CS5231: Systems Security

Lecture 3: Type Errors, Code Reuse, and Data-oriented Attacks

Temporal Memory Errors: Use-after-free

Lifetime & Scope of Variables

- **Scope**:
  - Region of code where a variable can be accessed
  - E.g. Global, Function-local, Heap (dynamic)
- **Lifetime**:
  - Portion of program execution during which storage is guaranteed
  - E.g. Auto vs. static

Are Programming Language Abstractions

- Not instruction set / hardware abstractions

```
1. int z=0;
2. int g(int x, int y) {
3.      char* buf;
4.      buf = malloc (50);
5.      scanf(“%s”, buf);
6.      free (buf);
7. …
8. }
```

Variable	Scope	Lifetime
z	Global, Line 1-8	Throughout the program
x	Local, Line 3-7	Execution of g
y	Local, Line 3-7	Execution of g
buf	Local, Line 3-7	Execution of g
	Constant Literal, Line 5	Execution of Line 5 (undefined?)
“%s”	Heap, Line 4-7	Line 5-6
*buf		

Question

```
int foo() {
   int *p;
   {
      int x = 5;
      p = &x;
   }
   return *p;
}
```

What will this program return?

Answer:
Undefined behavior as per C11 standard

Why?

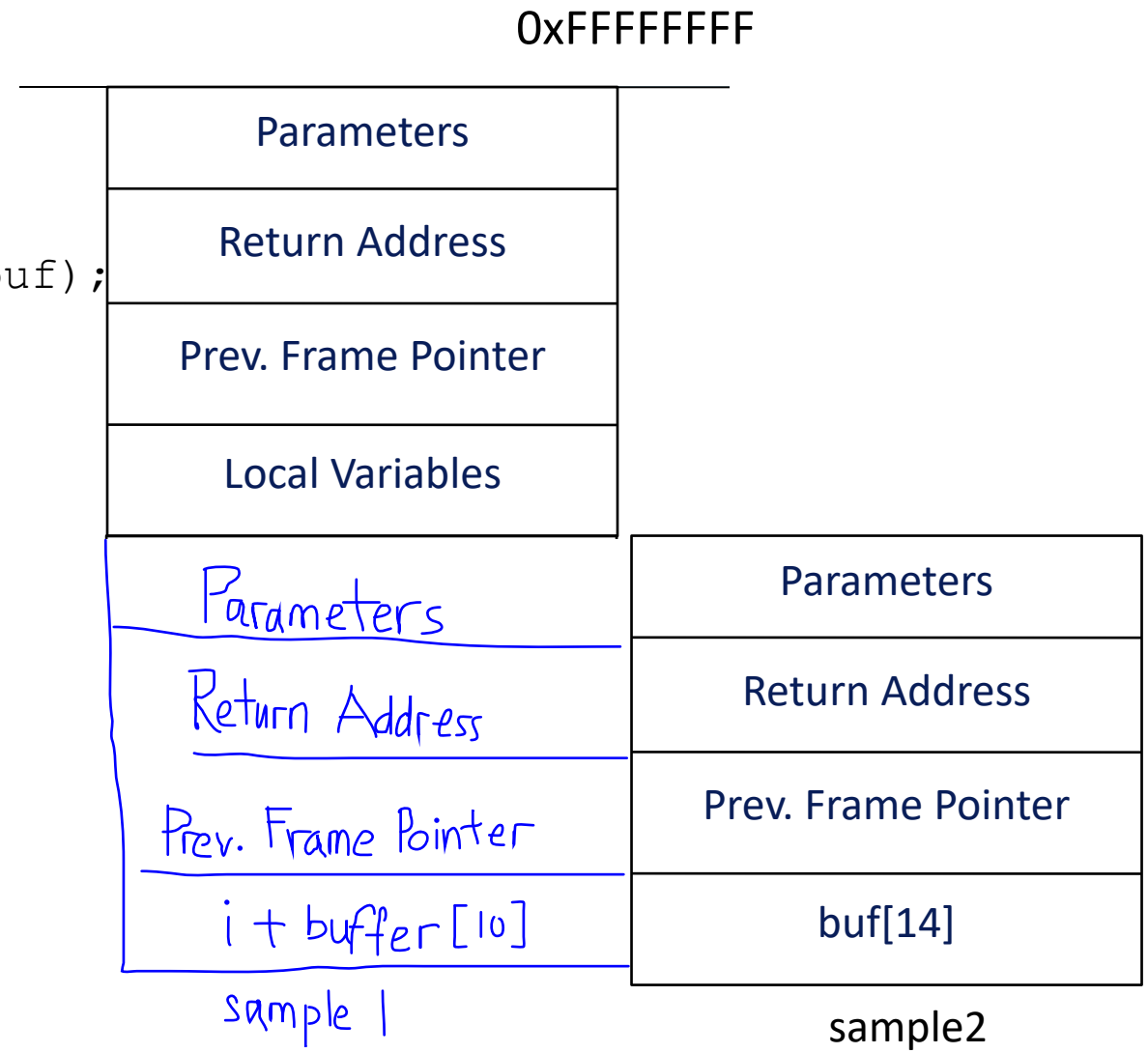
- The lifetime of x is within the inner {}
- The compiler can choose to remove the storage for “x”
- The pointer p is in scope at the last line
- The pointed-to object, however, is accessed out of scope!

Stack-based Temporal Memory Error

```
void sample1(void)
{
        int i;
        char buffer[10];
        gets(buffer);
        return;
}
```

```
void sample2(void)
{
        char buf[14];
        printf("%s\n", buf);
        return;
}
```

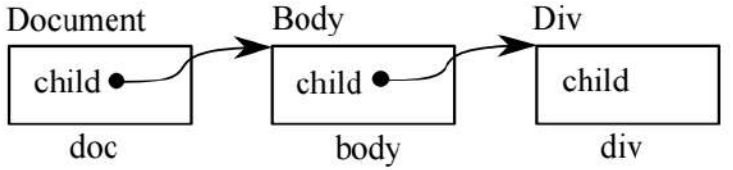
```
main()
  {
        sample1();
        ...
        sample2();
  }
```



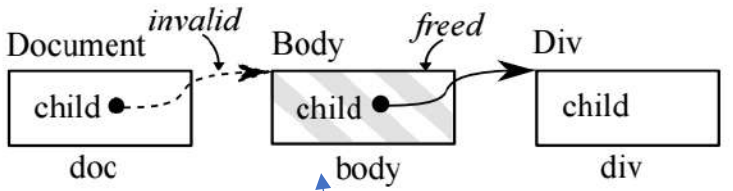
Heap-based Temporal Memory Errors
Use-after-free

Temporal Mem Error: When program accesses mem. beyond its valid lifetime!

```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
```



(a-b) objects are allocated and linked



(c-d) body is freed (so dangled), and doc reads the invliad memory

If accessing doc->child

Dangling Pointers [NDSS'15]

Type Errors (I): Integer Overflow

A Puzzle: Is this code free from buffer overflow?

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);

      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```


OWASP – Integer Overflow Integer Overflow (Sec 1,2,3)

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);   // Range? [0, 255]
                                              Or [-128, 127]
      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

-100

OWASP – Integer Overflow Integer Overflow (Sec1,2,3)

Implicit Type Conversions

Operation	Operand Values	Overflow / Underflow?
SUB – 32 Bit signed	-2^31 – 2^31	Underflow
ADD – 32 Bit Unsigned	2^32 + 2^32	Overflow
MUL – 32 Bit Unsigned	2^20 * 2^20	Overflow

The actual range of values for each type is compiler / machine –dependent. When operands of different types widths are used, C99 standard **automatically (implicitly) converts types**.

Type Promotions:
bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

Signed / Unsigned Coercions:
Defined by the C standard

https://stackoverflow.com/questions/50605/signed-to-unsigned-conversion-in-c-is-it-always-safe
https://www.geeksforgeeks.org/type-conversion-c/

Why doesn't hardware complain?

- Hardware: Arithmetic doesn't distinguish signed and unsigned integers --- works for both

Unsigned $\sum_{i=0}^{n-1} x_i 2^i$

$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ Signed

$i=0$

An advantage of two's complement is that signed and unsigned addition can be performed using the same operation. The same is true for subtraction and multiplication. Historically, this was advantageous because fewer instructions needed to be implemented. Also, unlike the *ones' complement* and *sign-magnitude* representations, two's complement has only one representation for zero. A drawback of two's complement is that its range, $-2^{n-1} \ldots 2^{n-1} - 1$, is asymmetric. Thus, there is a representable value, -2^{n-1}, that does not have a representable additive inverse—a fact that programmers can and do forget.

When an n-bit addition or subtraction operation on unsigned or two's complement integers overflows, the result "wraps around," effectively subtracting 2^n from, or adding 2^n to, the true mathematical result. Equivalently, the result can be considered to occupy $n+1$ bits; the lower n bits are placed into the result register and the highest-order bit is placed into the processor's carry flag.

TOSEM'15 paper – section 1,2,3 are relevant

Integer Overflow (TOSEM'15)

Why doesn't hardware complain?

- In C99 on x64, conversions reinterpret the bit representation!

3.2. The Usual Arithmetic Conversions

Most integer operators in C/C++ require that both operands have the same type and, moreover, that this type is not narrower than an `int`. The collection of rules that accomplishes this is called *the usual arithmetic conversions*. The full set of rules encompasses both floating point and integer values; here we will discuss only the integer rules. First, both operands are *promoted*:

> If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

If the promoted operands have the same type, the usual arithmetic conversions are finished. If the operands have different types, but either both are signed or both are unsigned, the narrower operand is converted to the type of the wider one.

If the operands have different types and one is signed and the other is unsigned, then the situation becomes slightly more involved. If the unsigned operand is narrower than the signed operand, and if the type of the signed operand can represent all values of the type of the unsigned operand, then the unsigned operand is converted to signed. Otherwise, the signed operand is converted to unsigned.

These rules can interact to produce counterintuitive results. Consider this function:

```
int compare (void) {
  long a = -1;
  unsigned b = 1;
  return a > b;
}
```

For a C/C++ implementation that defines `long` to be wider than `unsigned`, such as GCC for x86-64, this function returns zero. However, for an implementation that defines `long` and `unsigned` to have the same width, such as GCC for x86, this function returns one. The issue is that on x86-64, the comparison is between two signed integers, whereas on x86, the comparison is between two unsigned integers, one of which is very large. Some compilers are capable of warning about code like this.

C/C++ don't define exactly for all cases!

Where they do define, the behavior is implementation (size) specific to compilers, and varies by architecture!

Int Overflow (TOSEM)

Real-life Incidents Due to Integer Bugs

The New York Times Magazine

Little Bug, Big Bang

By James Gleick

Dec. 1, 1996

IT TOOK THE European Space Agency 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space.

CVE-2018-10299 Detail

Memory Safe ≠ Type Safe

Current Description

An integer overflow in the batchTransfer function of a smart contract implementation for Beauty Ecosystem Coin (BEC), the Ethereum ERC20 token used in the Beauty Chain economic system, allows attackers to accomplish an unauthorized increase of digital assets by providing two _receivers arguments in conjunction with a large _value argument, as exploited in the wild in April 2018, aka the "batchOverflow" issue.

Source: MITRE

Hide Analysis Description

From Vulnerabilities To Memory Exploits

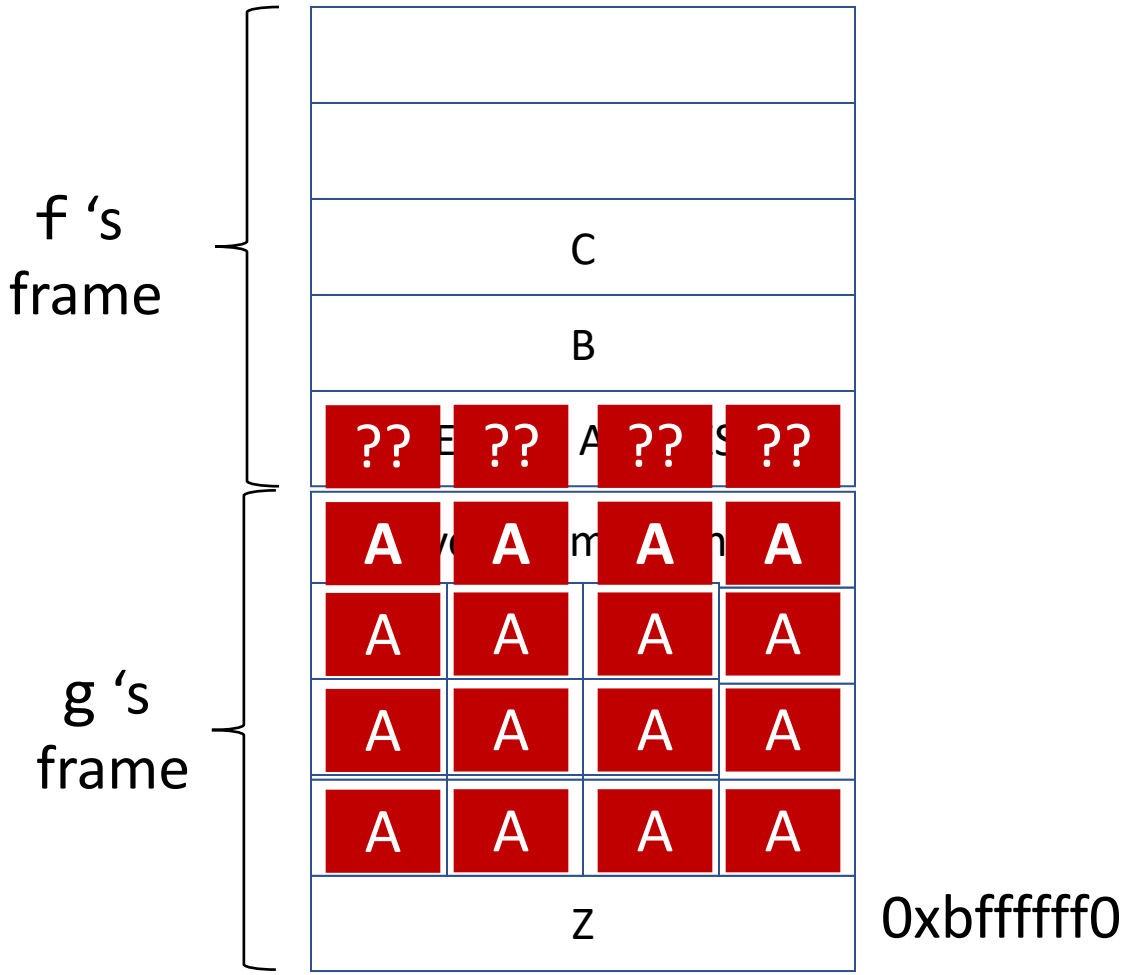
Recall: Control-flow Hijacking Exploits - Code Injection

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

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
```



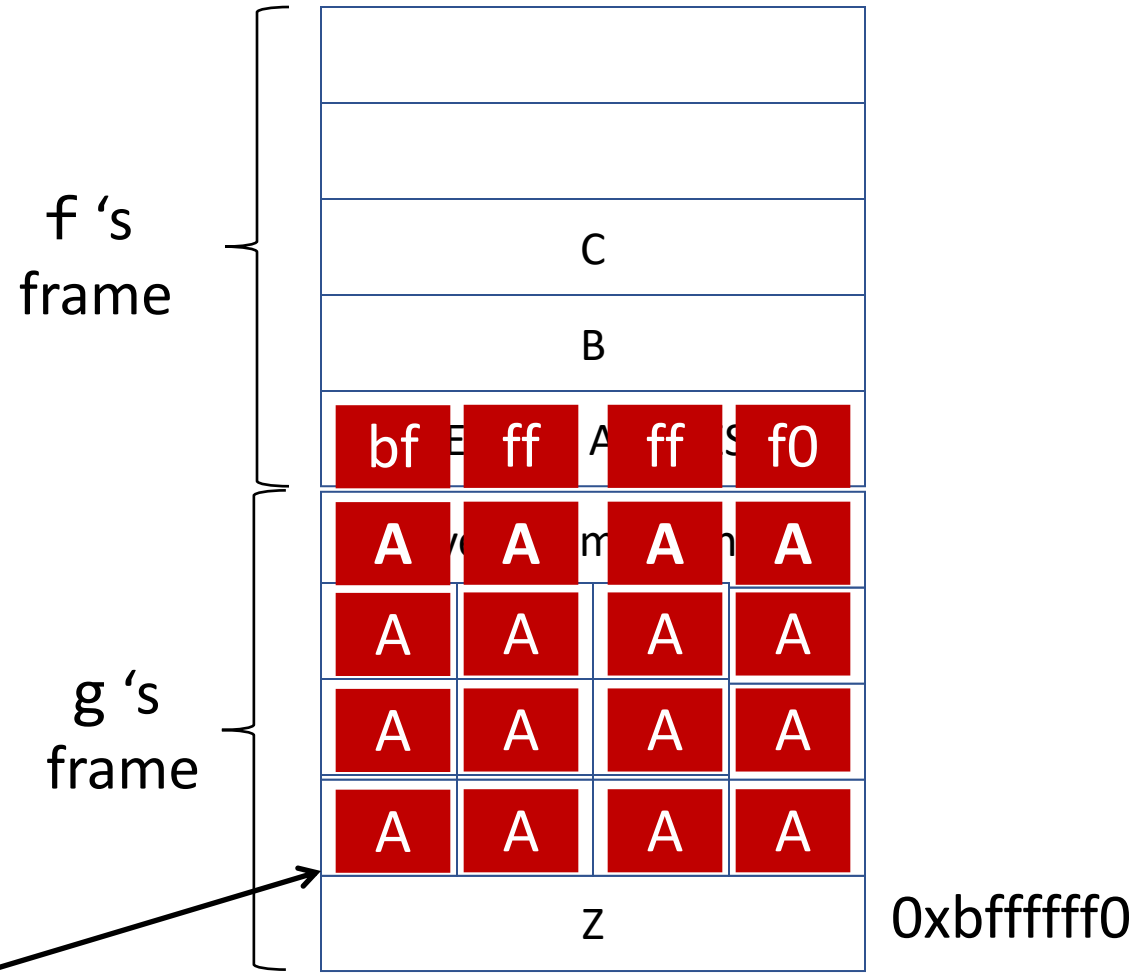
What should be Values for ??

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
```

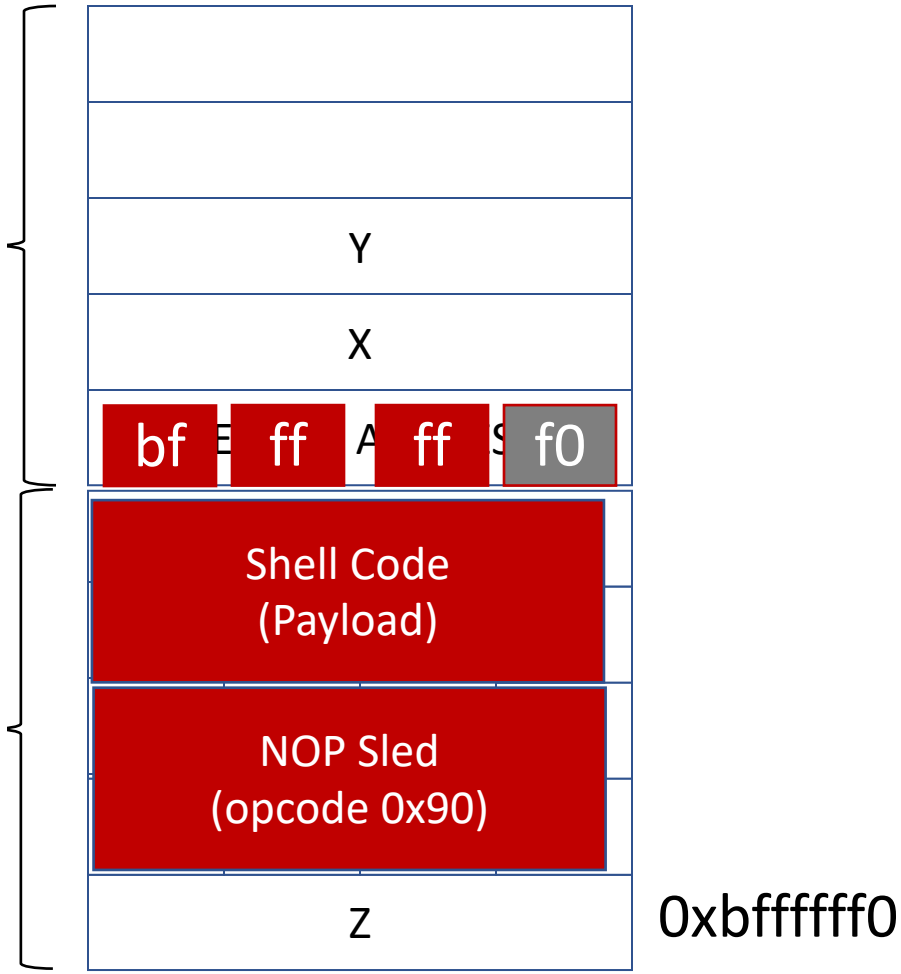
Will return inside attacker’s buffer



What should be Values for A?

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

New Rounds of Control-flow Attacks: Code Reuse

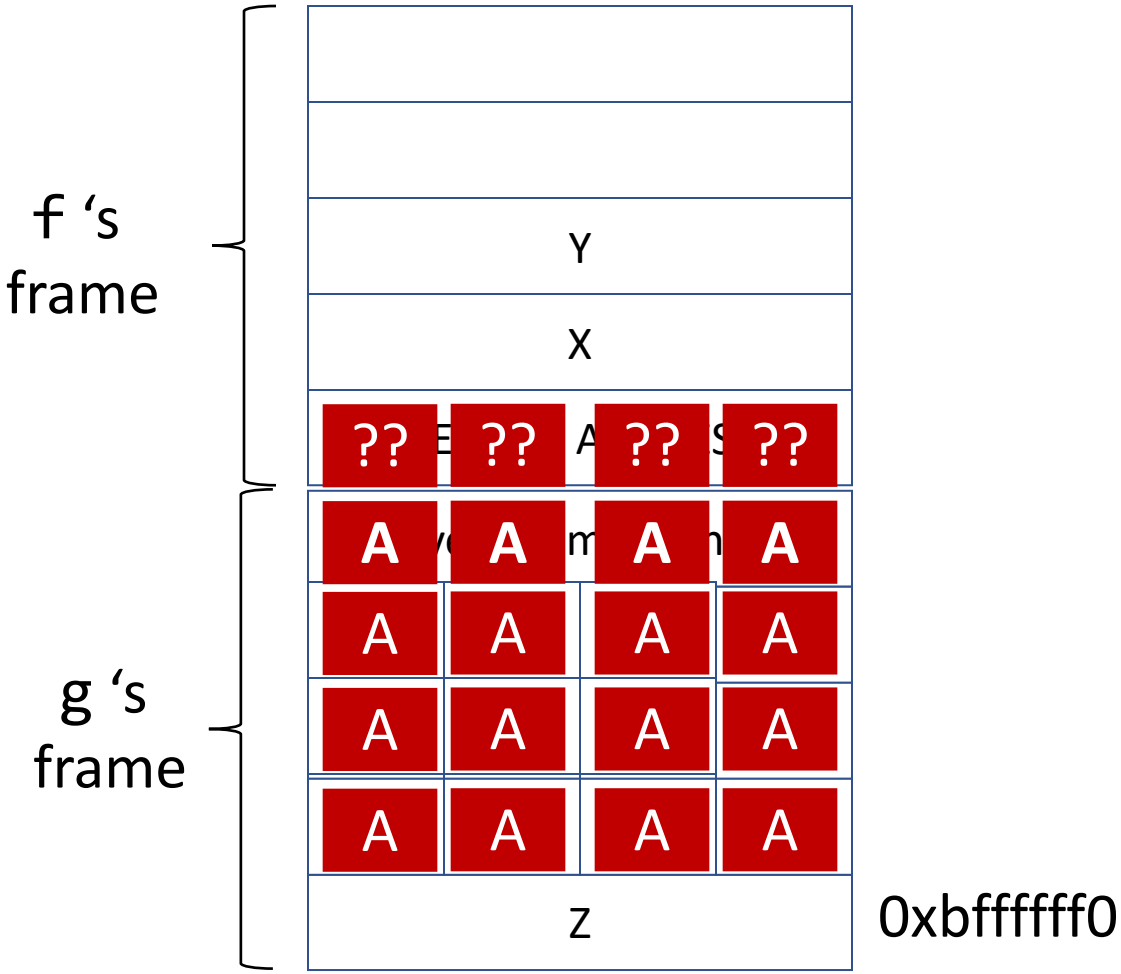
Control-oriented Exploits (II): Code Reuse

- 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 Attack: Return-to-libc

- Attacker hijacks control flow
- Jumps back to the code segment

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```

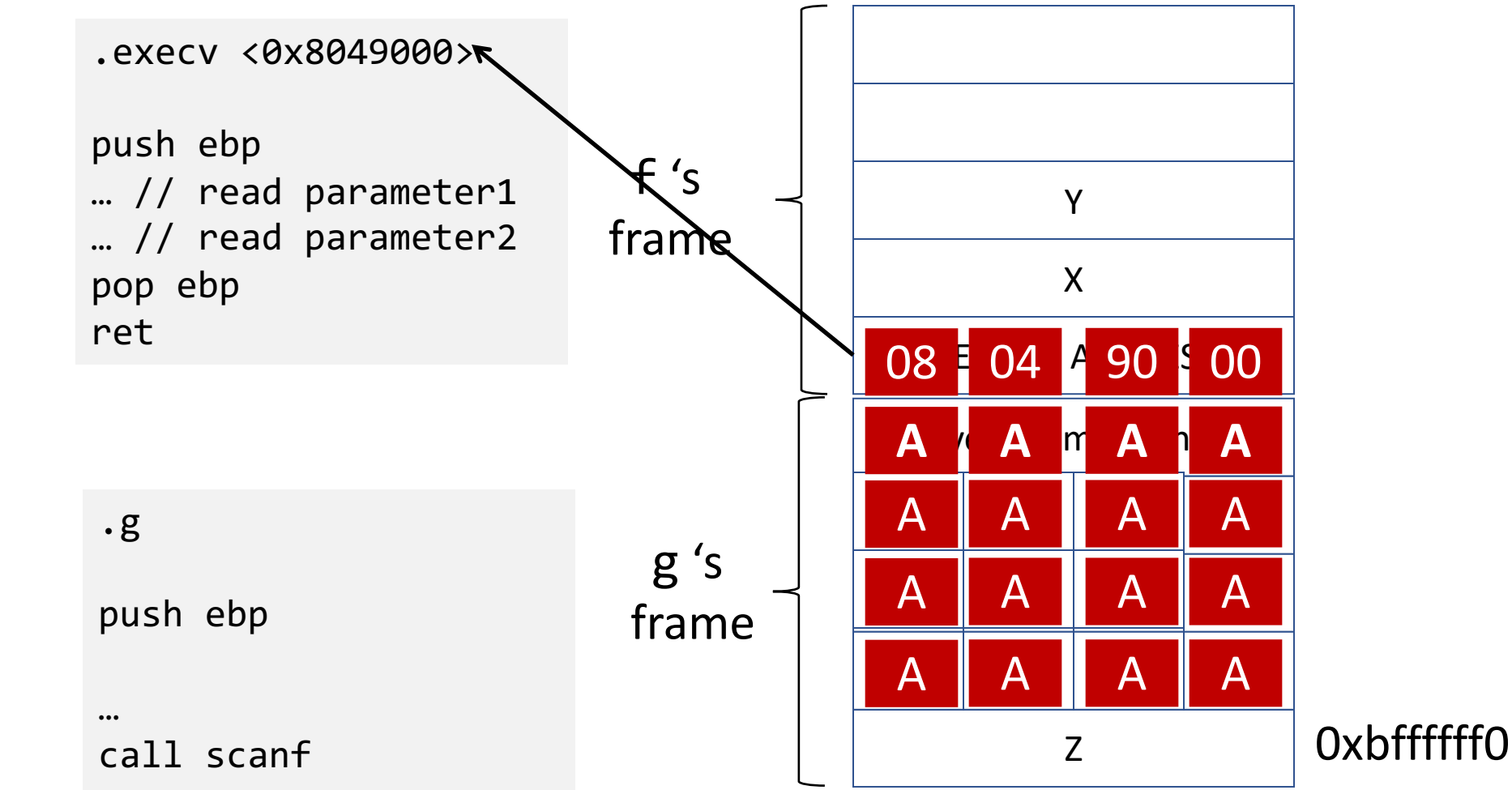


What should you do?

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
```

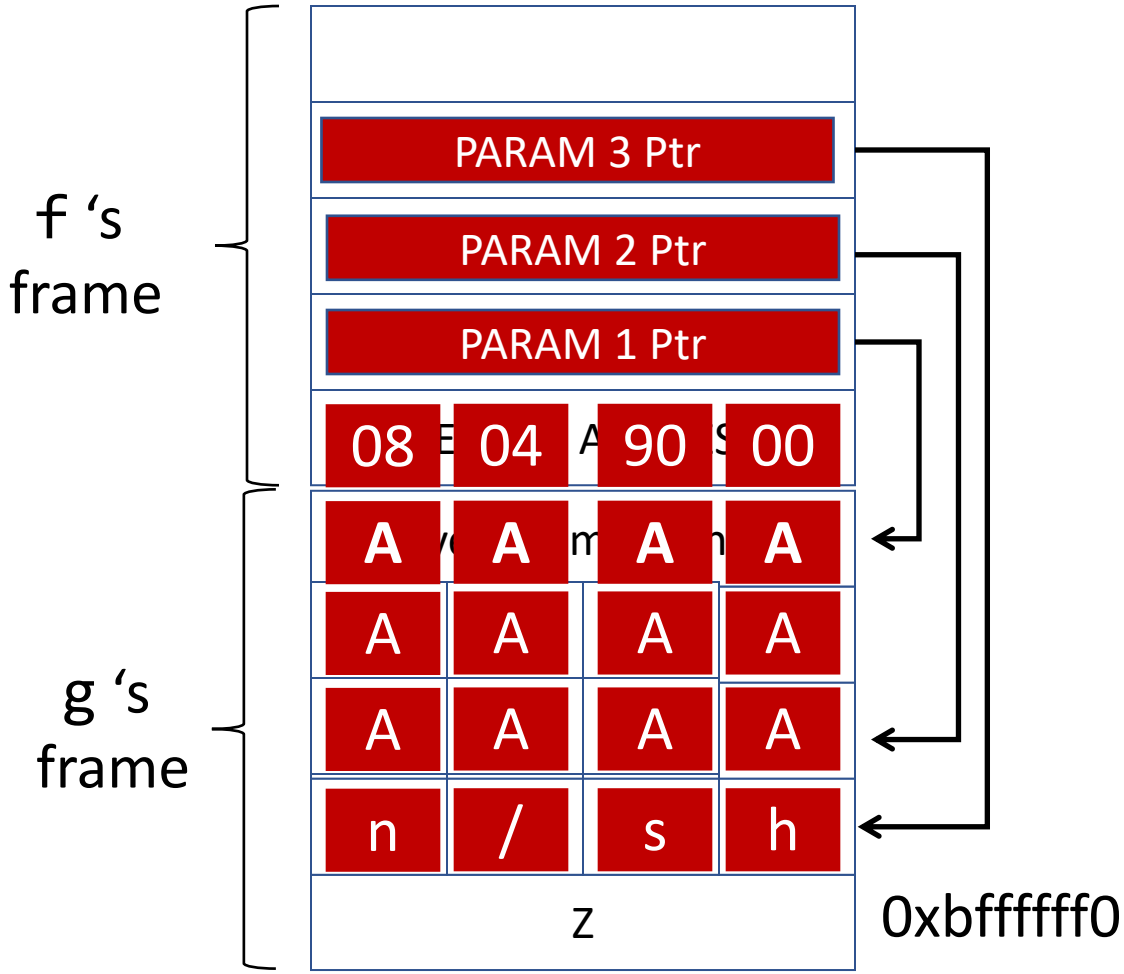


Change return address to code seg.

Code Reuse Attack: Return-to-libc

```
.execv <0x8049000>

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

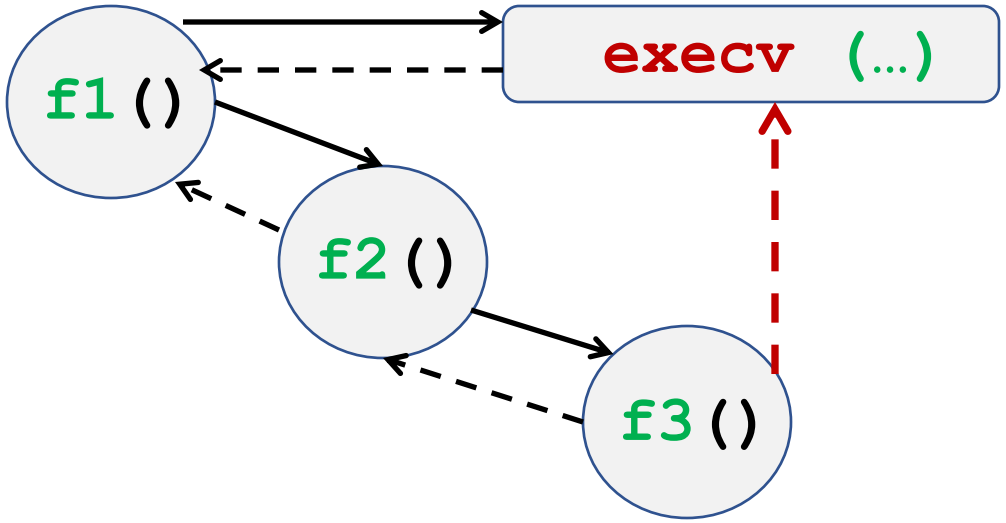
```
.g

push ebp


…
call scanf
…
pop ebp
ret
```



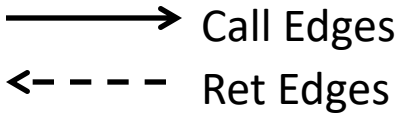
Setup arguments to execv on stack

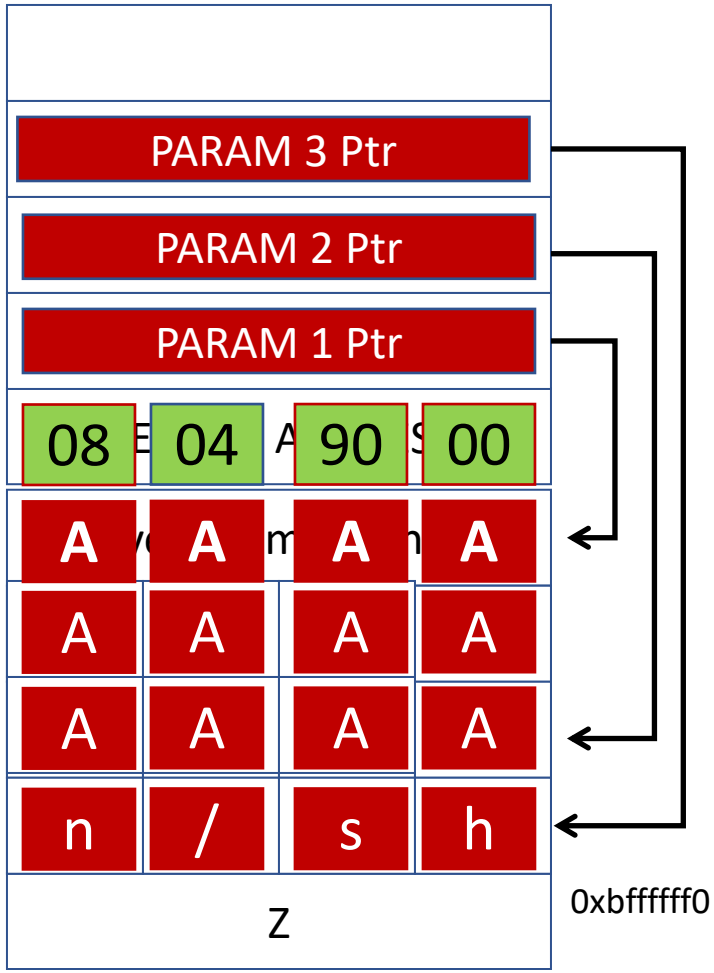
Return-to-Libc

- Introduce new Control Edges



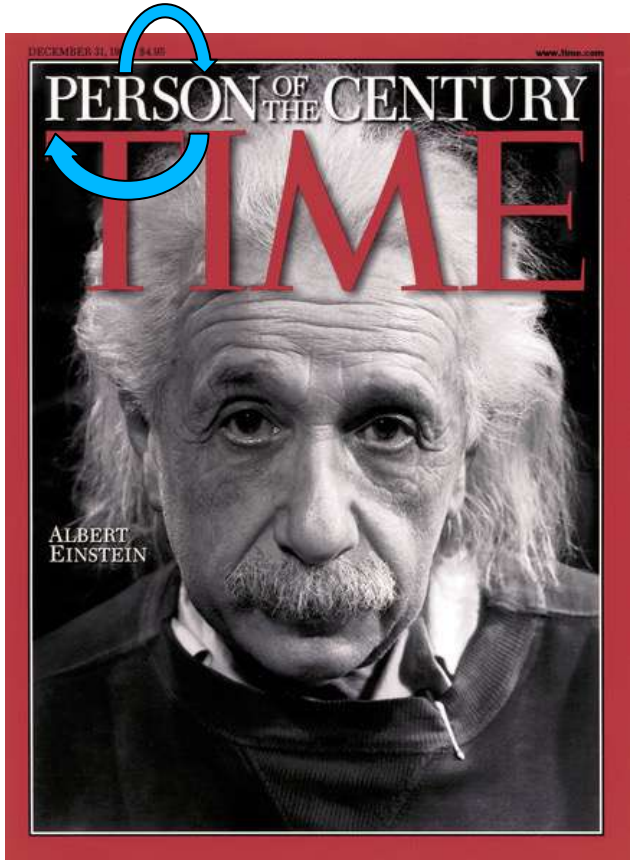
Control Flow Graph





Code Reuse (II): ROP

An Analagy



Can you see:

ROP? SECURITY?

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

- Key Observation:

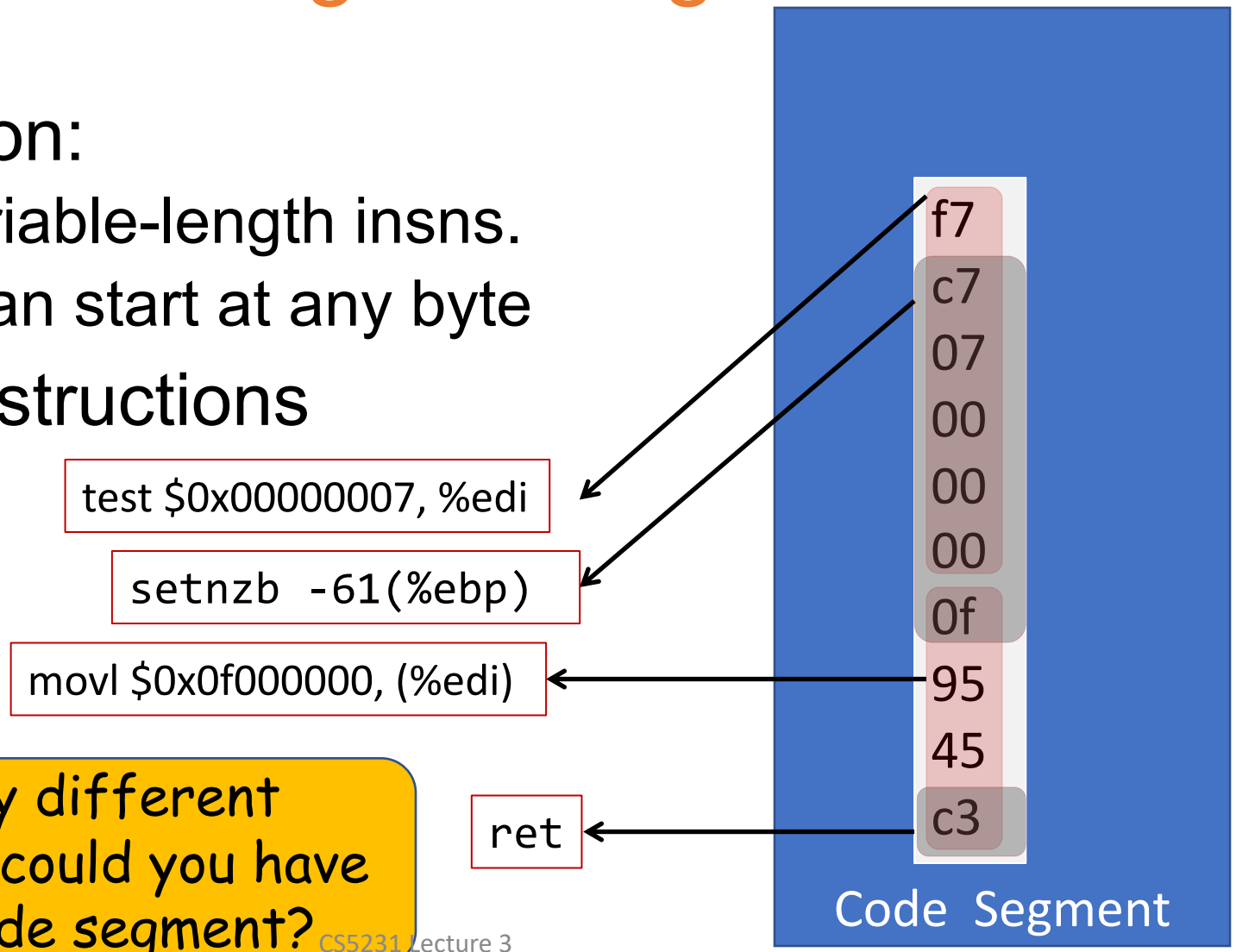
```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)

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

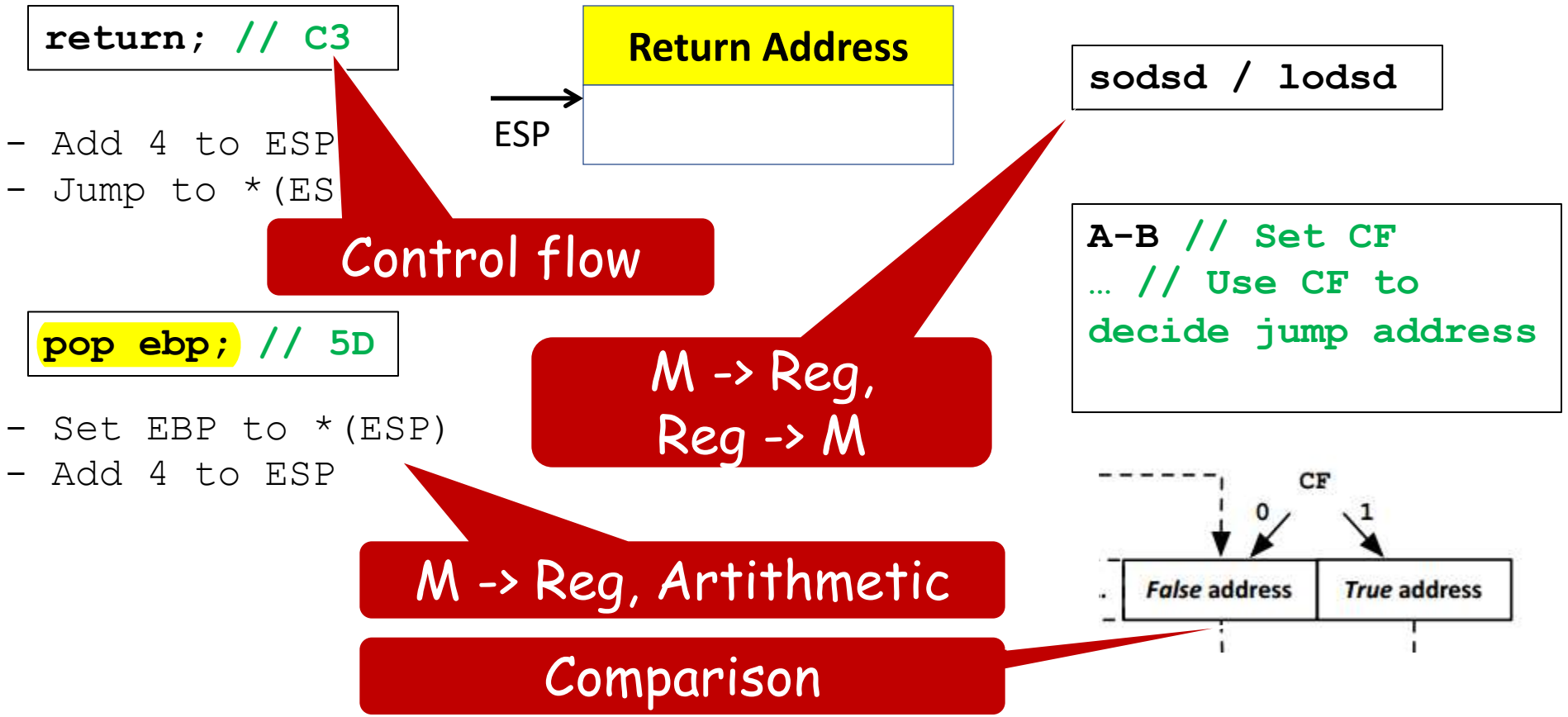
- Key Observation:
  - X86 has a variable-length insns.
  - Instructions can start at any byte
- Overlapping instructions



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

ROP Gadgets

ROP Gadget: Instruction sequence which end a control transfer



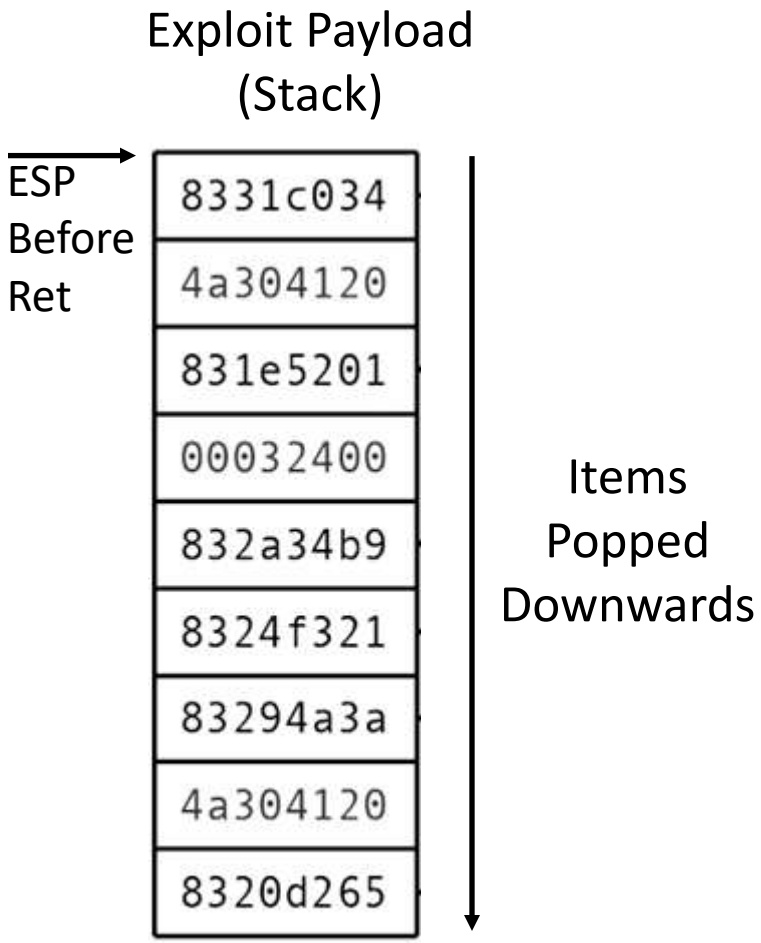
Microgadgets: Size Does Matter In Turing-complete Return-oriented Programming [WOOT'12]

ROP: An Example

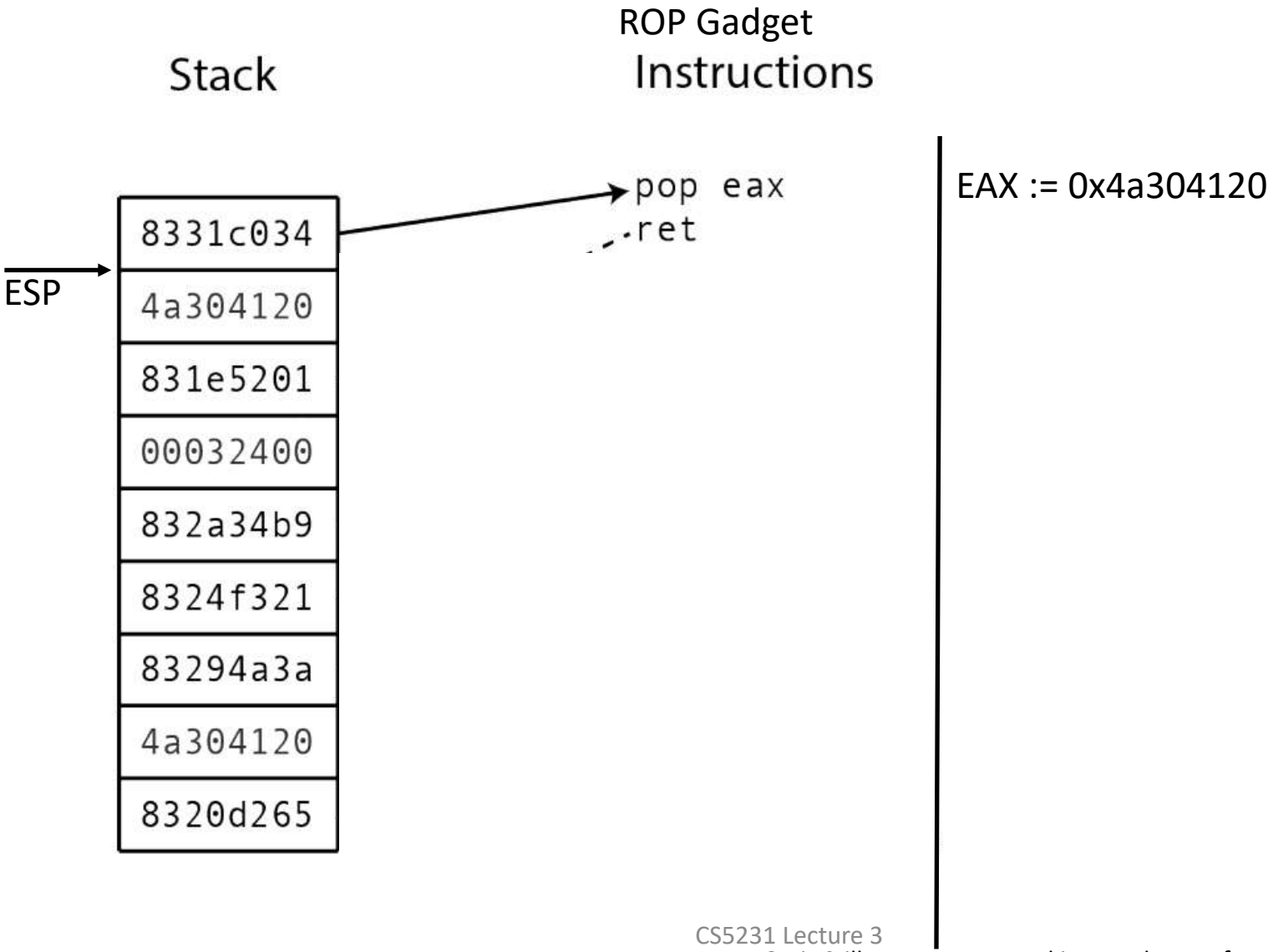
Let's say the attacker's goal is: `*(0x4a304120) = *(0x4a304120) + 0x00032400`

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
```

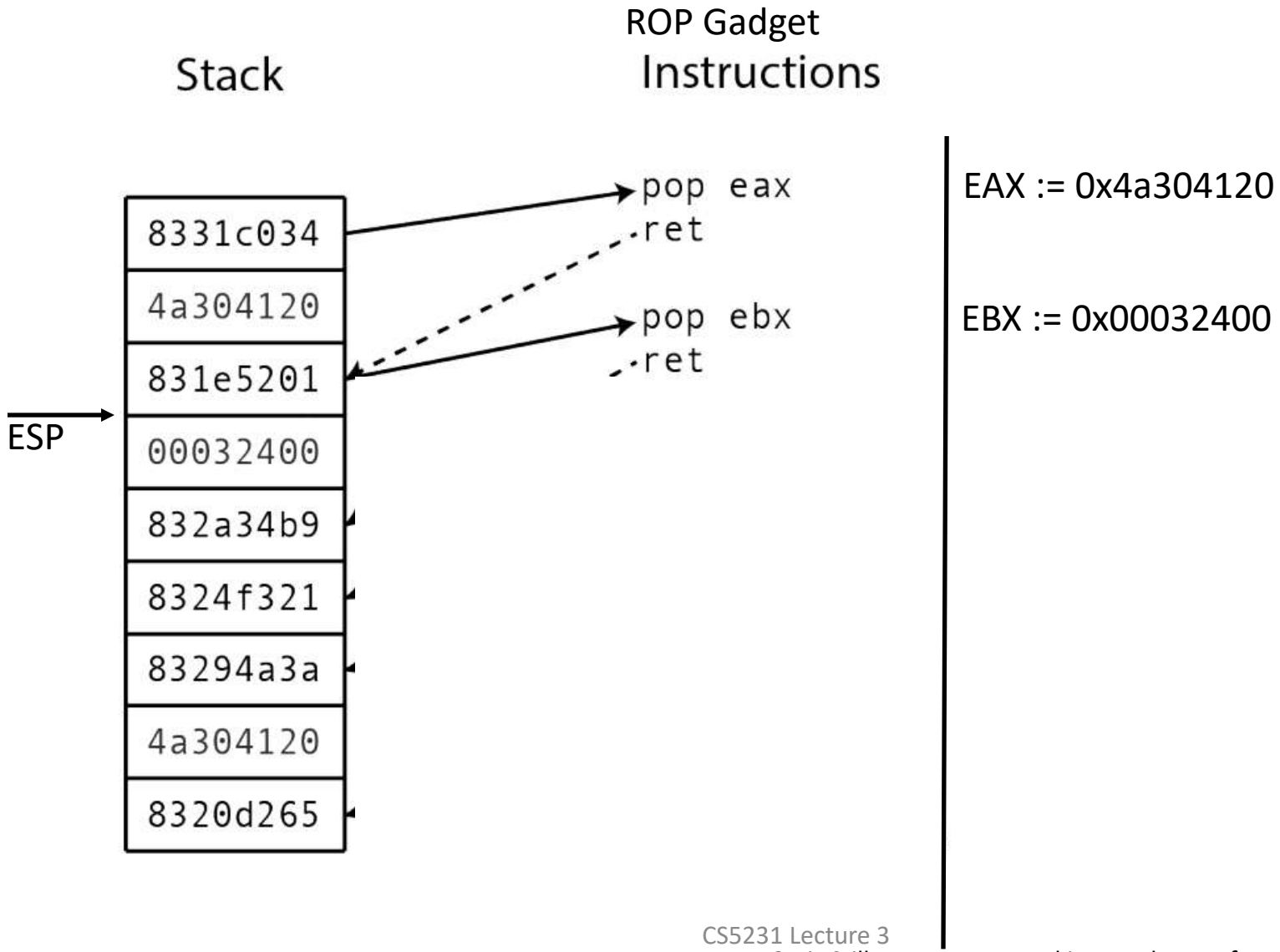


ROP: An Example



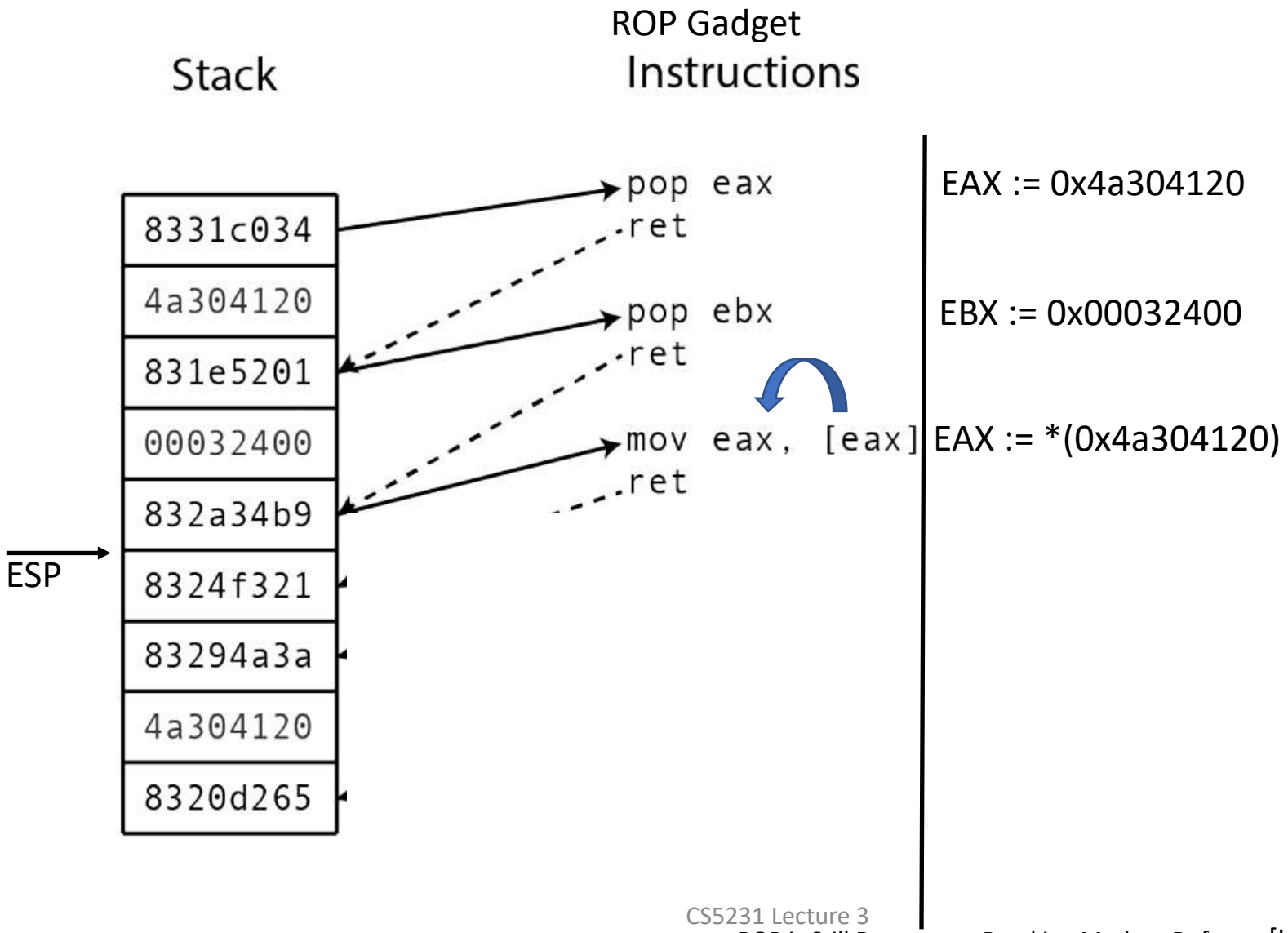
ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

ROP: An Example



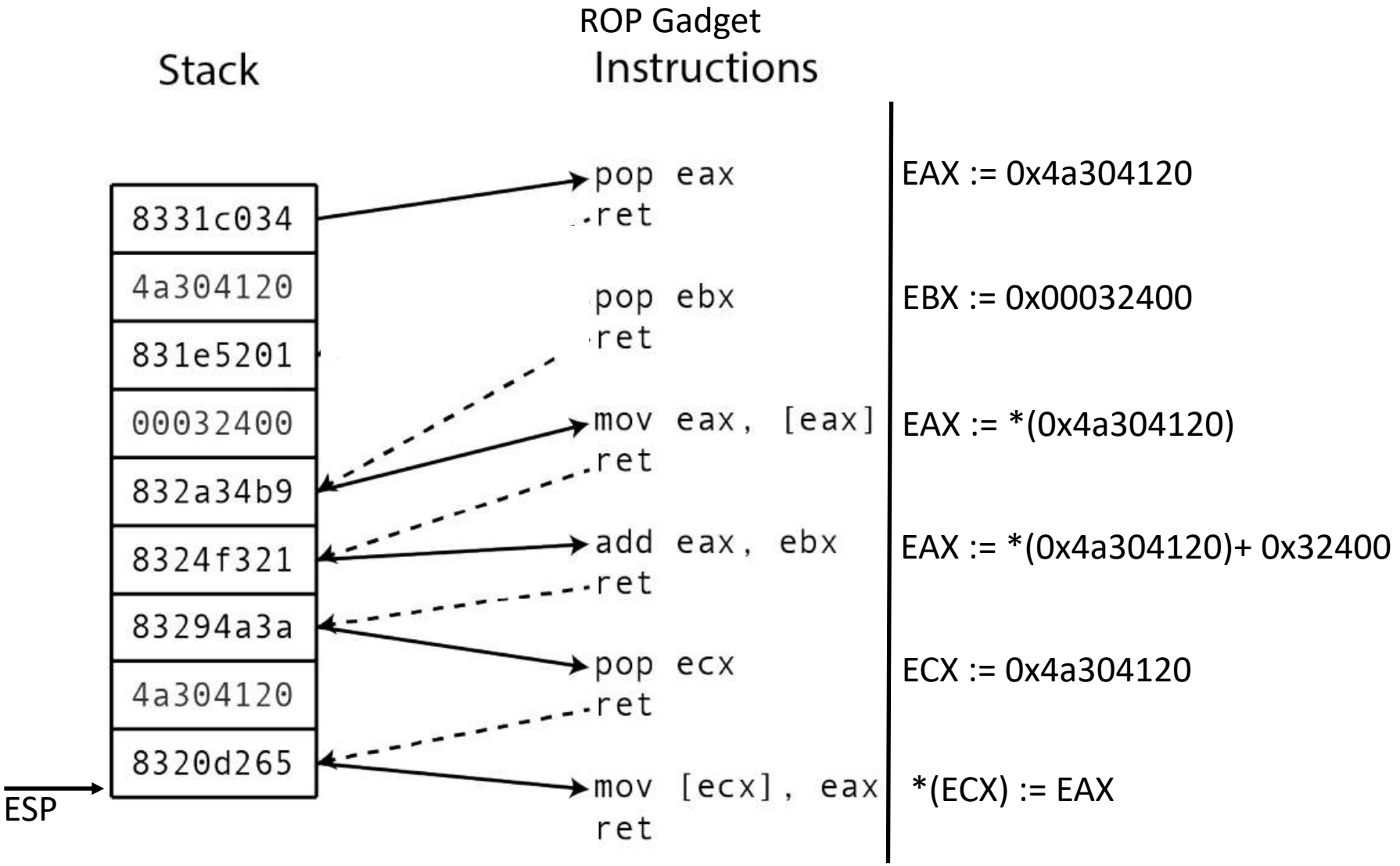
ROP is Still Dangerous: Breaking Modern Defenses[Usenix Security 2014]

ROP: An Example



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

ROP: An Example



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.

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 jump to attacker's controlled* ***code address***
- 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();
}
```

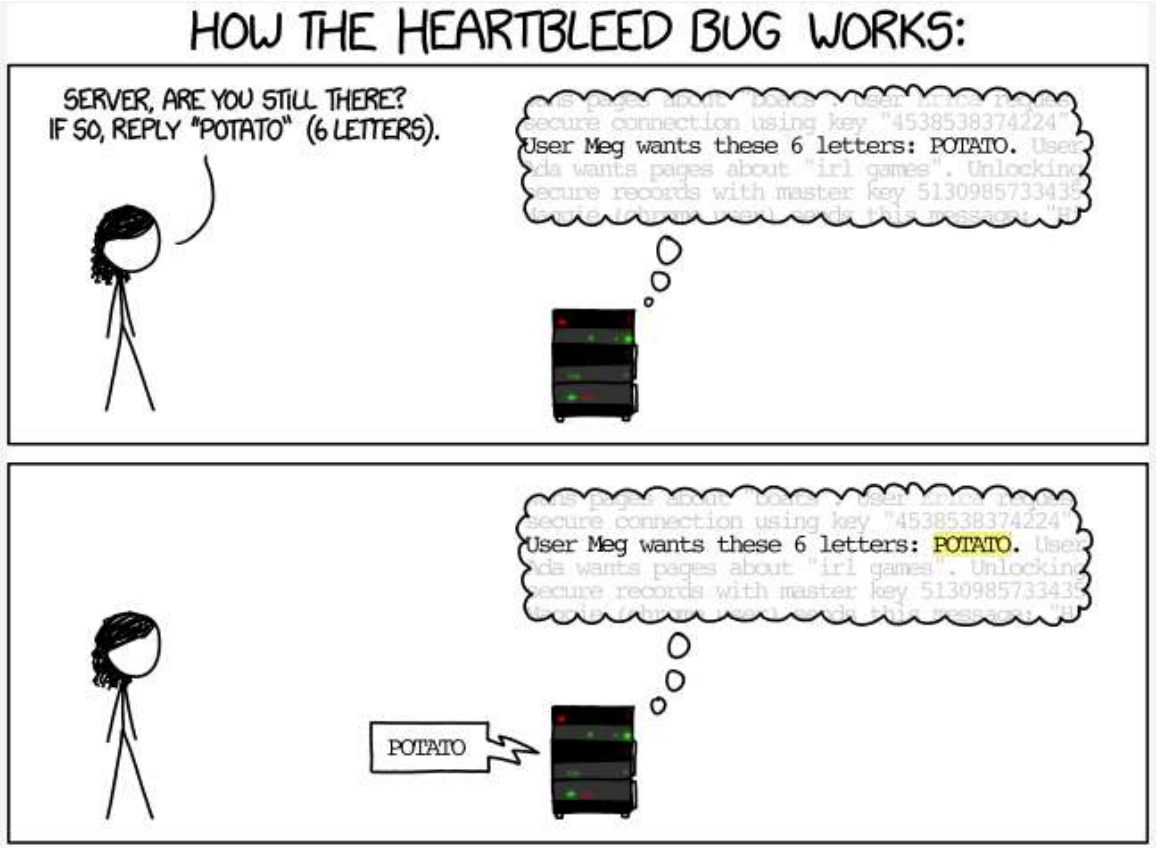
IE *SafeMode* Bypass+

+ 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.

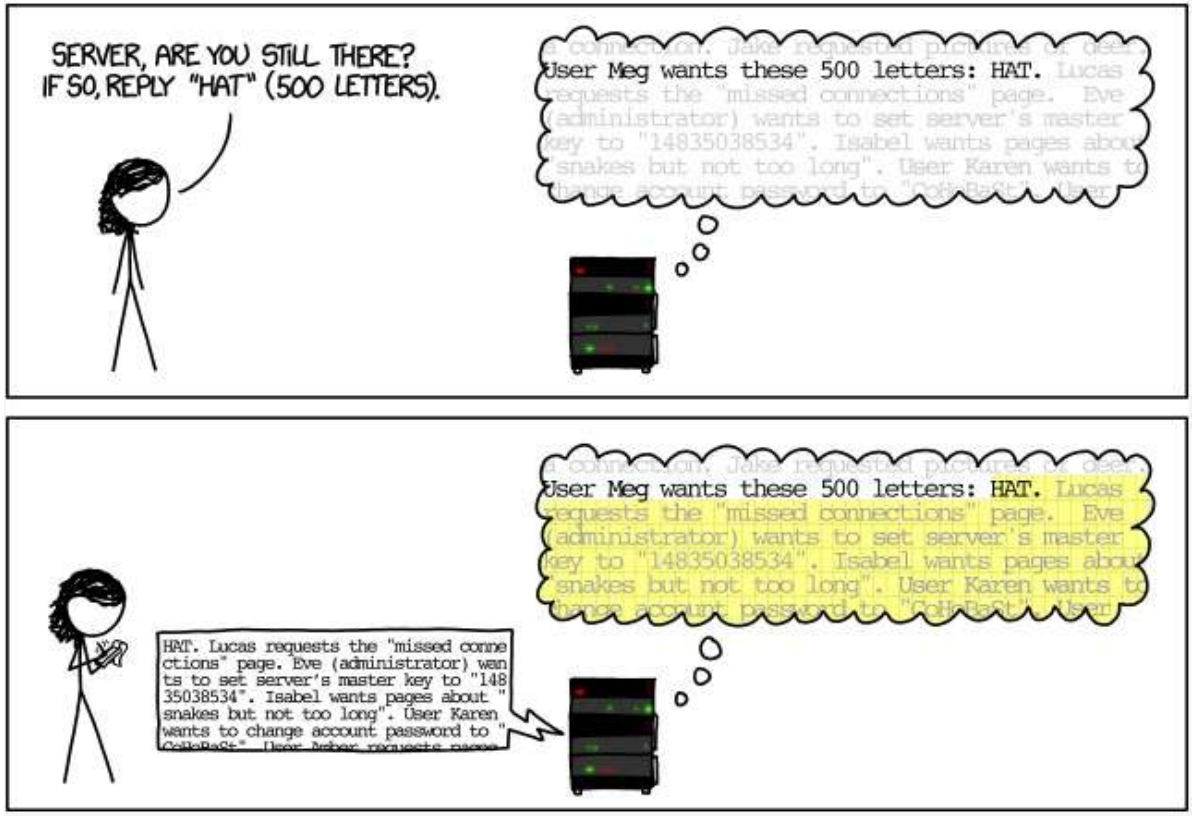
Data-oriented attacks w/o any memory corruption

- 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



Heartbleed Code

- 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 not give memory & type safety