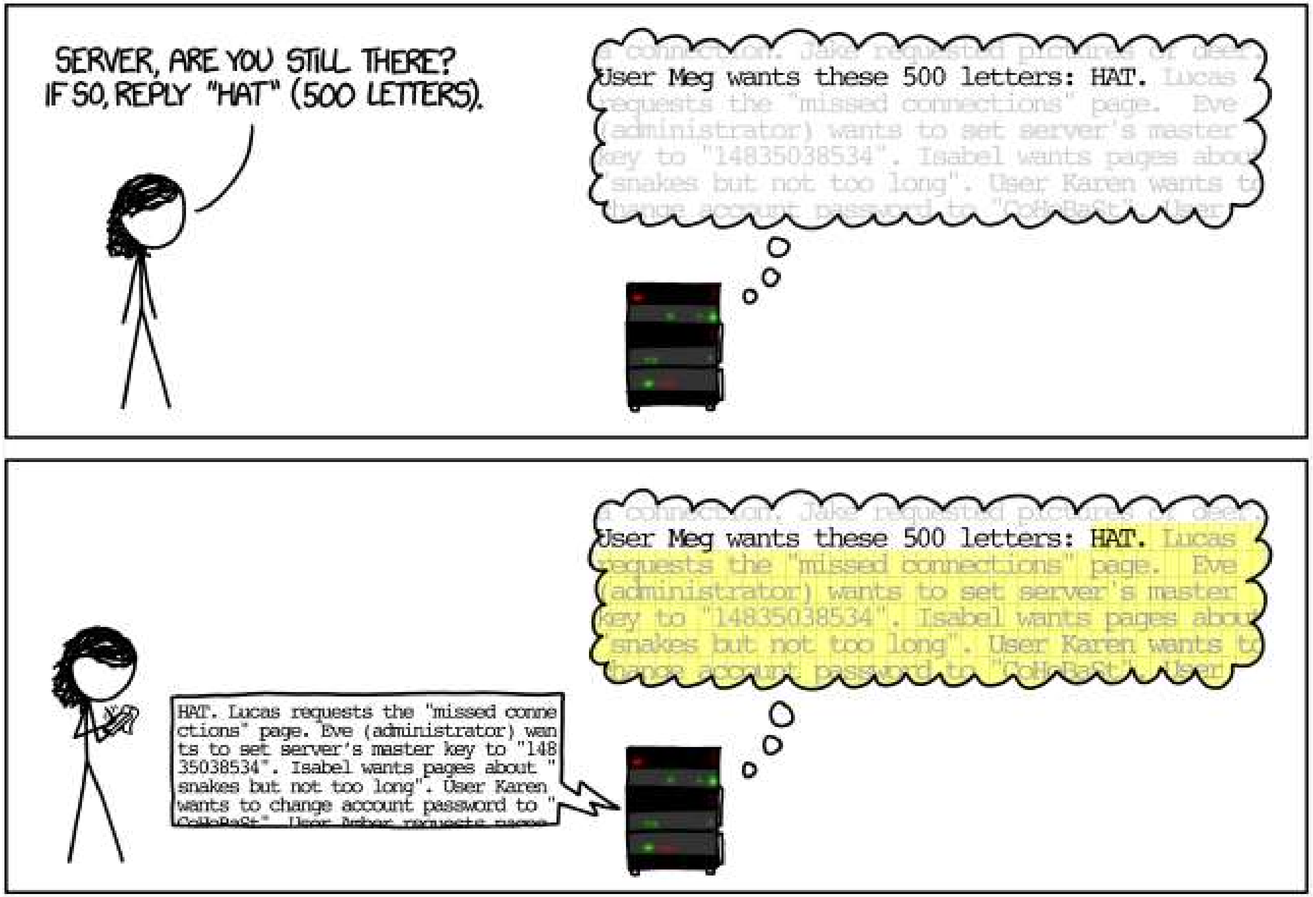
Data-oriented attacks w/o any memory corruption

- Do I need to corrupt anything for DOA?
- No! An example: Heartbleed



Data-oriented attacks w/o any memory corruption

- Do I need to corrupt anything for DOA?
- No, An example: Heartbleed



Data-oriented attacks w/o any memory corruption

- Do I need to corrupt anything for DOA?
- No! An example: Heartbleed

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
```

```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```

Key Takeaways & Summary

- Vulnerabilities vs. Exploits
  - Weakness (Flaw) vs. using it for a particular goal
- Exploit Types:
  - Control-flow hijacking vs. Data-oriented
- Attackers can achieve a variety of attacks
- C/C++: weak type safety, no memory safety
- Hardware does <u>not</u> give memory & type safety

End of Segment