CSE551: Advanced Computer Security

9. ROP & ASLR

Seongil Wi

Department of Computer Science and Engineering

The slide is based on Prof. Sang Kil Cha's lecture slide

Project Checkpoint

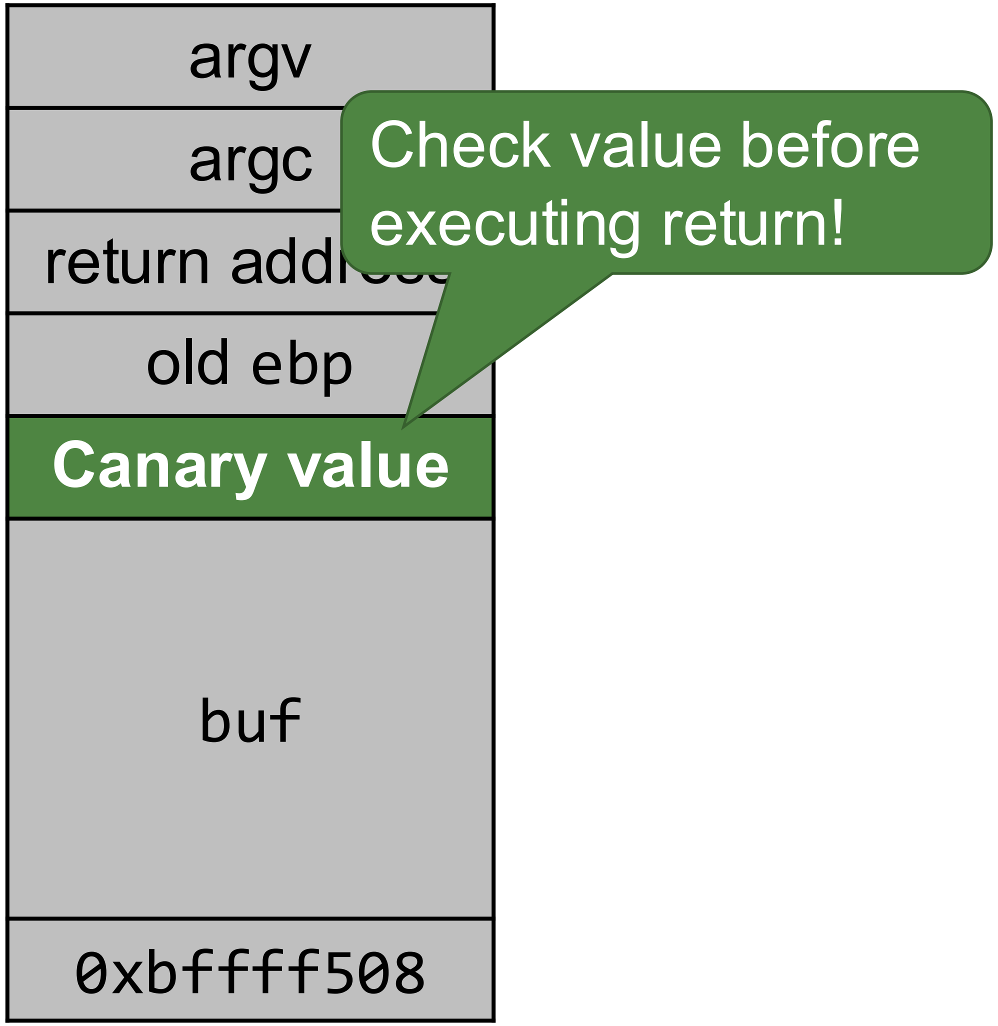
- Submit a single PDF file consisting of multiple presentation slides!
- For this checkpoint, there will be no presentation session; only the checkpoint submission is required
- Due: 10/23, 11:59PM

Project Checkpoint

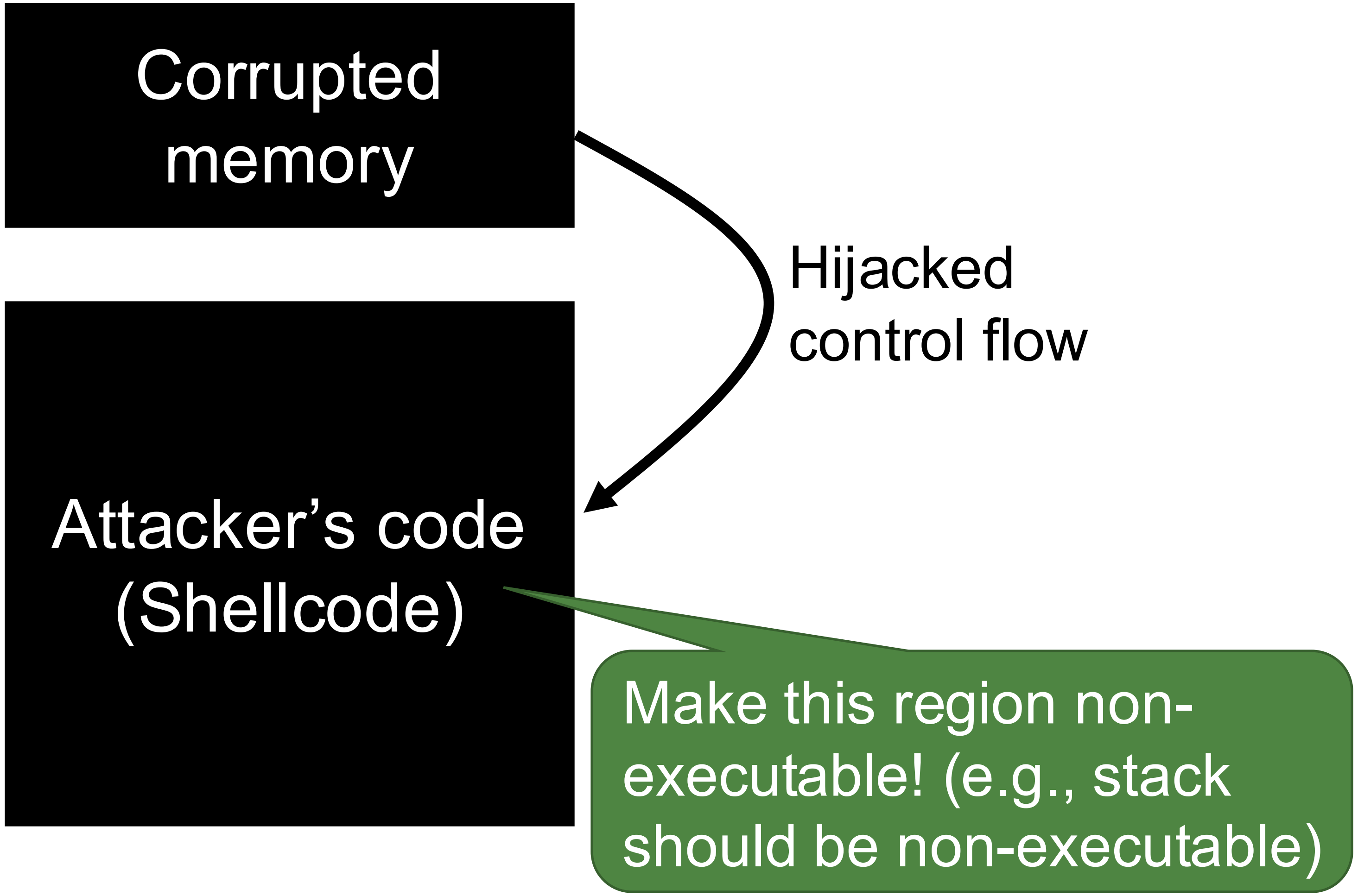
- You should upload a single PDF file on BlackBored.
- The name of the PDF file should have the following format: `[your ID-last name.pdf]`
  - If your name is Gil-dong Hong, and your ID is 20231234, then you should submit a file named “20231234-Hong.pdf”
  - If your team consists of two people, each member must submit a PDF file
- Submit a single PDF file consisting of multiple presentation slides. The PDF should include the following topics and contents (must be written in English!):
  - Introduction
  - Background
  - Motivation
  - Approach
  - Your Progress

Recap: Mitigating Memory Corruption Bugs

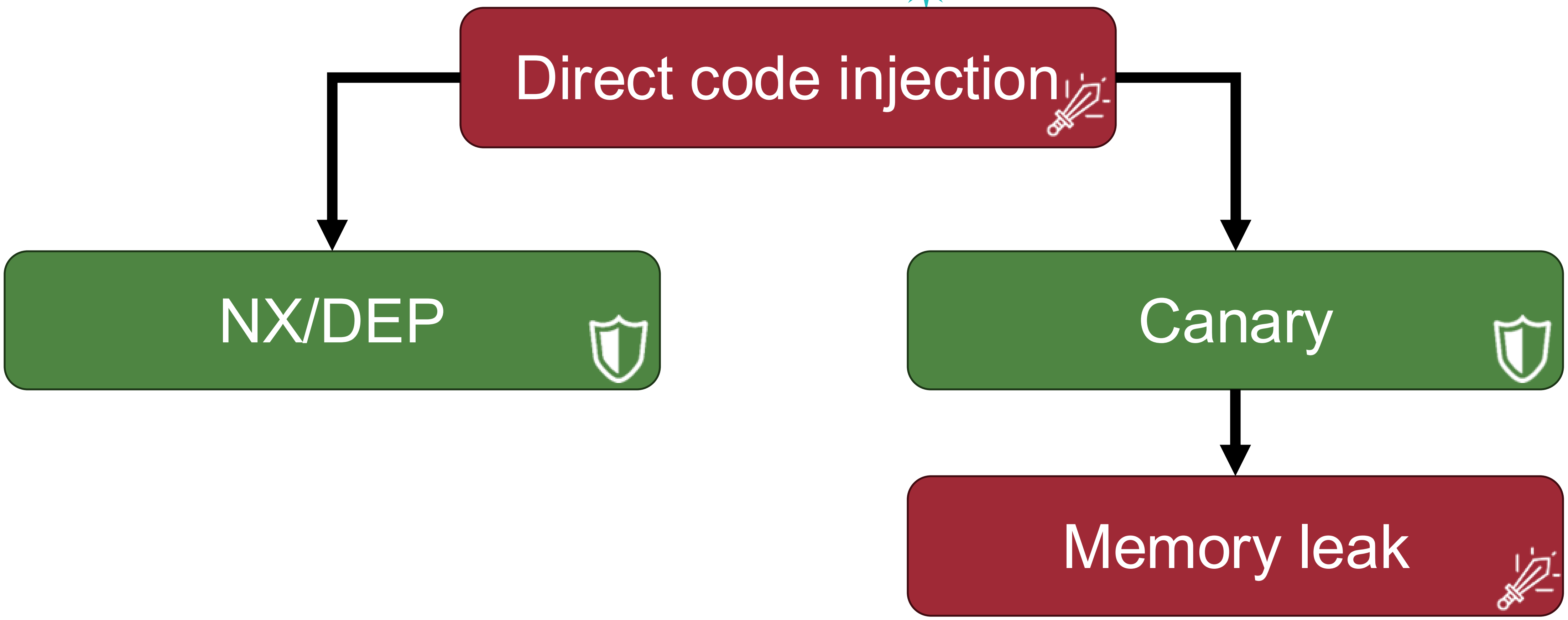
Mitigation #1: Canary



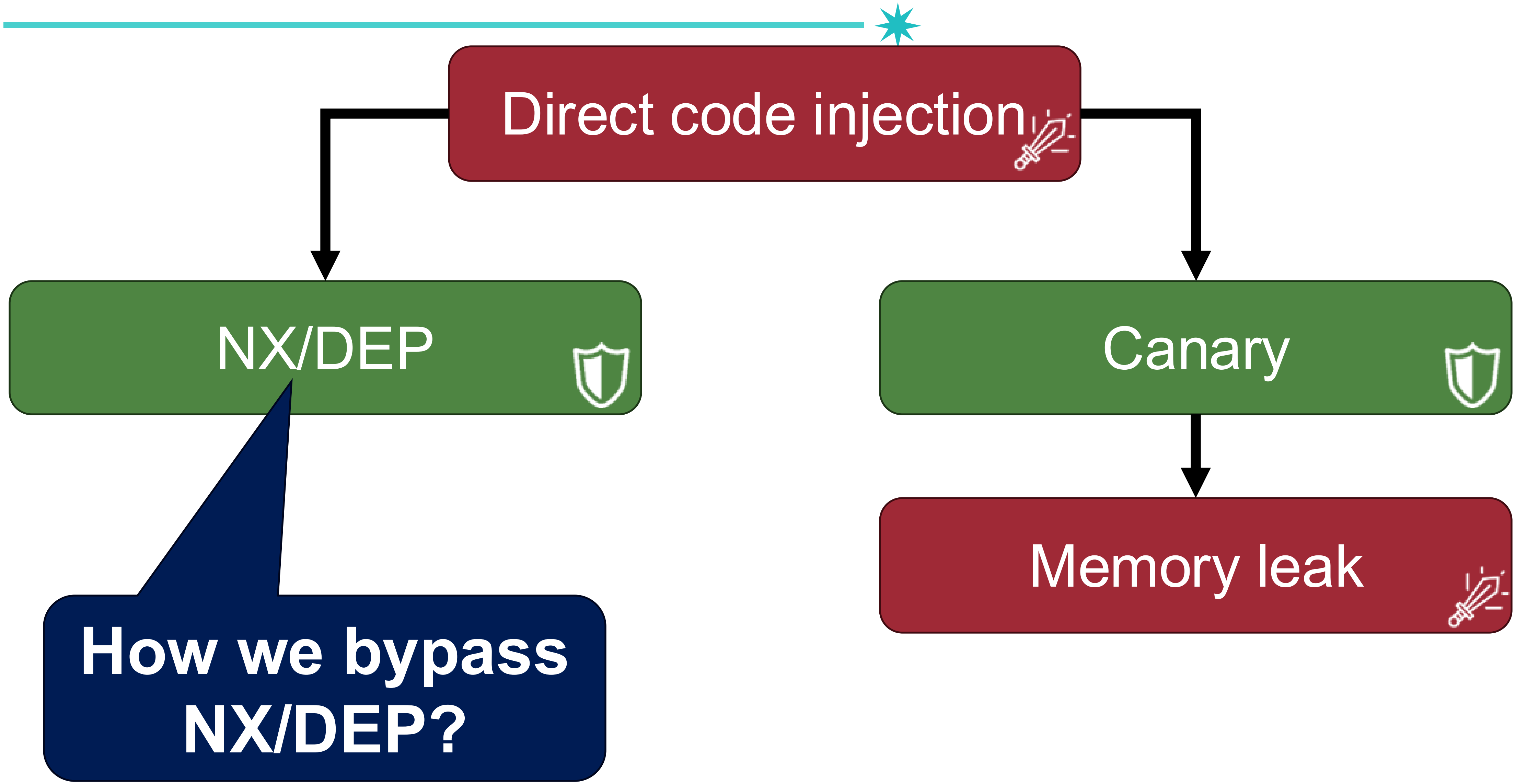
Mitigation #2: NX (No eXcute)



Control Hijack Attack / Defense So Far



Motivation

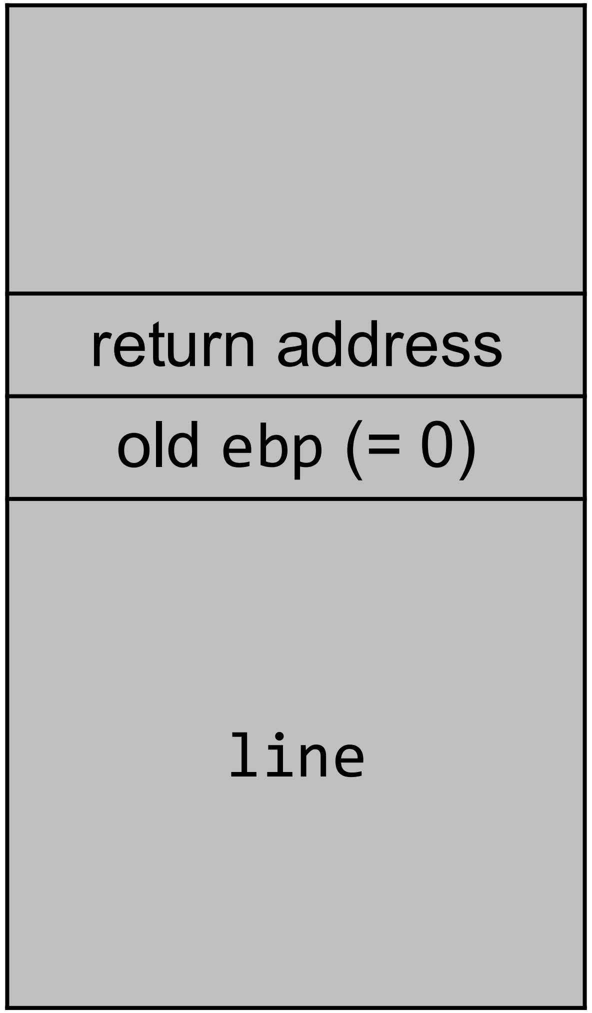

Direct code injection
NX/DEP
Canary
Memory leak
How we bypass NX/DEP?

Code-Reuse Attacks

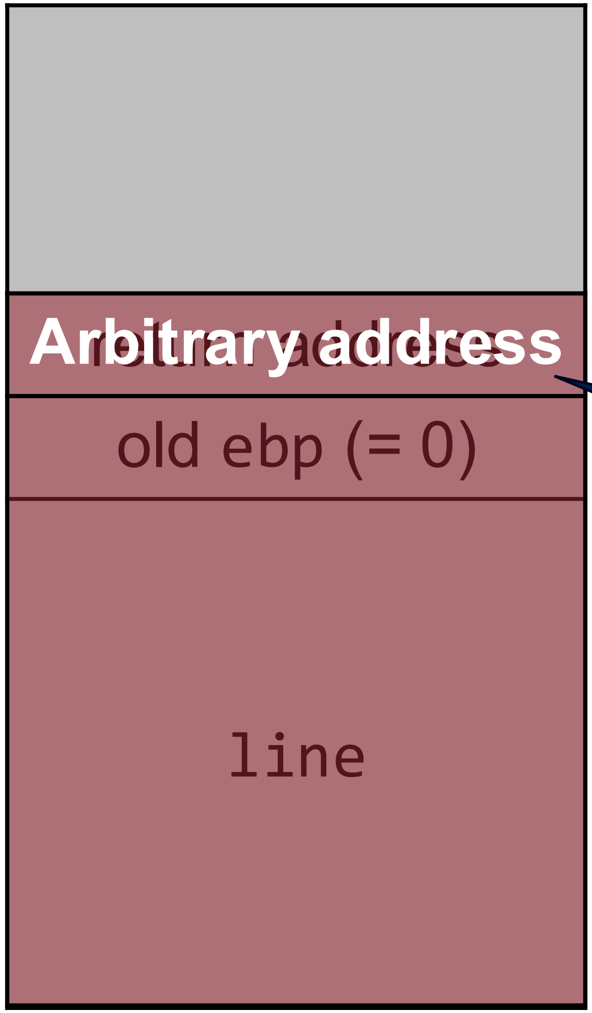
Bypassing DEP

- Return-to-stack exploit is disabled
- But, we can still jump to an arbitrary address of ***existing code*** **(= *Code Reuse Attack*)**

Main Idea: Jump to Existing Code



Main Idea: Jump to Existing Code



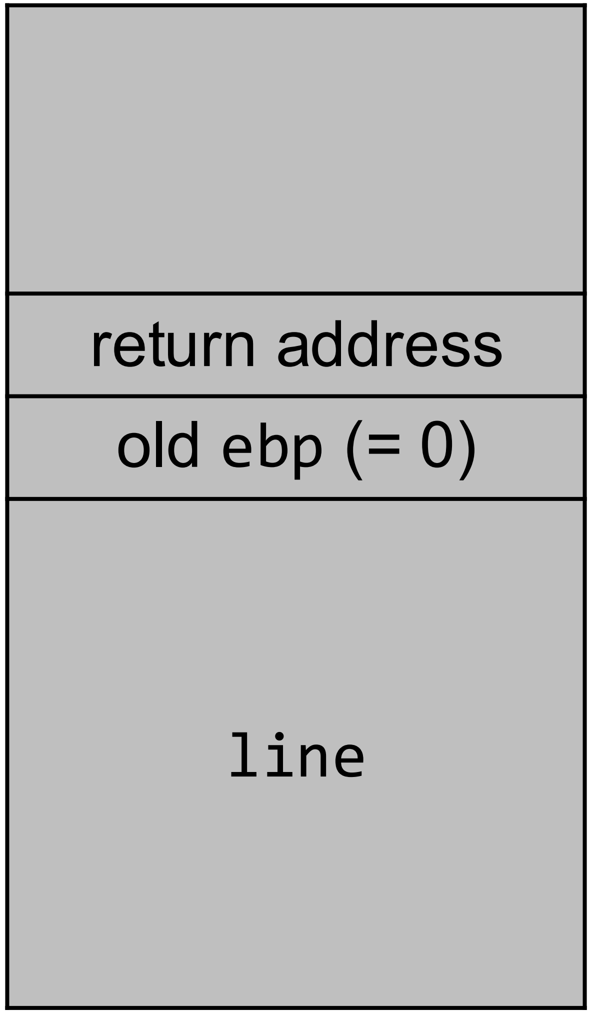
Jump to the ***existing code space***, not to the stack

Code Reuse Attack #1: Return-to-Libc

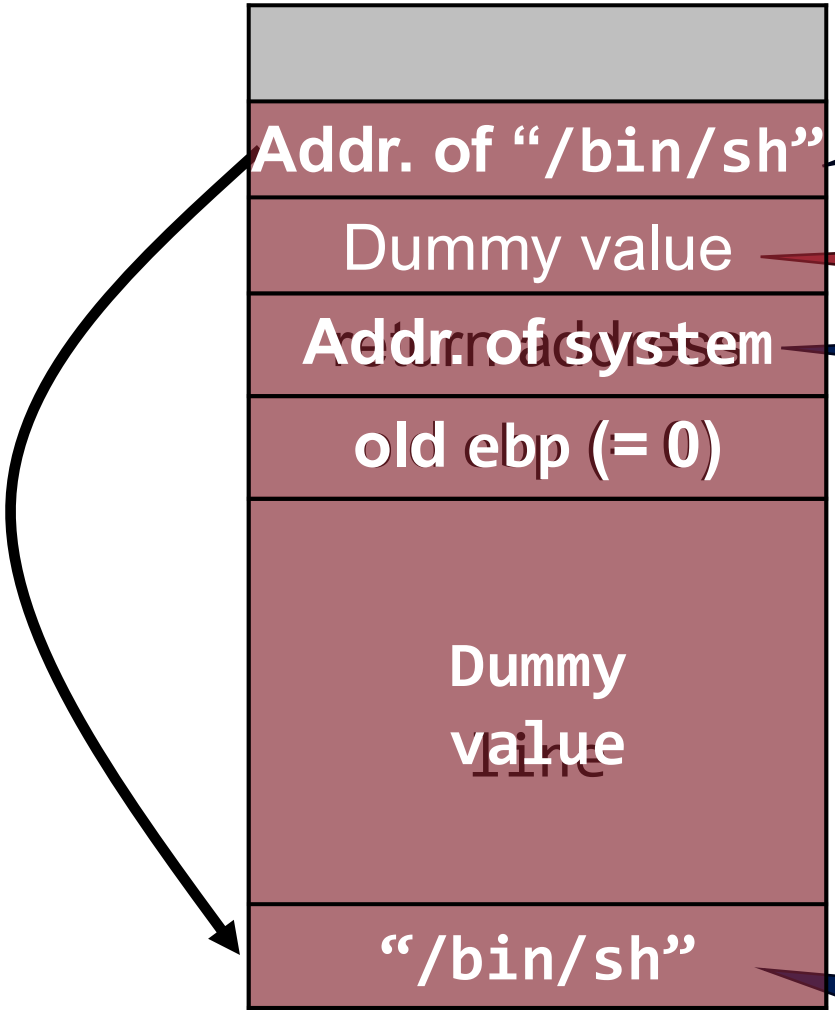
- LIBC (LIBrary C) is a standard library that most programs commonly use
  - For example, `printf` is in LIBC

- Many useful functions in LIBC to execute
  - `exec` family: `execl`, `execlp`, `execle`, …
  - `system`
  - `mprotect`
  - `mmap`

Code Reuse Attack #1: Return-to-Libc



Code Reuse Attack #1: Return-to-Libc



Argument to system

Why we insert dummy value?

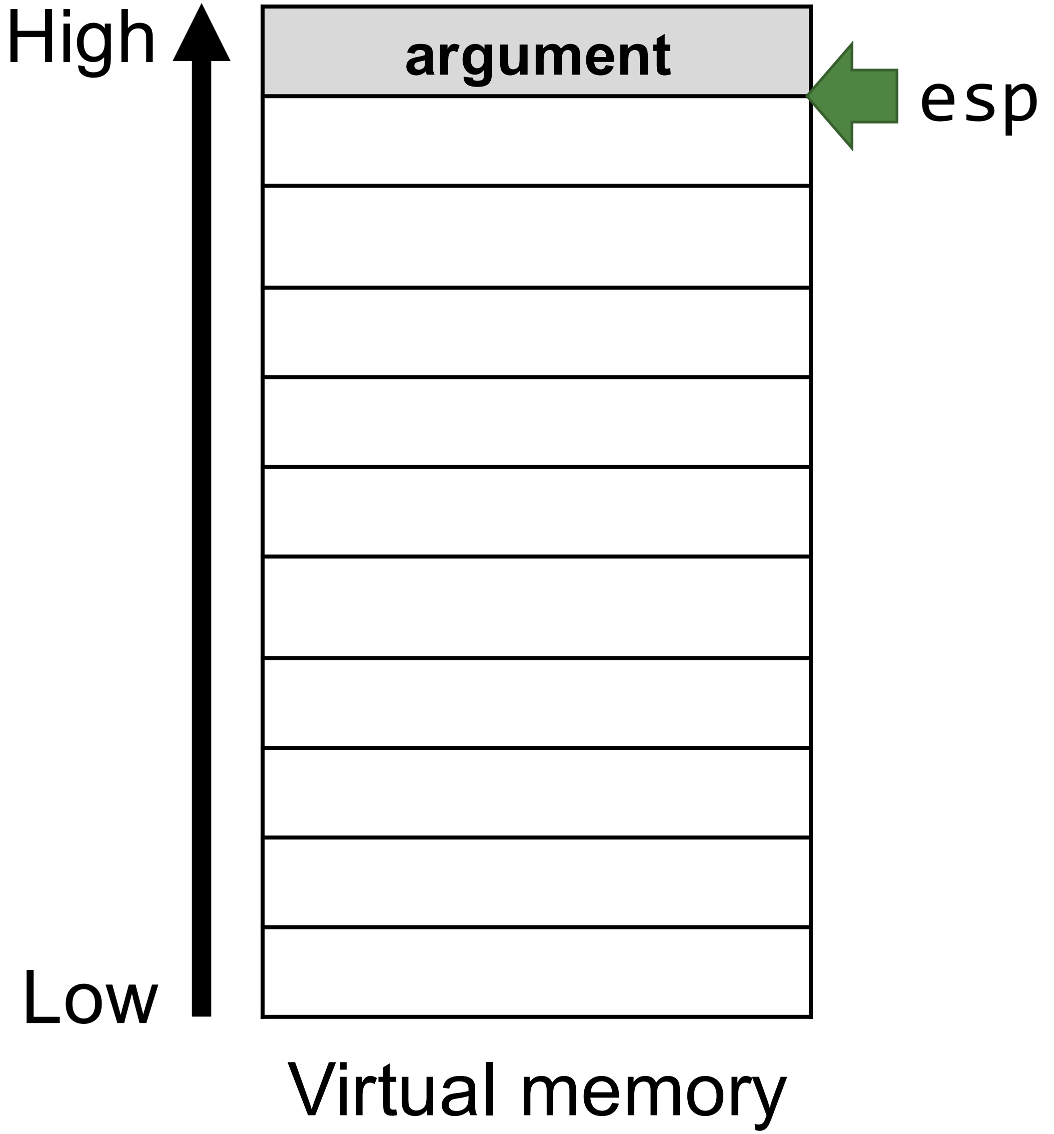
Return to system in Libc

No injected shellcode!
Just inject a string value

Recap: Function Call (call)

```
    …
    call system
nextret:
    nop
    nop
    ...
system (in libc):
    nop
    nop
```

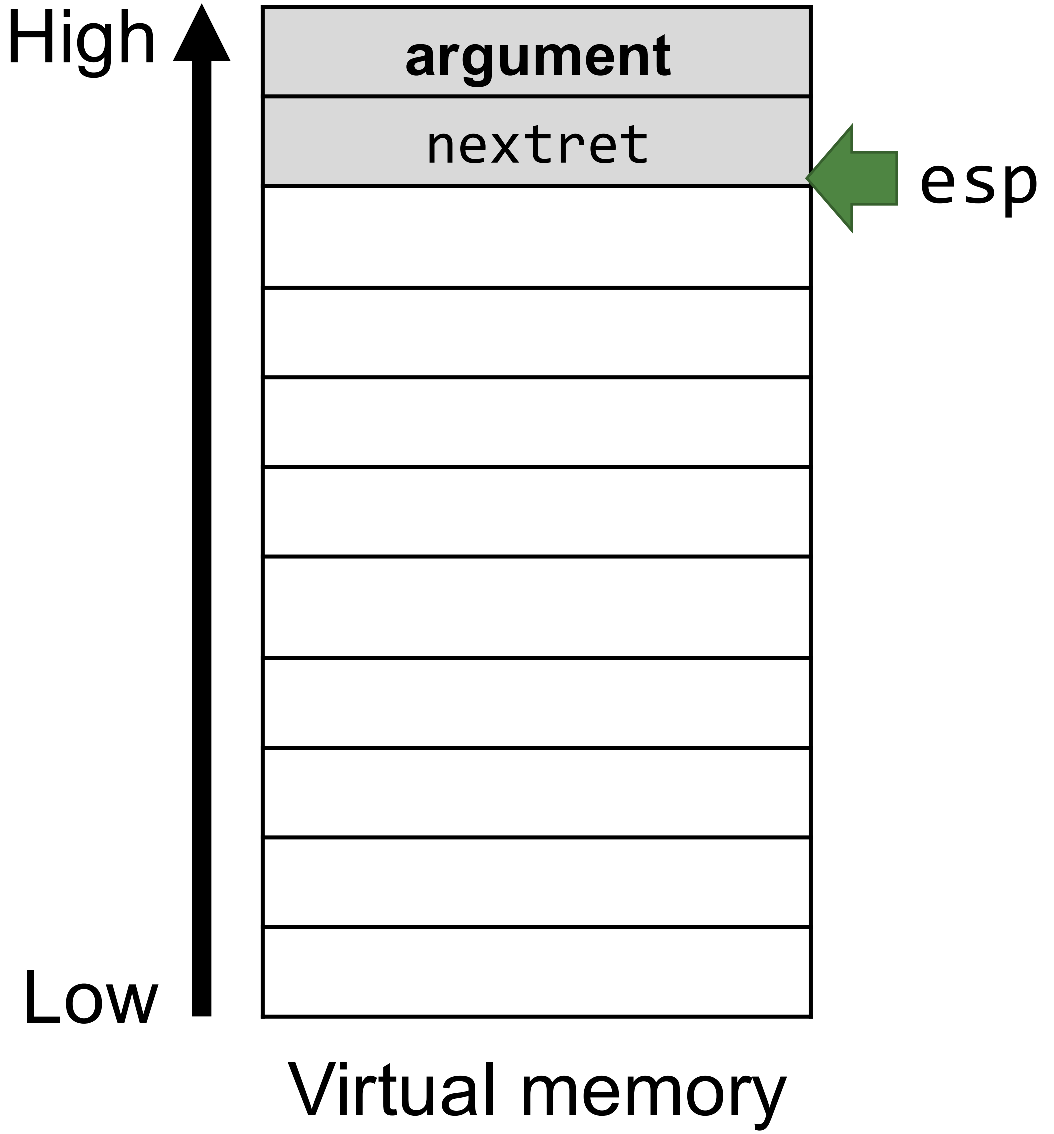
```
push nextret
jmp system
```



Recap: Function Call (call)

```
    …
    call system
nextret:
    nop
    nop
    ...
system (in libc):
    nop
    nop
```

```
push nextret
jmp system
```

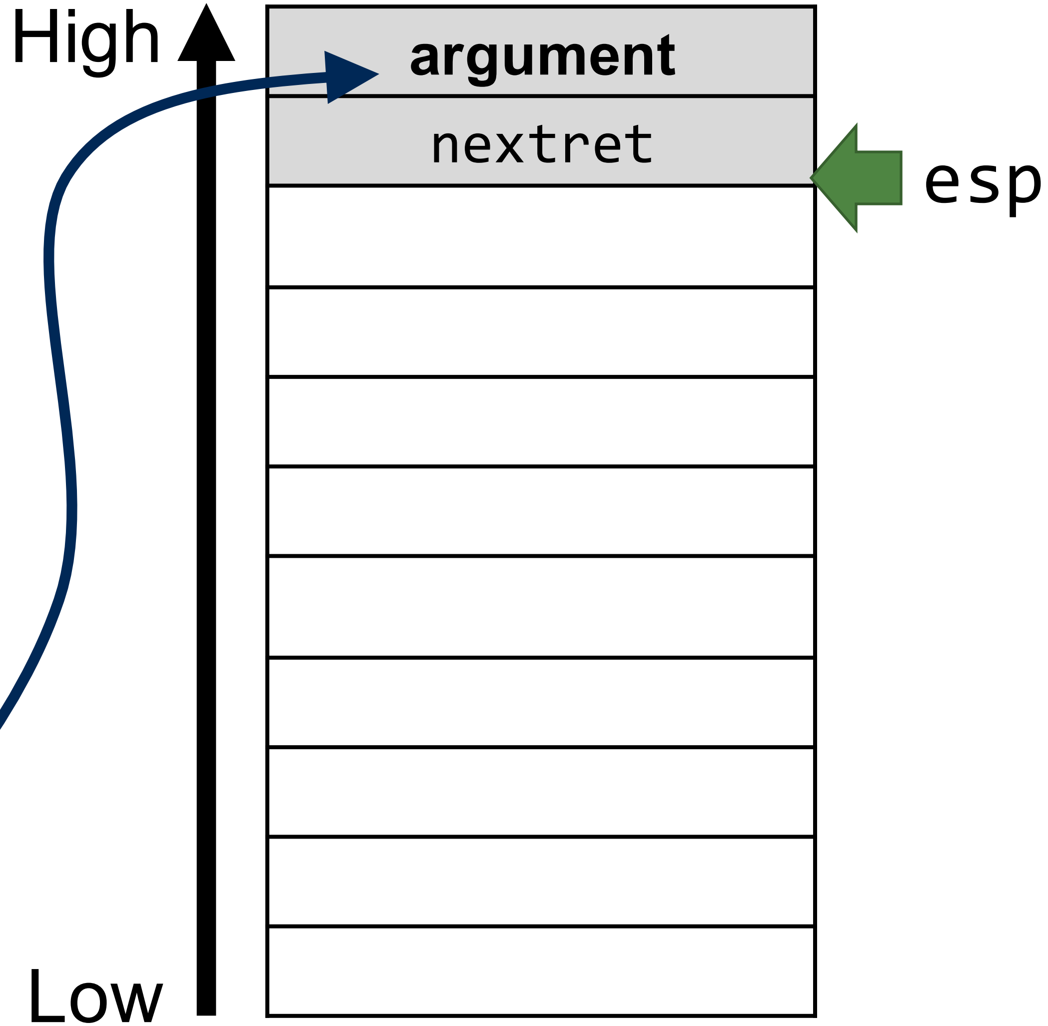


Recap: Function Call (`call`)

```
    …
    call system
nextret:
    nop
    nop
    ...
system (in libc):
    nop
    nop
```

```
push nextret
jmp system
```

Recognize `esp+0x4` as first parameter



Virtual memory

LIBC provides System Call Wrapper


```
08049162 <main>:
 8049162: 55                      push   ebp
 8049163: 89 e5                   mov    ebp,esp
 8049165: 83 ec 08                sub    esp,0x8
 8049168: c7 45 f8 08 a0 04 08    mov    DWORD PTR [ebp-0x8],0x804a008
 804916f: c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-0x4],0x0
 8049176: 6a 00                   push   0x0
 8049178: 8d 45 f8                lea    eax,[e
 804917b: 50                      push   eax
 804917c: 68 08 a0 04 08          push   0x804a008
 8049181: e8 c7 29 02 00          call   806c4b0 <__execve>
```

First argument of execve

You are actually calling a wrapper function around the syscall

LIBC provides System Call Wrapper

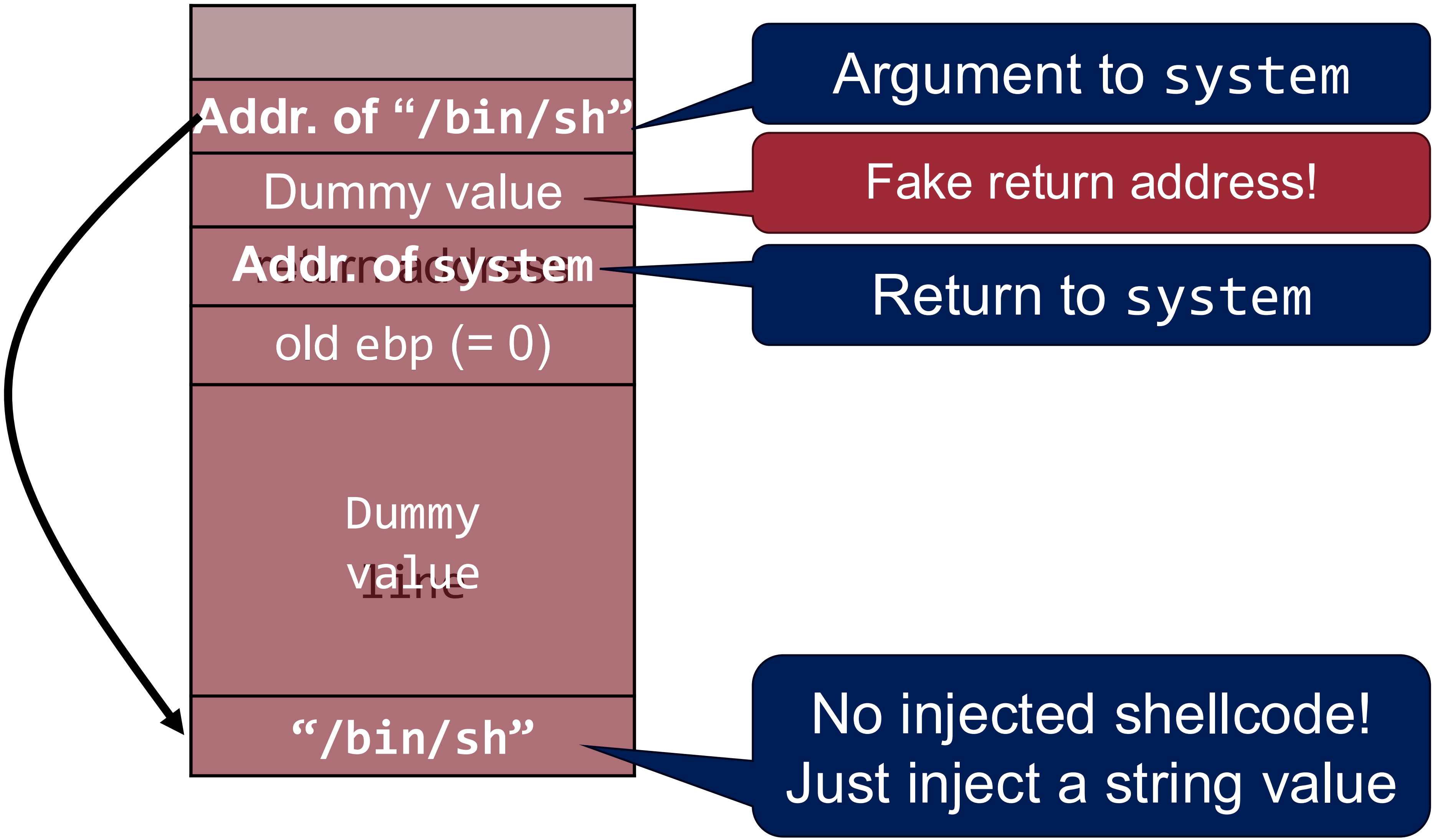
LIBC code

```
0806c4b0 <__execve>:
 806c4b0:    53                      push    ebx
 806c4b1:    8b 54 24 10             mov     edx,DWORD PTR [esp+0x10]
 806c4b5:    8b 4c 24 0c             mov     ecx,DWORD PTR [esp+0xc]
 806c4b9:    8b 5c 24 08             mov     ebx,DWORD PTR [esp+0x8]
 806c4bd:    b8 0b 00 00 00          mov     eax,0xb
 806c4c2:    cd 80                   int     0x80
```

Get first argument

System Call!

Code Reuse Attack #1: Return-to-Libc



Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

≠

```
victim@environment:/# ldd --version
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
```

Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC
- LIBC may not be used at all
- Some functions in LIBC can be excluded

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

≠

```
victim@environment:/# ldd --version
ldd (Ubuntu GLIBC 2.27-3ubuntu1) 2.27
```

Motivation of Return-oriented Programming

Return-to-LIBC requires LIBC function calls, but …☹

- Different versions of LIBC
- LIBC may not be used at all
- Some functions in LIBC can be excluded

```
attacker_local@environment:~$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.17) 2.31
```

Can we spawn a shell without the use of LIBC functions?

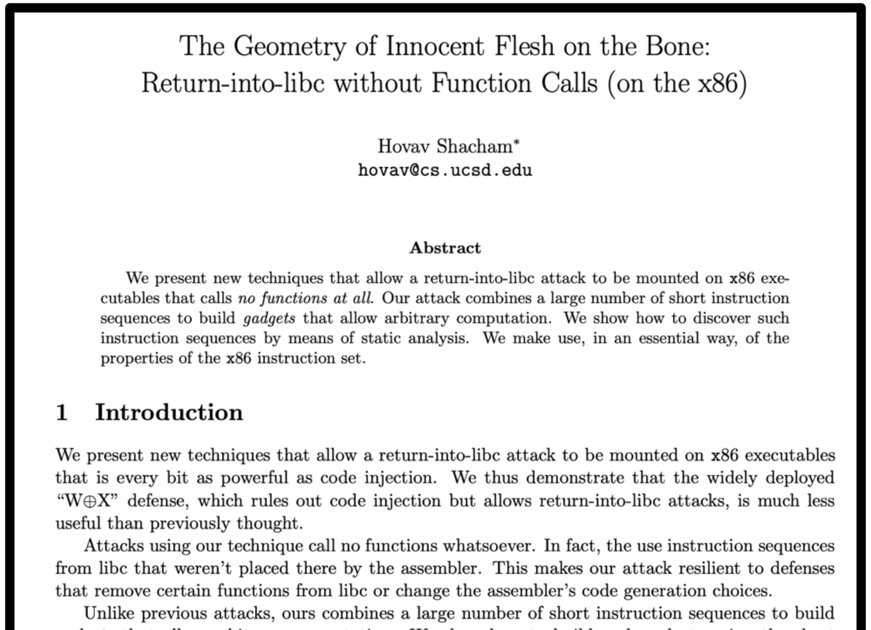
Return-oriented Programming (ROP)

Code Reuse Attack #2: ROP

Generalized Code Reuse Attack

Formally introduced by Hovav in CCS 2007

“The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)”



The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed “W⊕X” defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren’t placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler’s code generation choices.

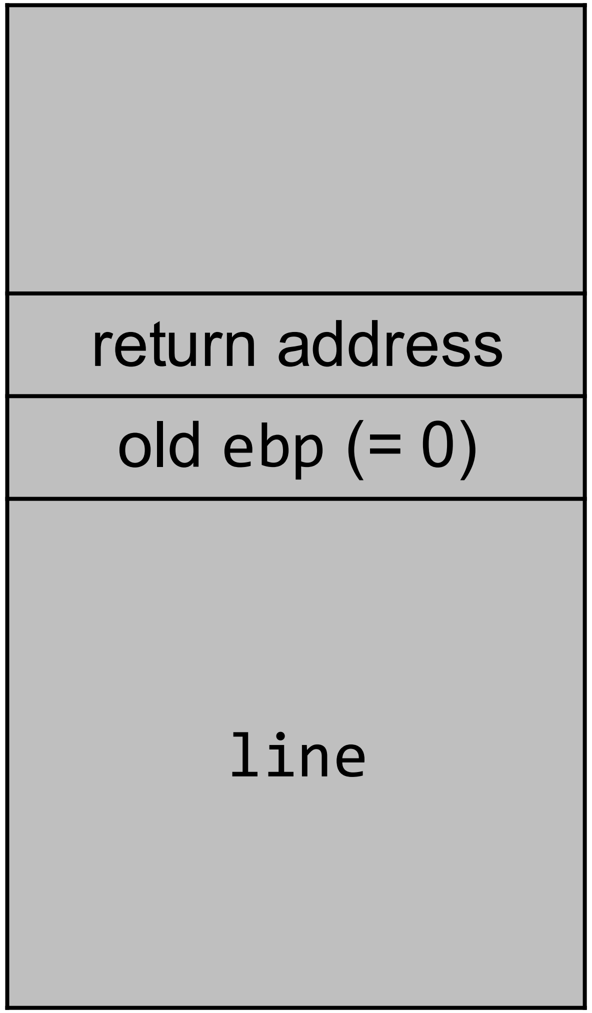
Unlike previous attacks, ours combines a large number of short instruction sequences to build

Main Idea: Return (`ret`) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

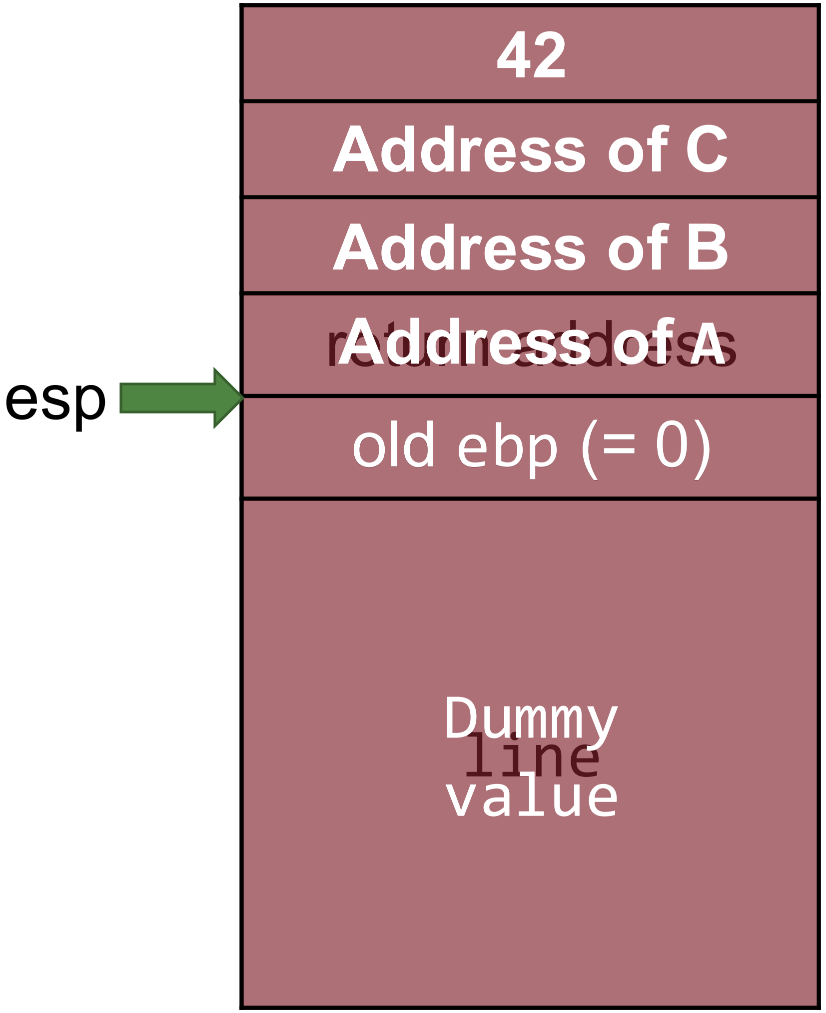


Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

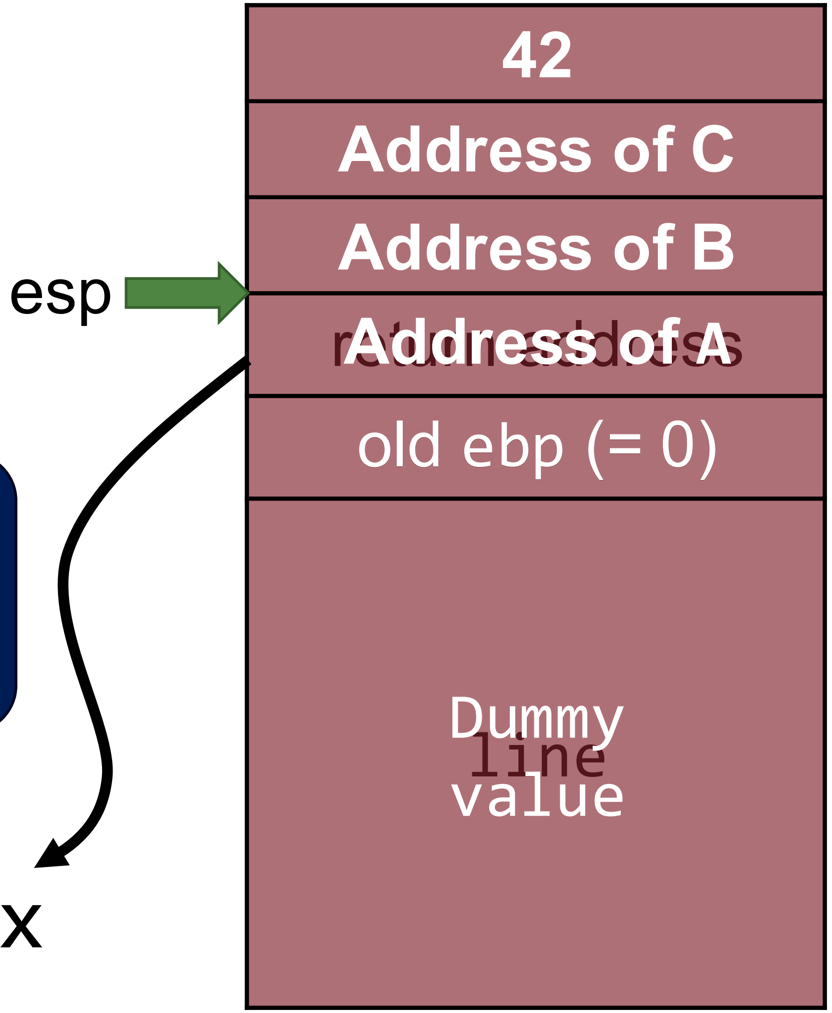
Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

Somewhere in the binary code

A
```
add eax, ebx
ret
```



Main Idea: Return (ret) Chaining

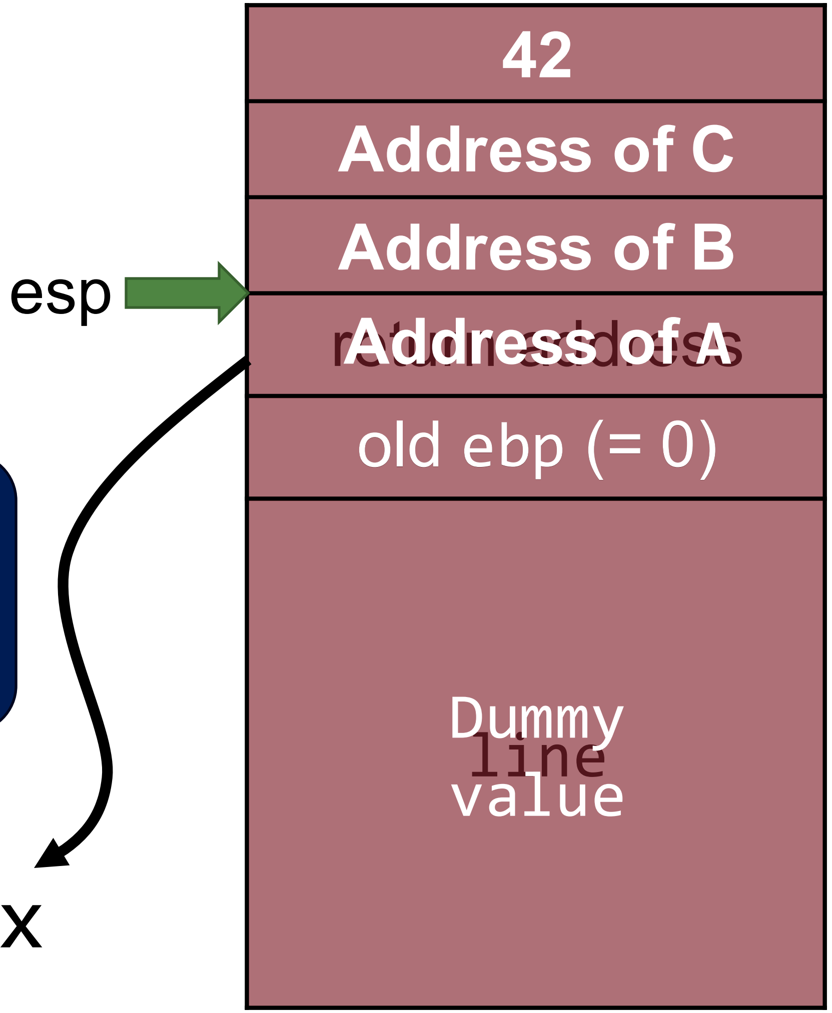
Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

ROP Gadget:
Instruction sequence that ends with ret

A

```
add eax, ebx
ret
```

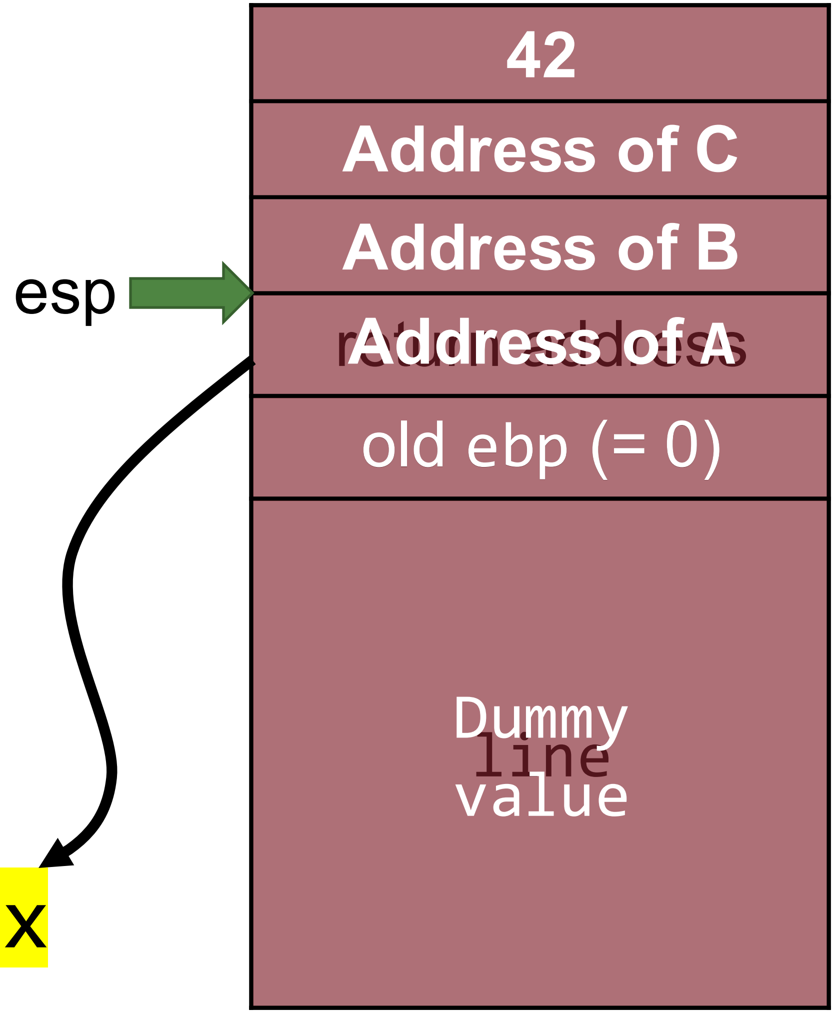


Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



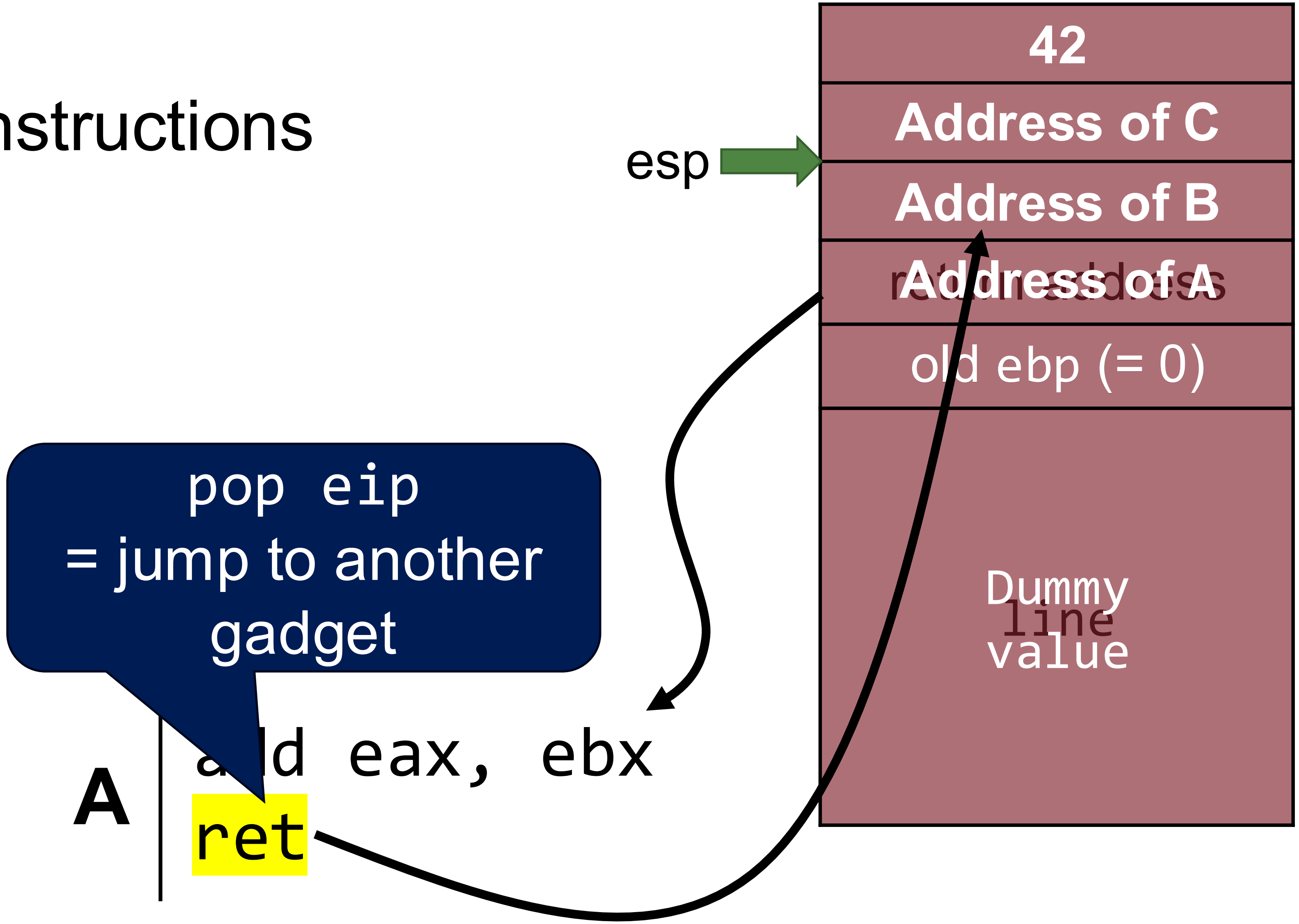
A

```
add eax, ebx
ret
```

Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

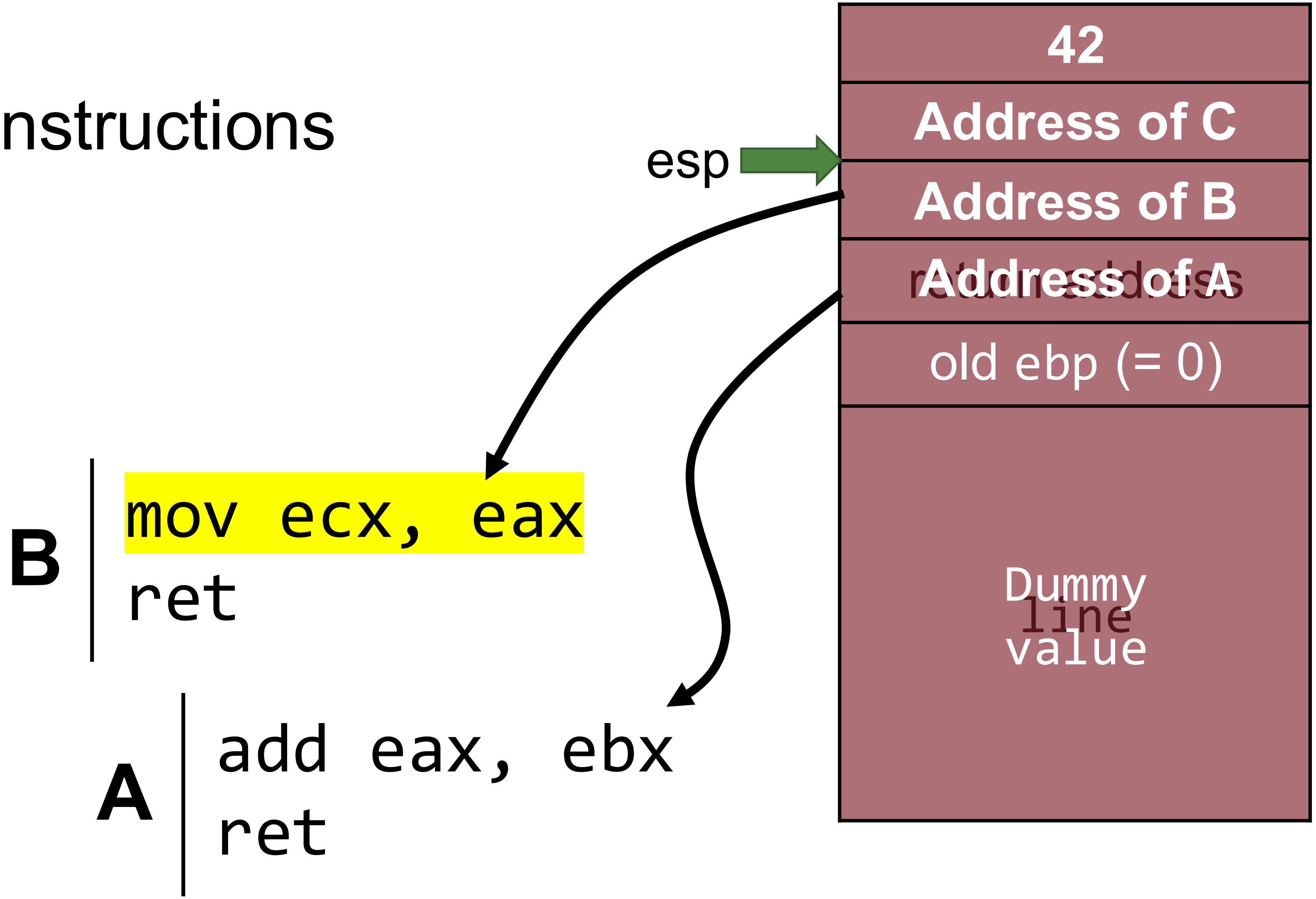


Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

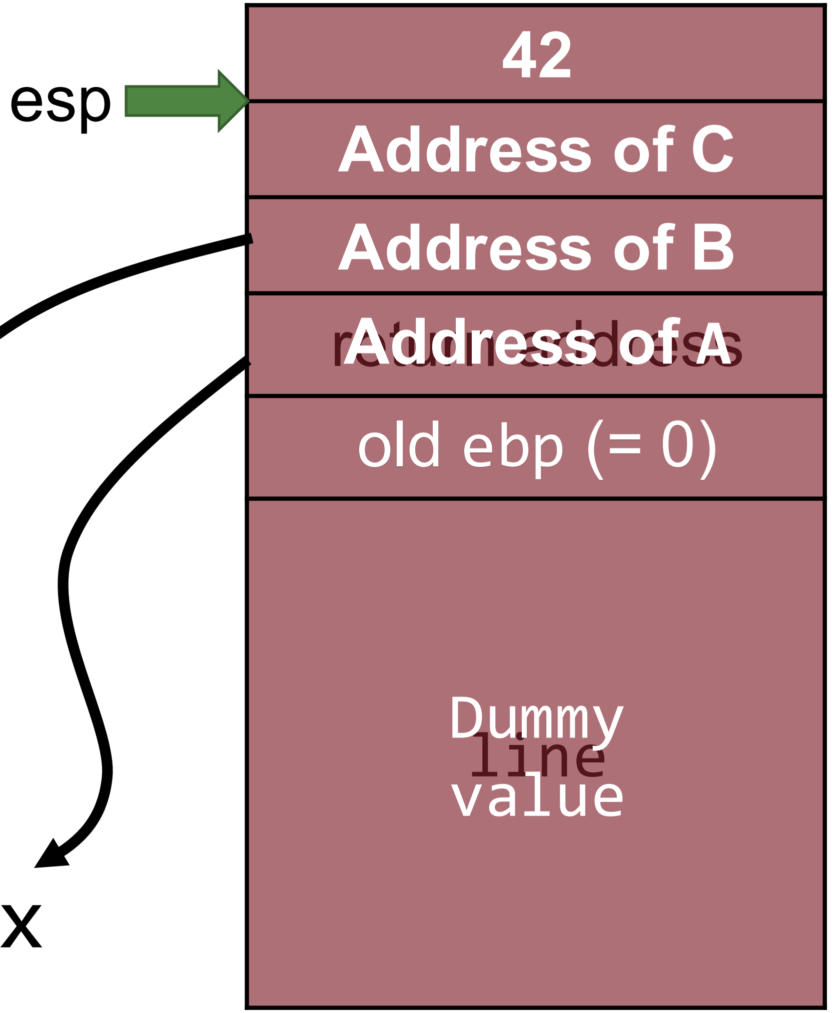
Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

B
```
mov ecx, eax
ret
```

A
```
add eax, ebx
ret
```

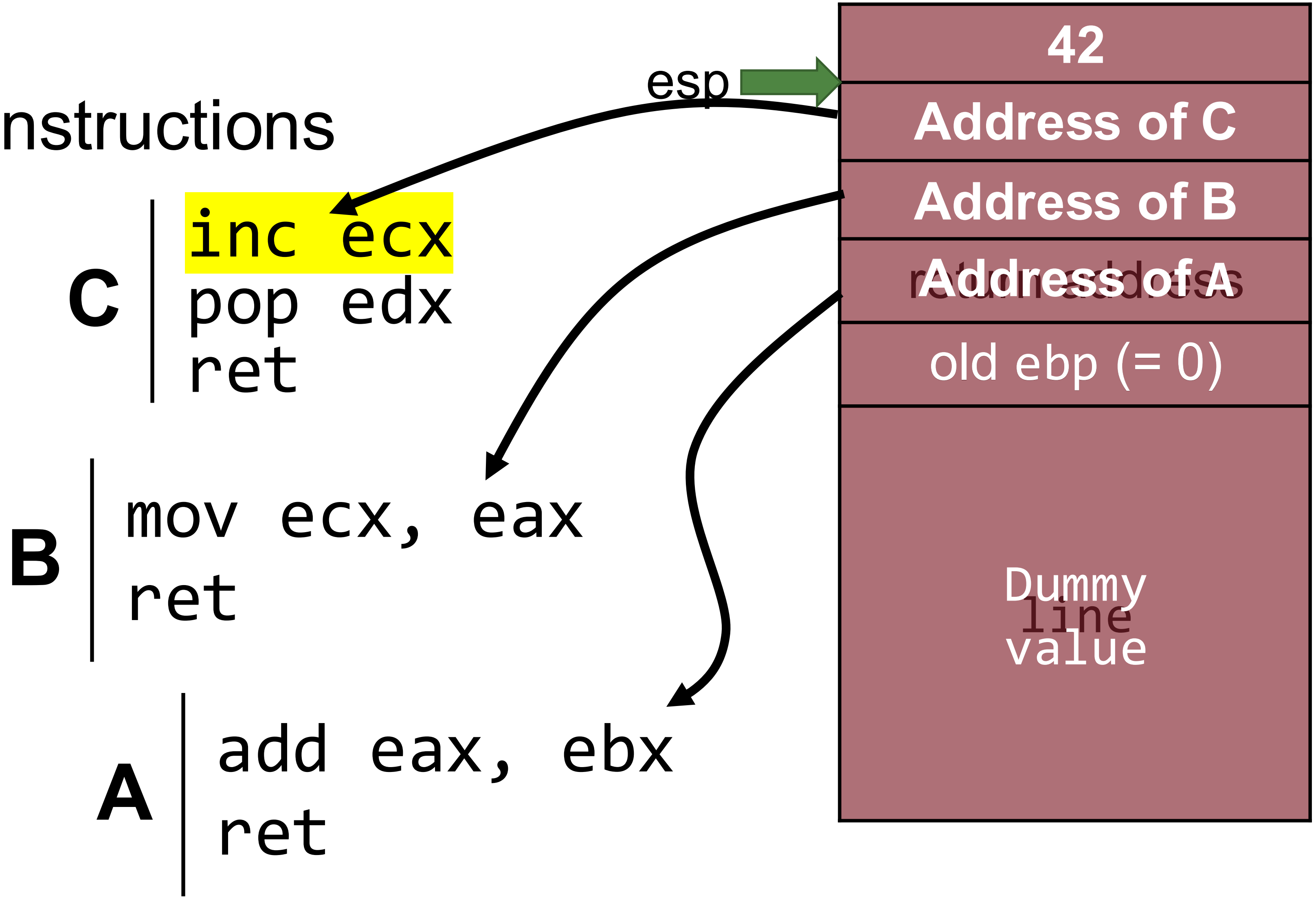


Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```



Main Idea: Return (ret) Chaining

Attacker's goal:

execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C

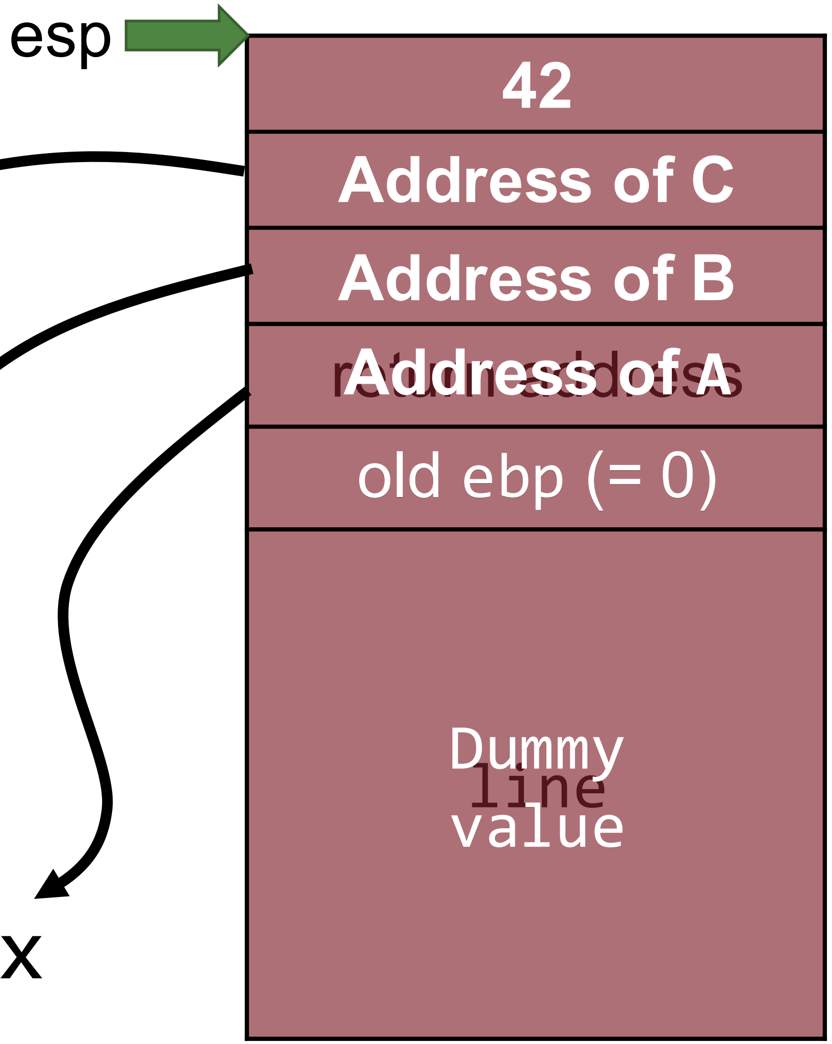
```
inc ecx
pop edx
ret
```

B

```
mov ecx, eax
ret
```

A

```
add eax, ebx
ret
```



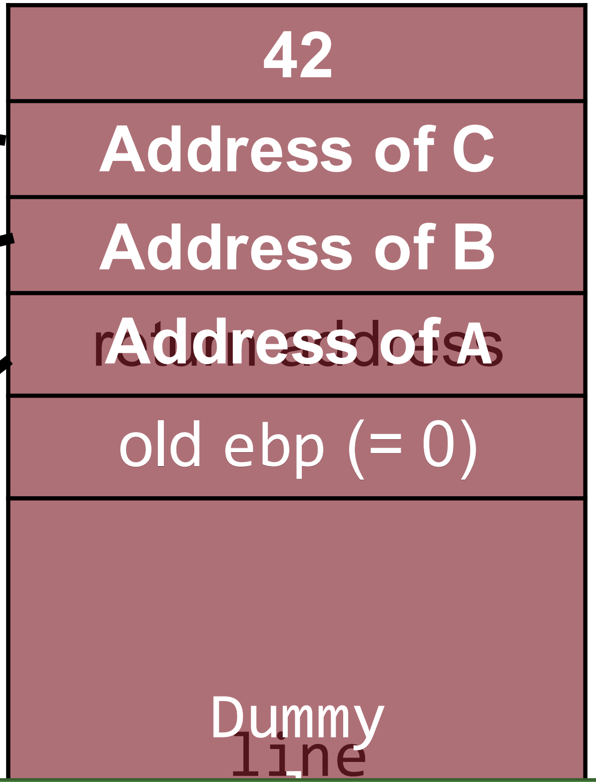
Main Idea: Return (ret) Chaining

Attacker's goal:
execute following instructions

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C
```
inc ecx
pop edx
ret
```

B
```
mov ecx, eax
ret
```



Return chaining with ROP gadgets
allows **arbitrary computation!**

ROP Practice

Goal: Modify `ptr` to be `0x42424242` with ROP

```
mov [ptr], 0x42424242
```

Gadget **A**

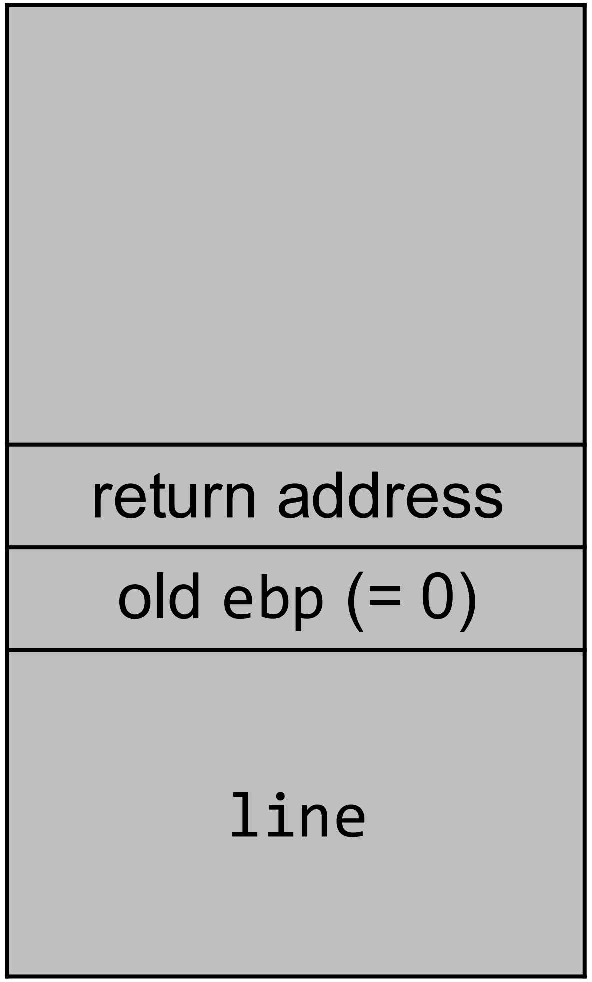
```
pop eax
ret
```

Gadget **B**

```
pop ebx
ret
```

Gadget **C**

```
mov [eax], ebx
ret
```



ROP Workflow

1. Disassemble binary
2. Identify useful instruction sequences (i.e., gadgets)
   - E.g., an instruction sequence that ends with `ret` is useful
   - E.g., an instruction sequence that ends with `jmp reg` can be useful (`pop eax; jmp eax`)
3. Assemble gadgets to perform some computation
   - E.g., spawning a shell

Challenge: Gathering as **many gadgets** as possible

Many Gadgets in Regular Binaries?

x86 instructions have their lengths ranging from 1 byte to 18 bytes, i.e., it uses ***variable-length encoding***

x86 instructions have variable lengths

```
08048aac <main>:
 8048aac:       8d 4c 24 04             lea    ecx,[esp+0x4]
 8048ab0:       83 e4 f0                and    esp,0xfffffff0
 8048ab3:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048ab6:       55                      push   ebp
 8048ab7:       89 e5                   mov    ebp,esp
 8048ab9:       51                      push   ecx
 8048aba:       83 ec 14                sub    esp,0x14
 8048abd:       c7 45 f0 88 ad 0a 08    mov    DWORD PTR [ebp-0x10],0x80aad88
 8048ac4:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 8048acb:       83 ec 04                sub    esp,0x4
 8048ace:       6a 00                   push   0x0
 8048ad0:       8d 45 f0                lea    eax,[ebp-0x10]
 8048ad3:       50                      push   eax
 8048ad4:       68 88 ad 0a 08          push   0x80aad88
 8048ad9:       e8 02 39 01 00          call   805c3e0 <__execve>
 ...
```

Many Gadgets in Regular Binaries?

x86 instructions have their lengths ranging from 1 byte to 18 bytes, i.e., it uses ***variable-length encoding***

Therefore, there can be both **intended** and **unintended gadgets** in x86 binaries

Disassembling x86


```
eip
↓
e8 05 ff ff ff          call 8048330
81 c3 59 12 00 00       add ebx,0x1259
```

What if we disassemble the code from the second byte (05)?

Unintended ret Insturction

```
   eip
    ↓
e8 05 ff ff ff           add eax, 0x81ffffff
81 c3 59 12 00 00        ret
```

Totally different, but still valid instructions!

Unintended ret Insturction


```
   eip
    ↓
e8 05 ff ff ff               add eax, 0x81ffffff
81 c3 59 12 00 00            ret
```

Unintended ret Insturction

```
   eip
    ↓
e8 05 ff ff ff          add eax, 0x81ffffff
81 c3 59 12 00 00       ret
```

Disassemble from Any Addresses in Memory Pages

- This is perfectly legal
- We can find lots of ***unintended*** `ret` instructions

Finding Unintended Gadgets

Algorithm GALILEO:

create a node, *root*, representing the `ret` instruction;
place *root* in the trie;
for *pos* **from** 1 **to** *textseg_len* **do**:
  if the byte at *pos* is `c3`, i.e., a `ret` instruction, **then**:
    call BUILDFROM(*pos*, *root*).

Find c3 (ret) instruction

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):

for *step* **from** 1 **to** *max_insn_len* **do**:
  if bytes $\left[(pos - step) \ldots (pos - 1)\right]$ decode as a valid instruction *insn* **then**:
    ensure *insn* is in the trie as a child of *parent_insn*;
    if *insn* isn't boring **then**:
      call BUILDFROM(*pos* − *step*, *insn*).

Excerpt from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"

Finding Unintended Gadgets

Algorithm GALILEO:
    create a node, *root*, representing the `ret` instruction;
    place *root* in the trie;
    for *pos* **from** 1 **to** *textseg_len* **do**:
        if the byte at *pos* is `c3`, i.e., a `ret` instruction, **then**:
            call BUILDFROM($pos, root$).

Find c3 (ret) instruction

Procedure BUILDFROM(index *pos*, instruction
    for *step* **from** 1 **to** *max_insn_len* **do**:
        if bytes $\left[(pos - step) \ldots (pos -$
            ensure *insn* is in the tr as a child of *parent_insn*;
            if *insn* isn't boring **then**:
                call BUILDFROM($pos - step, insn$).

"Boring" Instructions

1. The *insn* is a leave instruction
2. The *insn* is pop ebp
3. The *insn* is unconditional jump

Excerpt from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"

Many Gadgets in Regular Binaries?

Also, program size may matter!

Larger code ⇒ More chance to get useful gadgets

Exploit Hardening Made Easy,
USENIX Security 2011

Q: Exploit Hardening Made Easy

Edward J. Schwartz, Thanassis Avgerinos and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{edmcman, thanassis, dbrumley}@cmu.edu

Abstract

Prior work has shown that return oriented programming (ROP) can be used to bypass W⊕X, a software defense that stops shellcode, by reusing instructions from large libraries such as libc. Modern operating systems have since enabled address randomization (ASLR), which randomizes the location of libc, making these techniques unusable in practice. However, modern ASLR implementations leave smaller amounts of executable code unrandomized and it has been unclear whether an attacker can use these small code fragments to construct payloads in the general case.

In this paper, we show defenses as currently deployed can be bypassed with new techniques for automatically

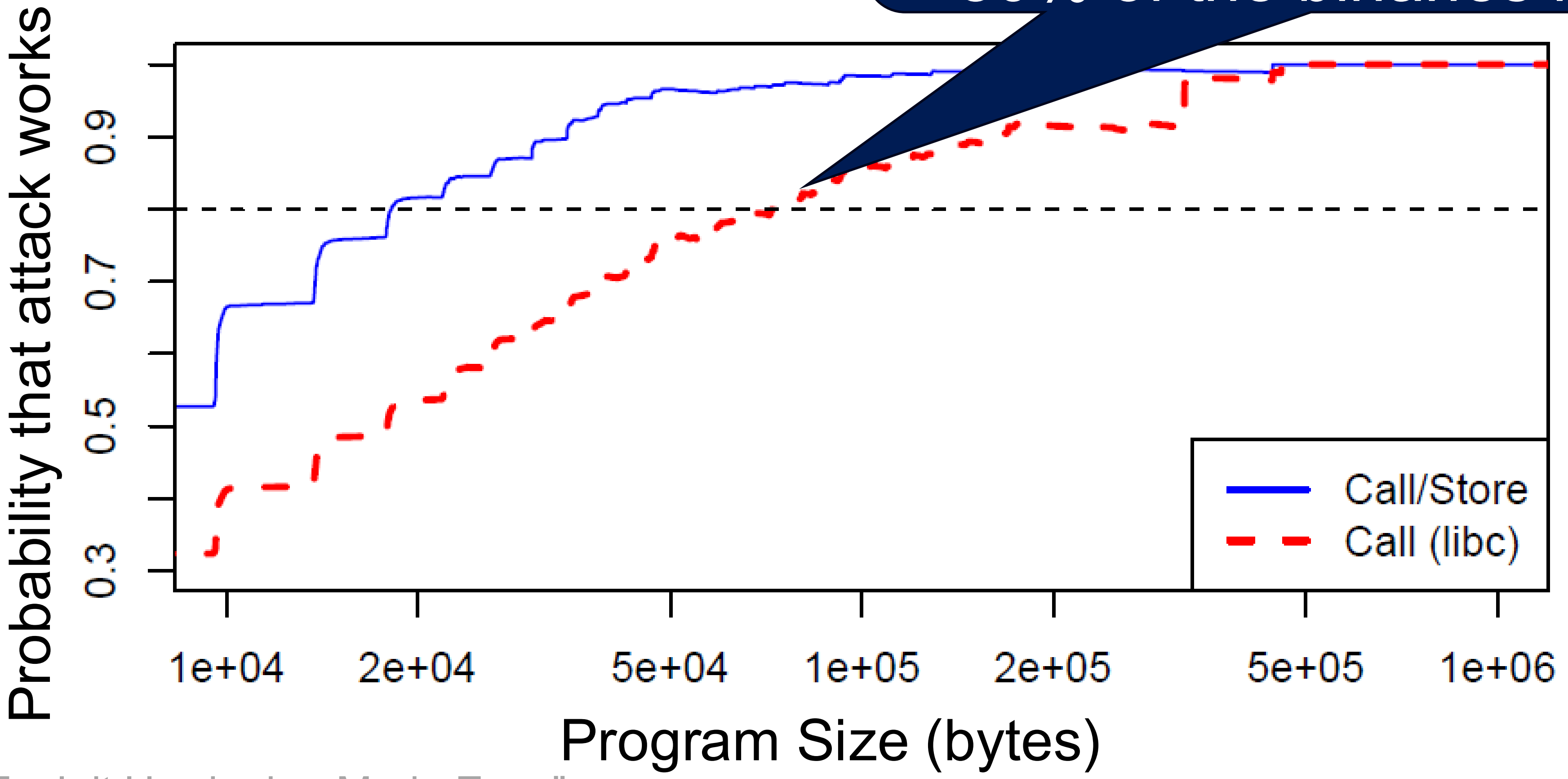
could be to spawn a remote shell to control the program, to install malware, or to exfiltrate sensitive information stored by the program.

Luckily, modern OSes now employ W⊕X and ASLR together — two defenses intended to thwart control flow hijacks. Write xor eXecute (W⊕X, also known as DEP) prevents an attacker's payload itself from being directly executed. Address space layout randomization (ASLR) prevents an attacker from utilizing structures within the application itself as a payload by randomizing the addresses of program segments. These two defenses, when used together, make control flow hijack vulnerabilities difficult to exploit.

However, ASLR and W⊕X are not enforced com-

For All /usr/bin Programs

Show that 100KB was enough to successfully create exploits for 80% of the binaries in /usr/bin



Excerpt from "Exploit Hardening Made Easy"

Question

How can we mitigate code reuse attacks (ROP)?

Defenses against Code Reuse Attacks

- Detection
  - ROP Payload Detection Using Speculative Code Execution, ***MALWARE 2011***
  - Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, ***USENIX Security 2013***
  - ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks, ***NDSS 2014***

New Attack

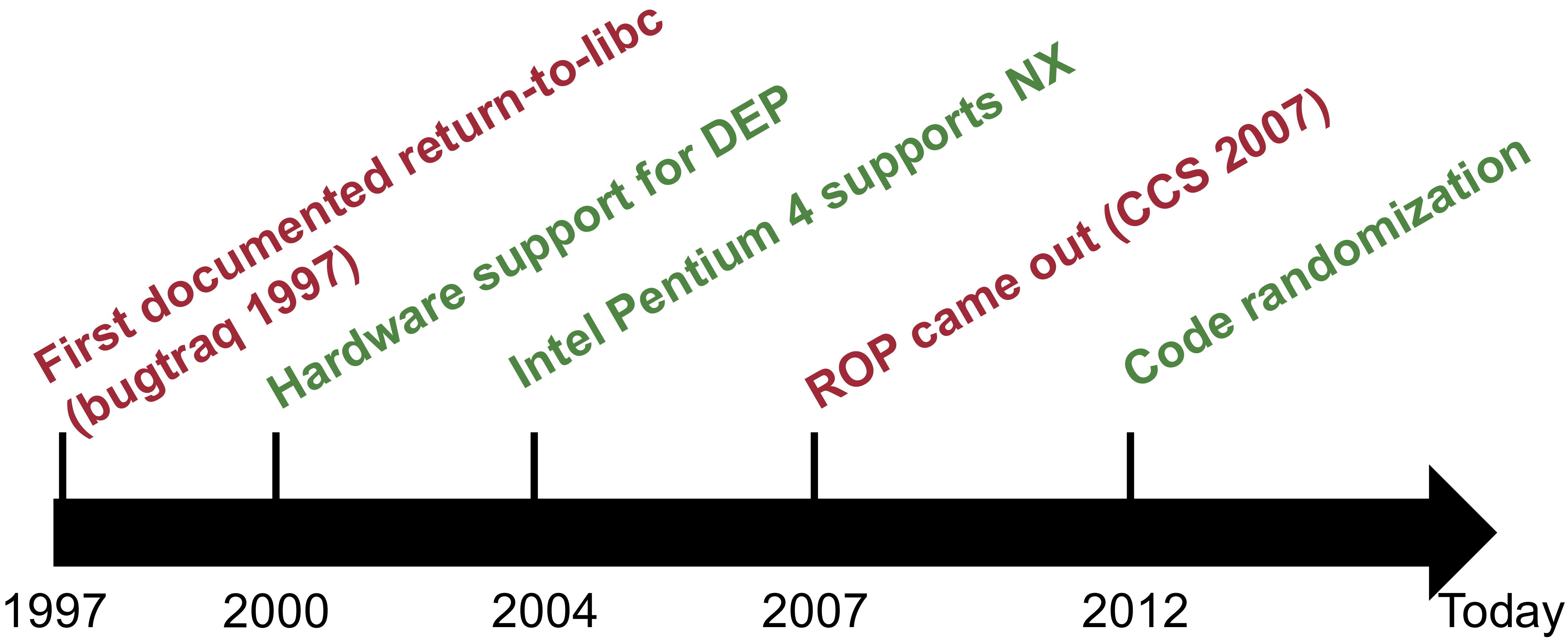
Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard, ***USENIX Security 2014***

Defenses against Code Reuse Attacks

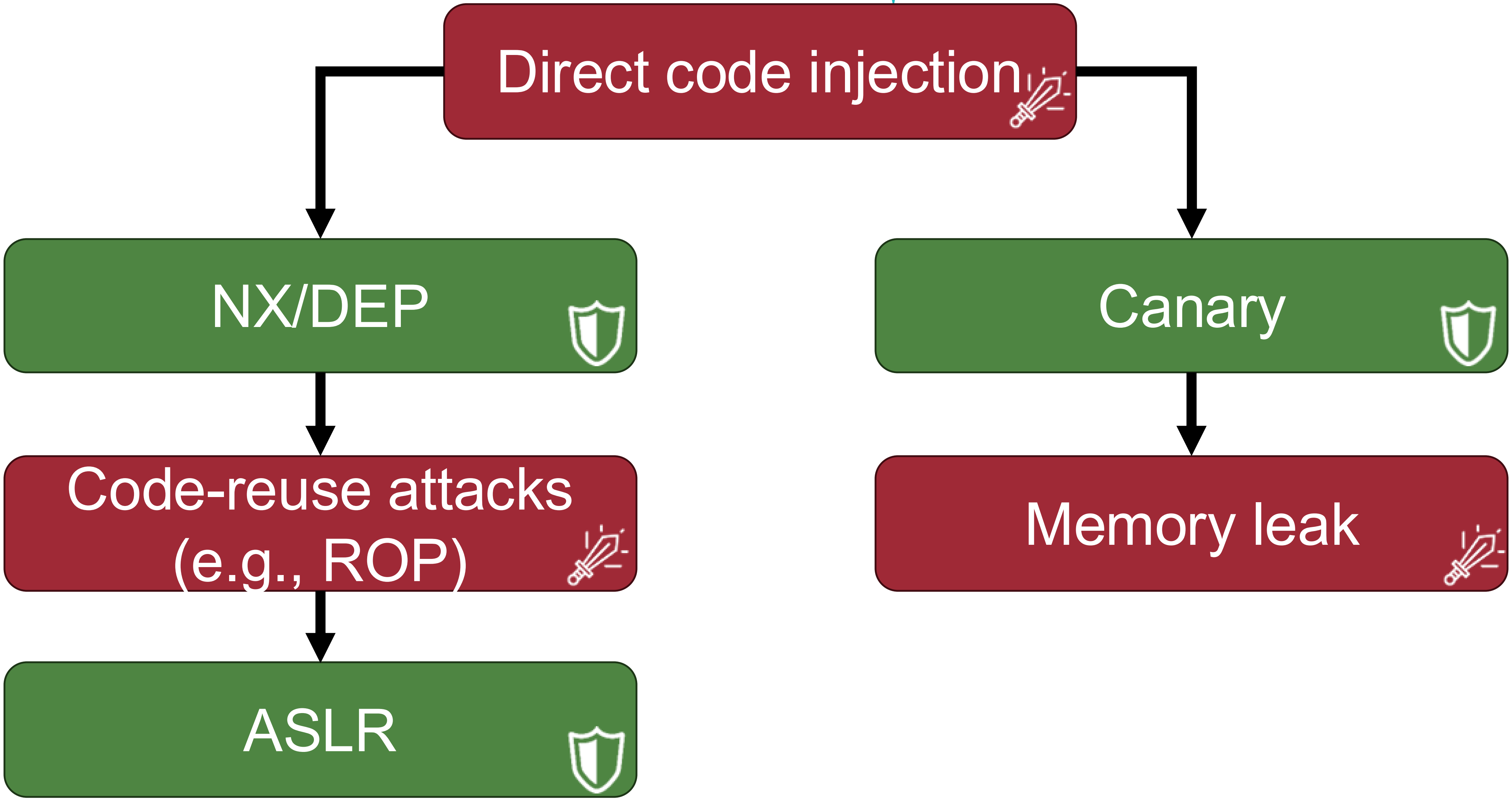
- Code Modification & Randomization
  - Defeating Return-Oriented Rootkits With “Return-less” Kernels, ***EuroSys 2010***
  - Binary stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code, ***CCS 2012***
  - Smashing the Gadgets: Hindering Return-Oriented Programming using in-Place Code Randomization, ***Oakland 2012***

- Enforcing Safety Policy
  - Control-Flow Integrity, ***CCS 2005***
  - Securing Software by Enforcing Data-Flow Integrity, ***NDSS 2014***
  - Code-Pointer Integrity, ***OSDI 2014***

DEP and Code Reuse Attacks

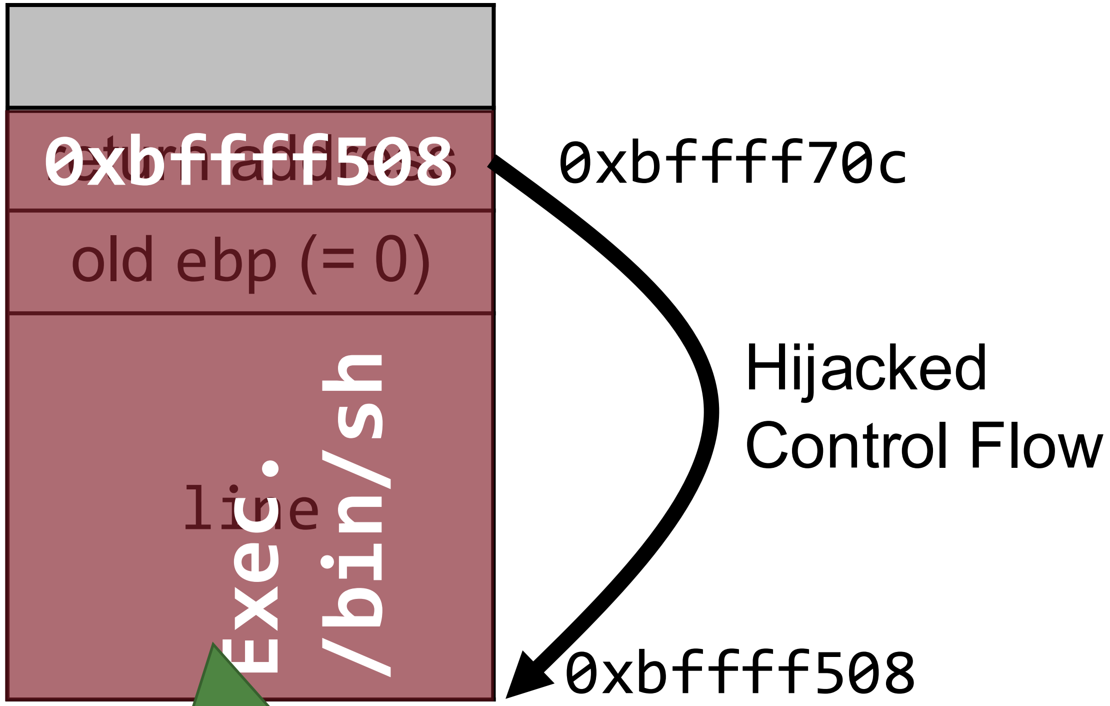


Control Hijack Attack / Defense So Far



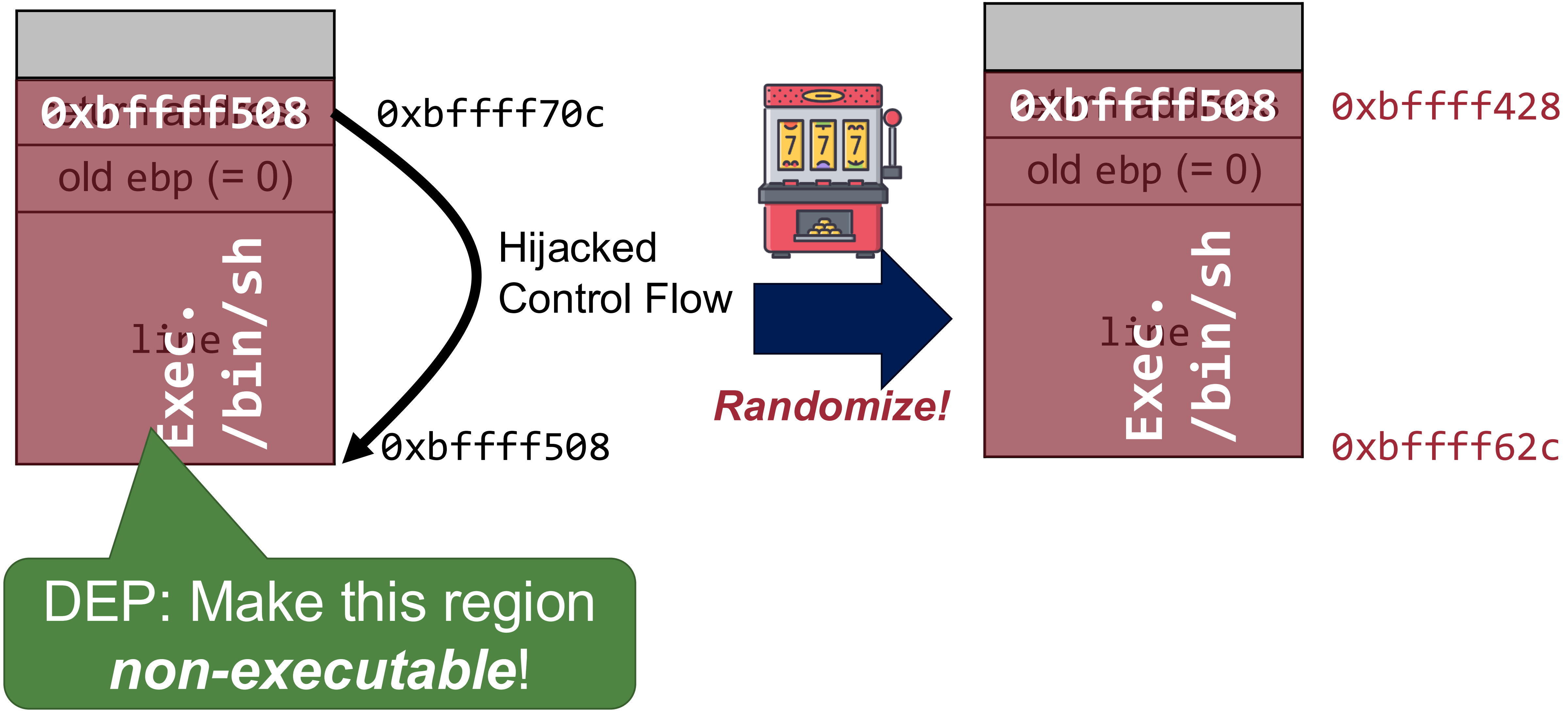
Address Space Layout Randomization (ASLR)

Control Flow Hijack Attack

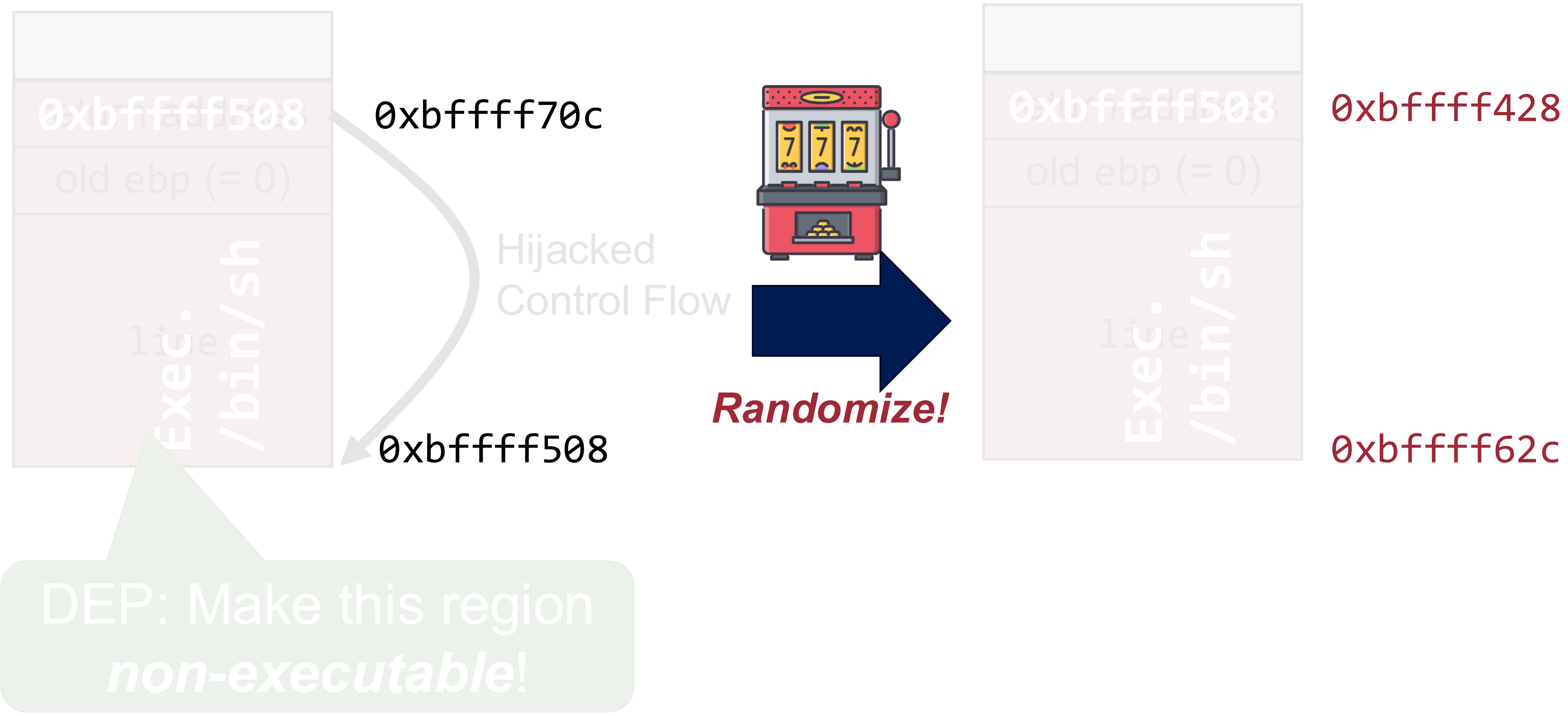


DEP: Make this region ***non-executable***!

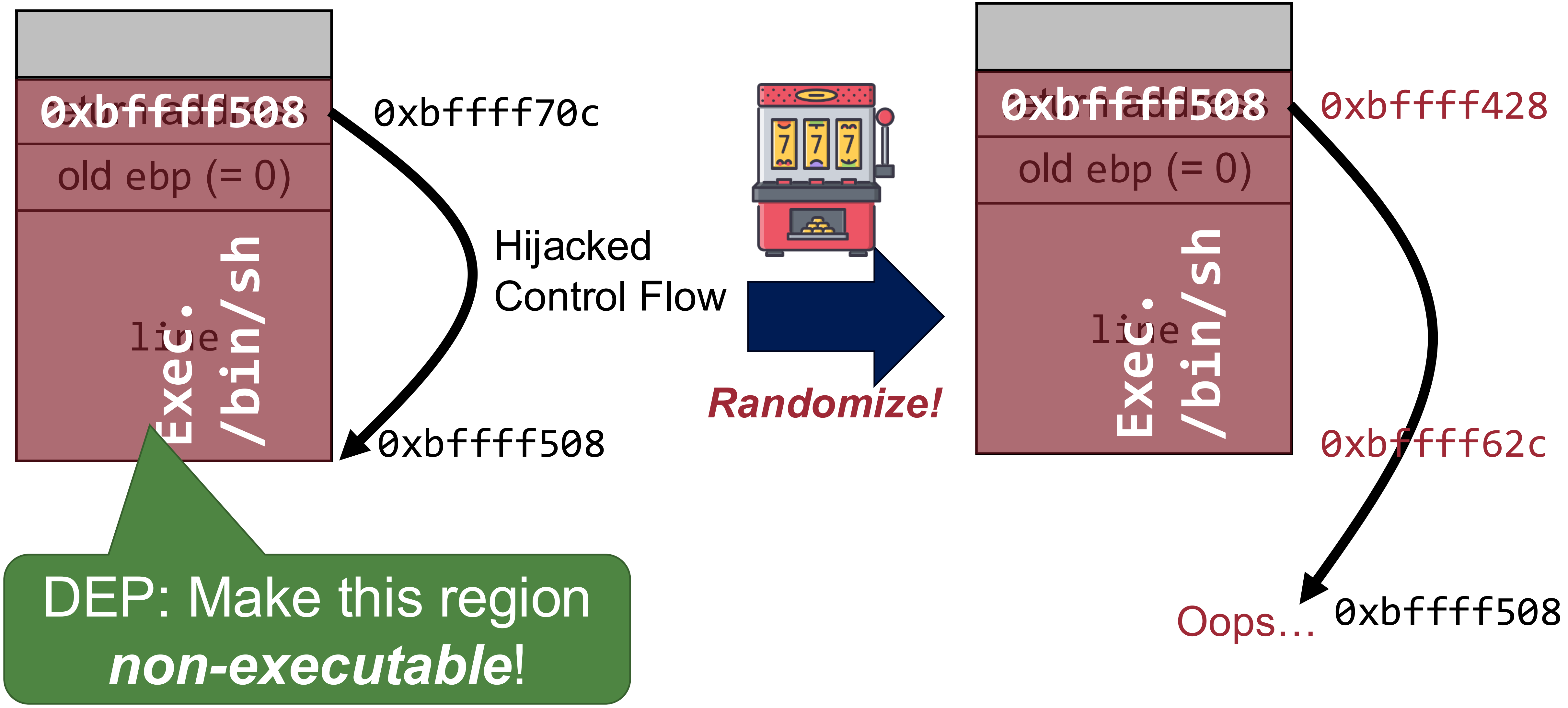
Different Perspective: ASLR



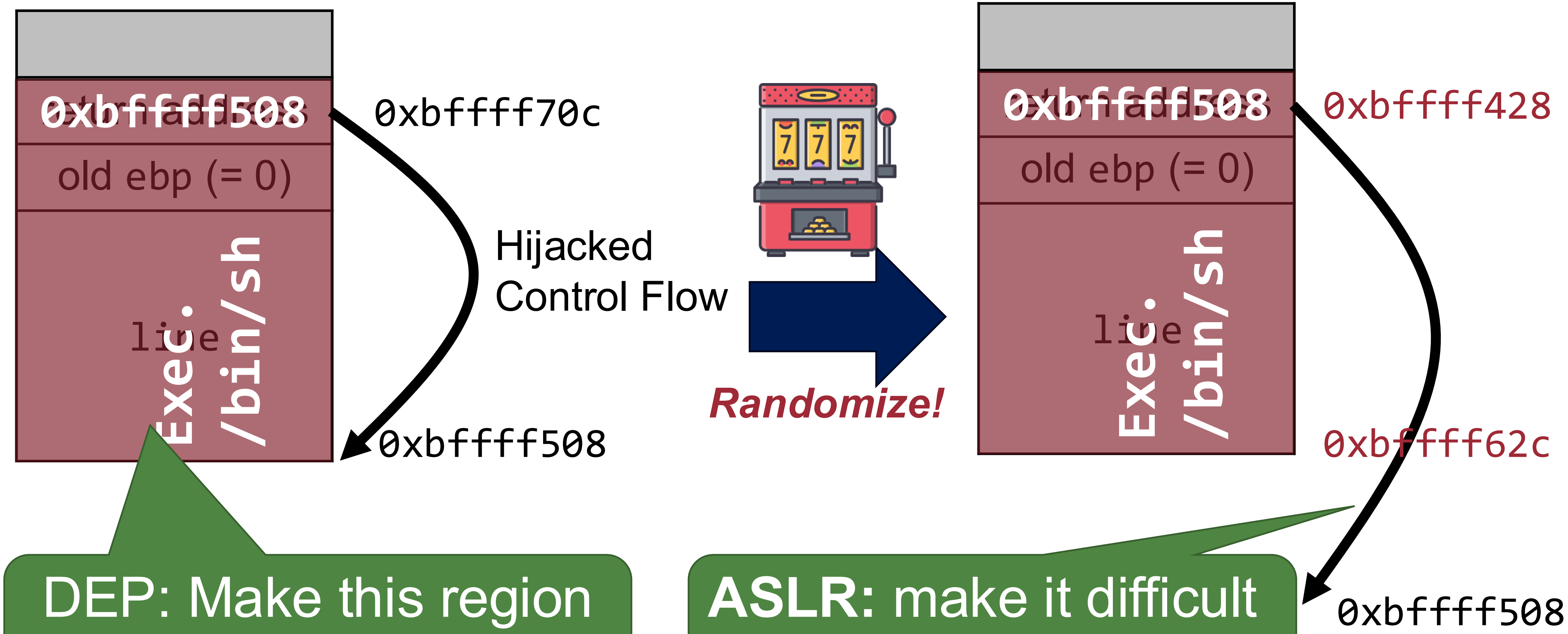
Different Perspective: ASLR



Different Perspective: ASLR



Different Perspective: ASLR



DEP: Make this region ***non-executable***!

ASLR: make it difficult to guess the address

World without ASLR

- Use the same address space over and over again!

Printing out ESP

```
#include <stdio.h>

int main (void) {
    int x = 42;
    return printf(“%08p\n”, &x);   // printing out esp
}
```

World with ASLR

- ASLR is ON by default [Ubuntu-Security]

You can enable ASLR by:

```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

World with ASLR

- ASLR is ON by default [Ubuntu-Security]

You can enable ASLR by:

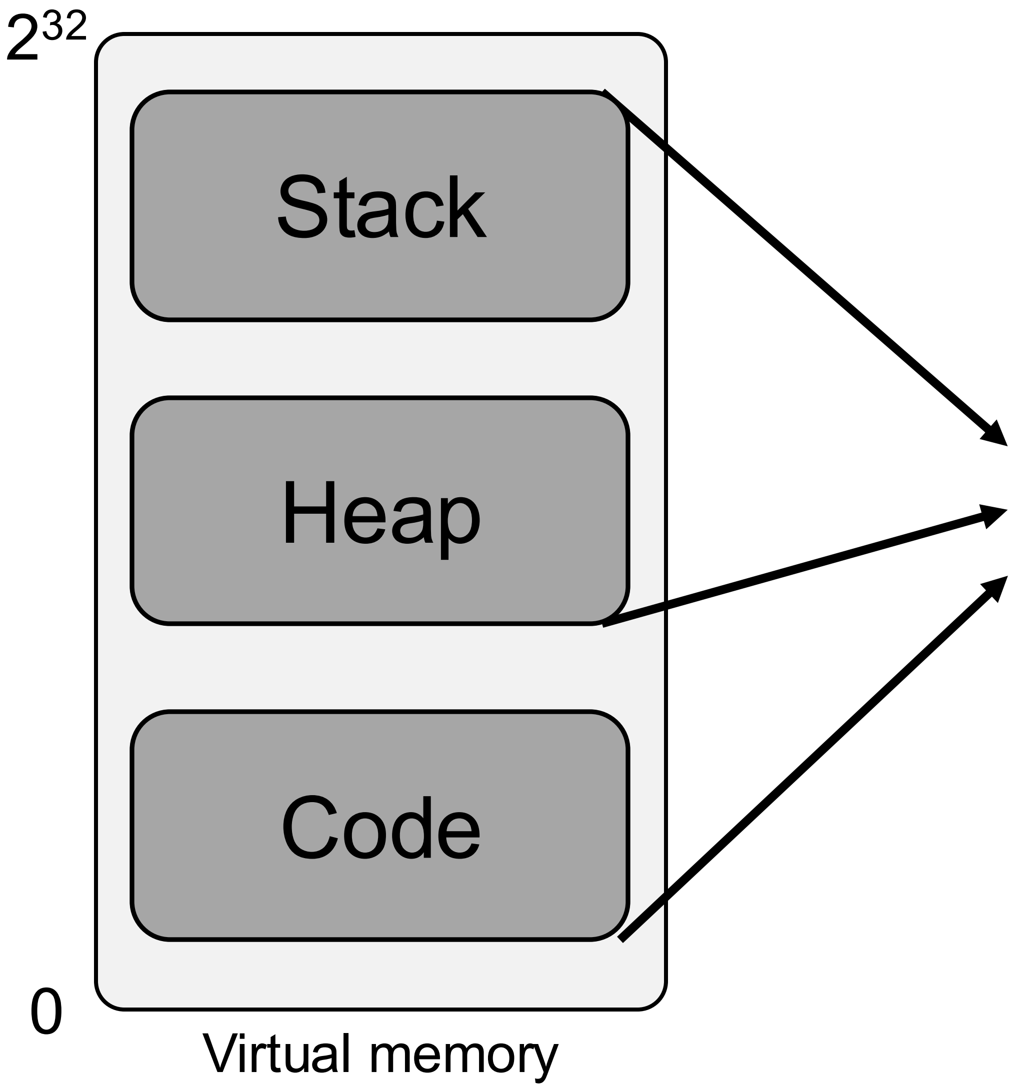
```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

Why 2?

Manual Says

Value	Description
0	Turn ASLR off
1	Make the address the stack and the library space randomized
2	Also, support heap randomization

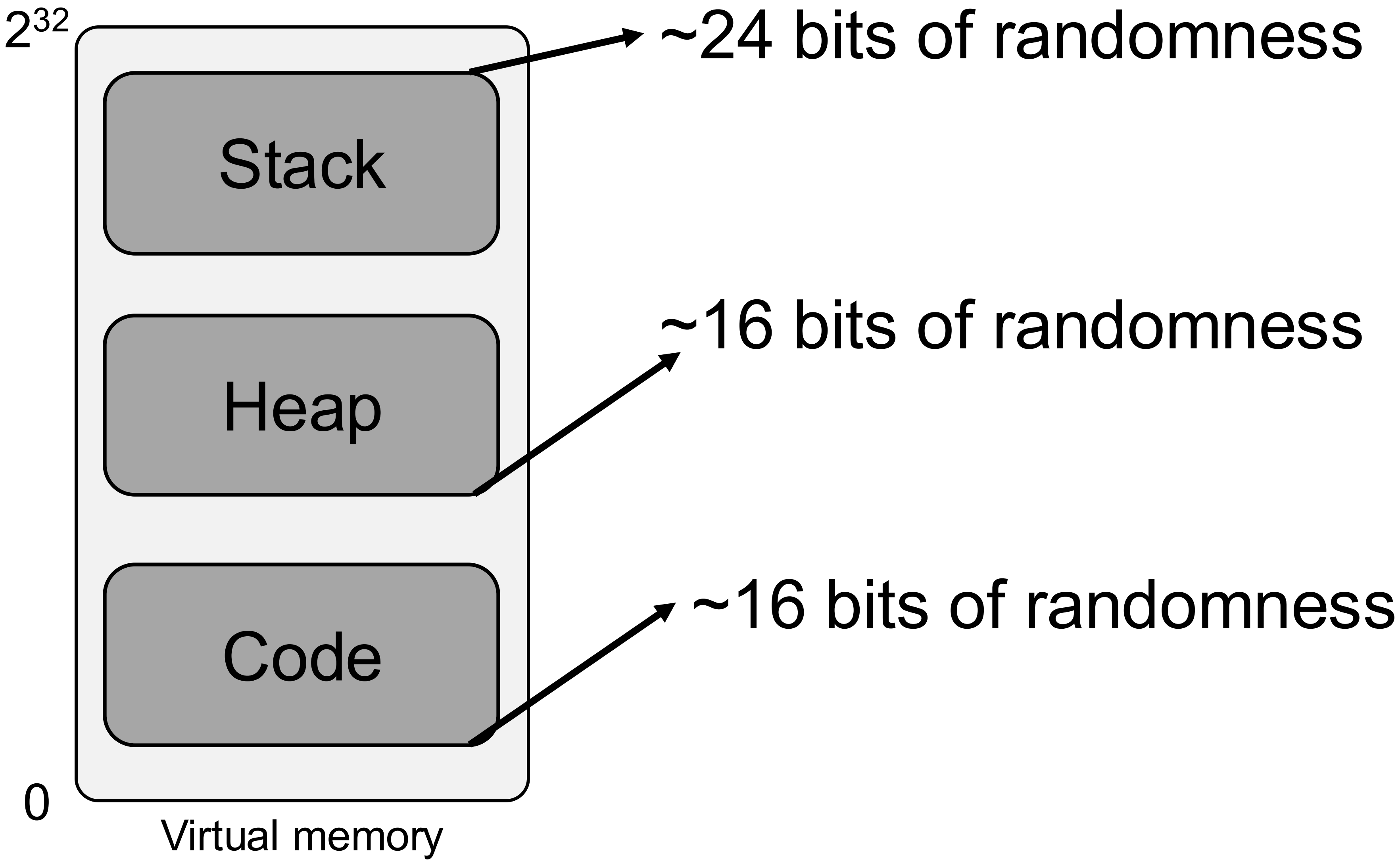
ASLR Randomizes Virtual Memory Areas



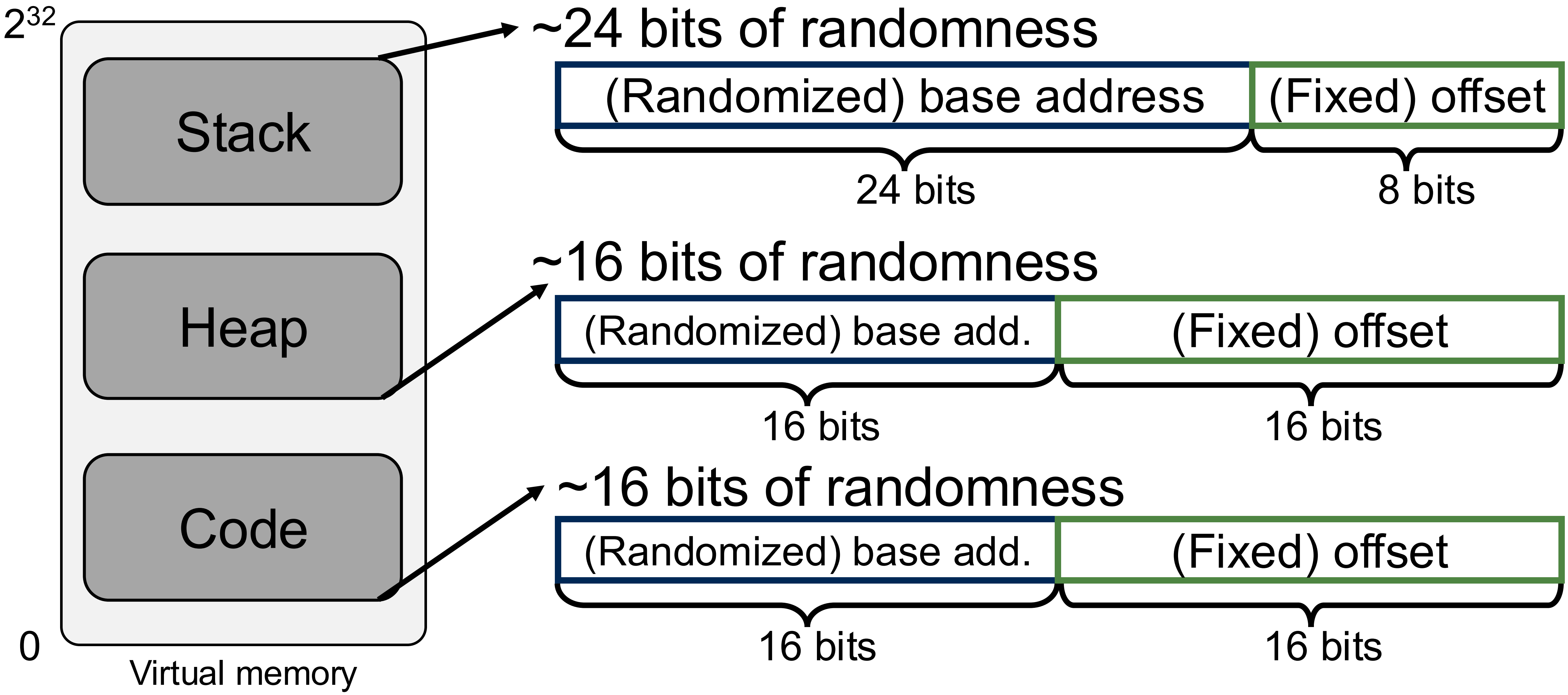
Randomized **based addresses**

The starting location for each area, also called the *base offset*, is selected randomly

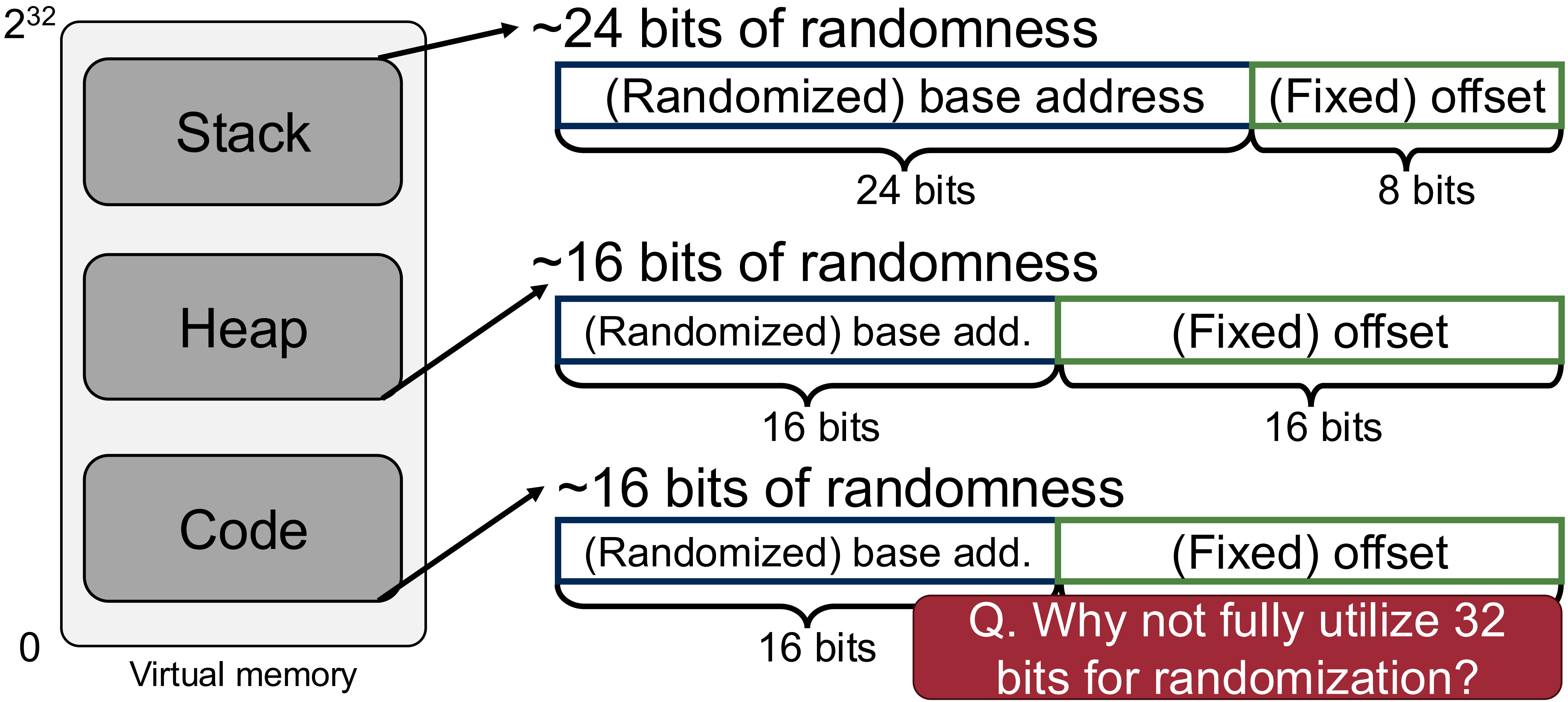
ASLR Randomizes Virtual Memory Areas



ASLR Randomizes Virtual Memory Areas

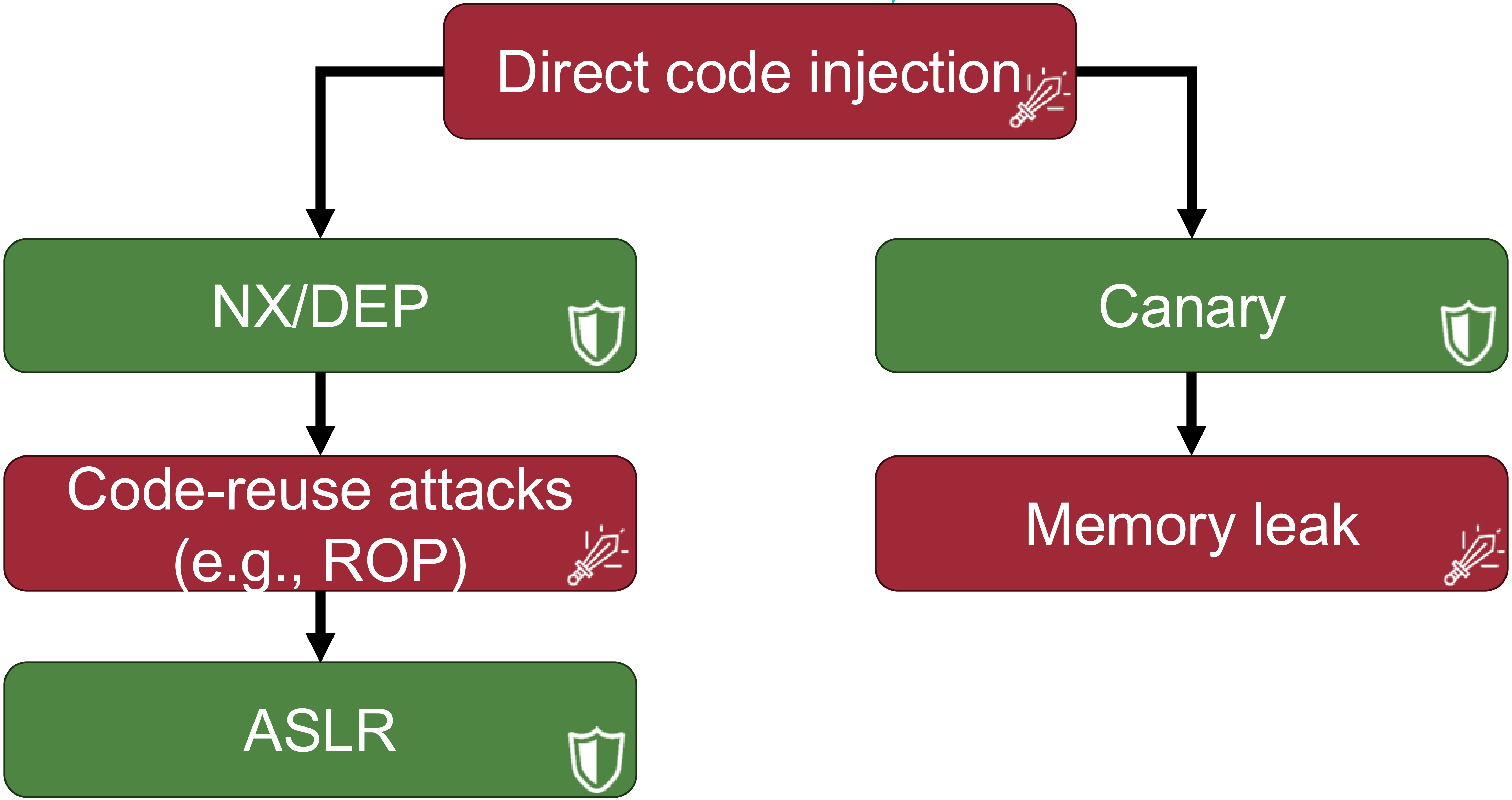


ASLR Randomizes Virtual Memory Areas



Q. Why not fully utilize 32 bits for randomization?

Control Hijack Attack / Defense So Far



Previous Exploits will *NOT* Work w/ ASLR

- ASLR will randomize the ***base addresses*** of the stack, heap, and code segments

- We cannot know the address of our shellcode nor library functions
  - Thus, no return-to-stack nor return-to-LIBC

Are we safe now?

Attacking ASLR
Part 1. Entropy

Attack #1: Entropy is Small on x86

- Just 16 bits are used for heap and libraries on x86 (Therefore, entropy is small on x86)

- **Brute-forcing** is possible for server applications that use ***fork***ing

On the Effectiveness of Address-Space Randomization, ***CCS 2004***

On the Effectiveness of Address-Space Randomization

Hovav Shacham
hovav@cs.stanford.edu

Matthew Page
mpage@stanford.edu

Ben Pfaff
blp@cs.stanford.edu

Eu-Jin Goh
eujin@cs.stanford.edu

Nagendra Modadugu
nagendra@cs.stanford.edu

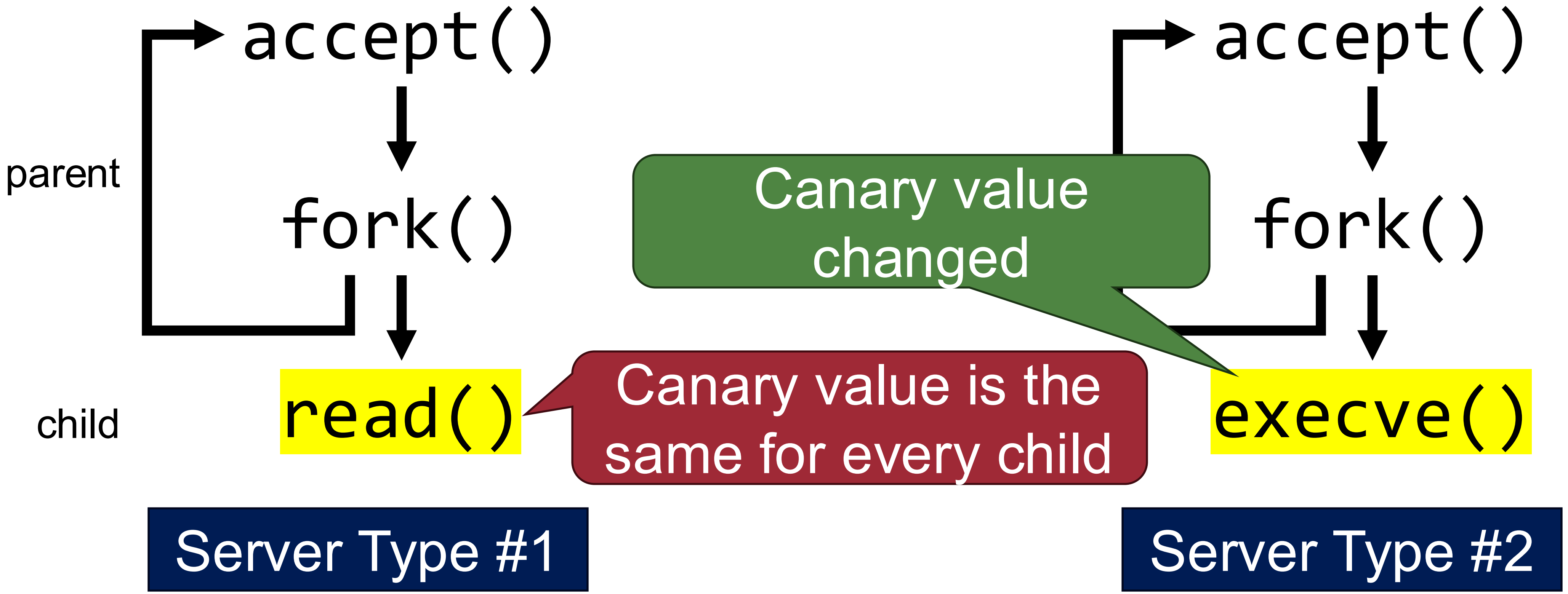
Dan Boneh
dabo@cs.stanford.edu

Abstract

Address-space randomization is a technique used to fortify systems against buffer overflow attacks. The idea is to introduce artificial diversity by randomizing the memory location of certain system components. This mechanism is available for both Linux (via PaX ASLR) and OpenBSD. We study the effectiveness of address-space randomization and find that its utility on 32-bit architectures is limited by the number of bits available for address randomization. In particular, we demonstrate a *derandomization attack* that will convert any standard buffer-overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original, albeit somewhat slower: on average 216 sec-

Recap: Reused Canary Value

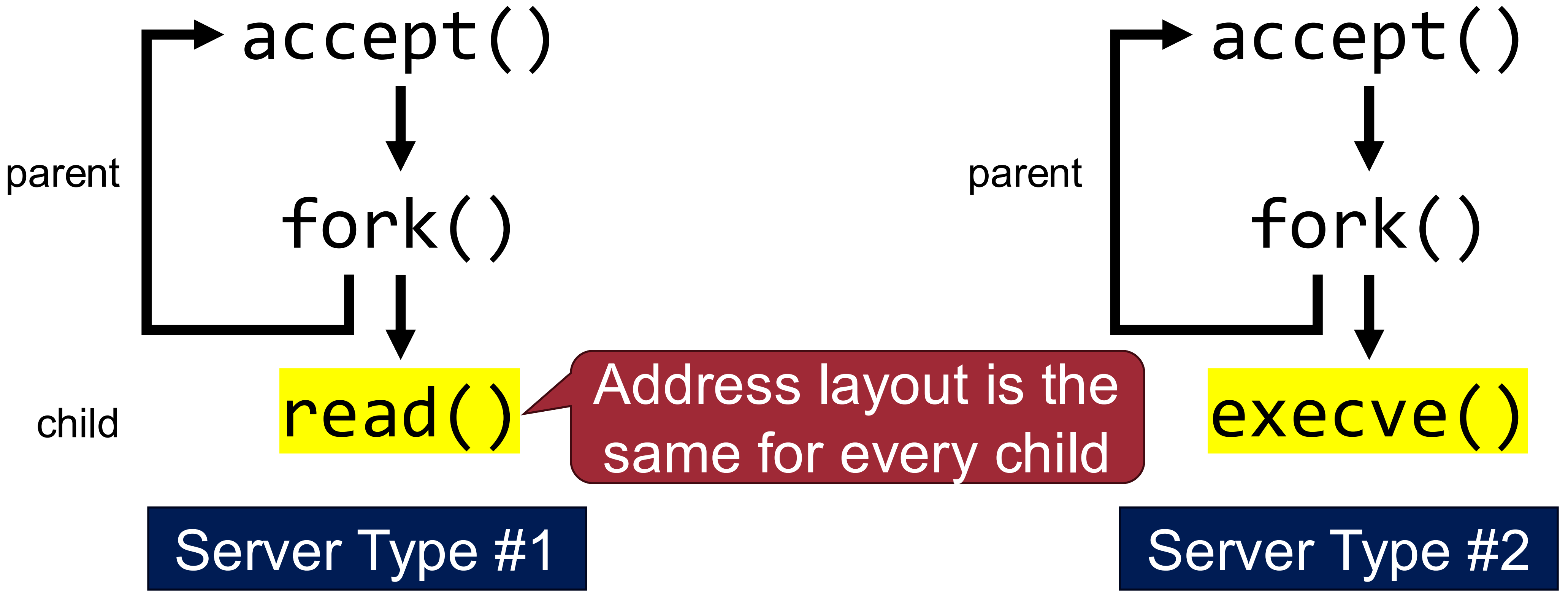
- Uses a random canary value for ***every process creation***



e.g., OpenSSH does this

Remained Address Space

- Uses a random canary value for ***every process creation***



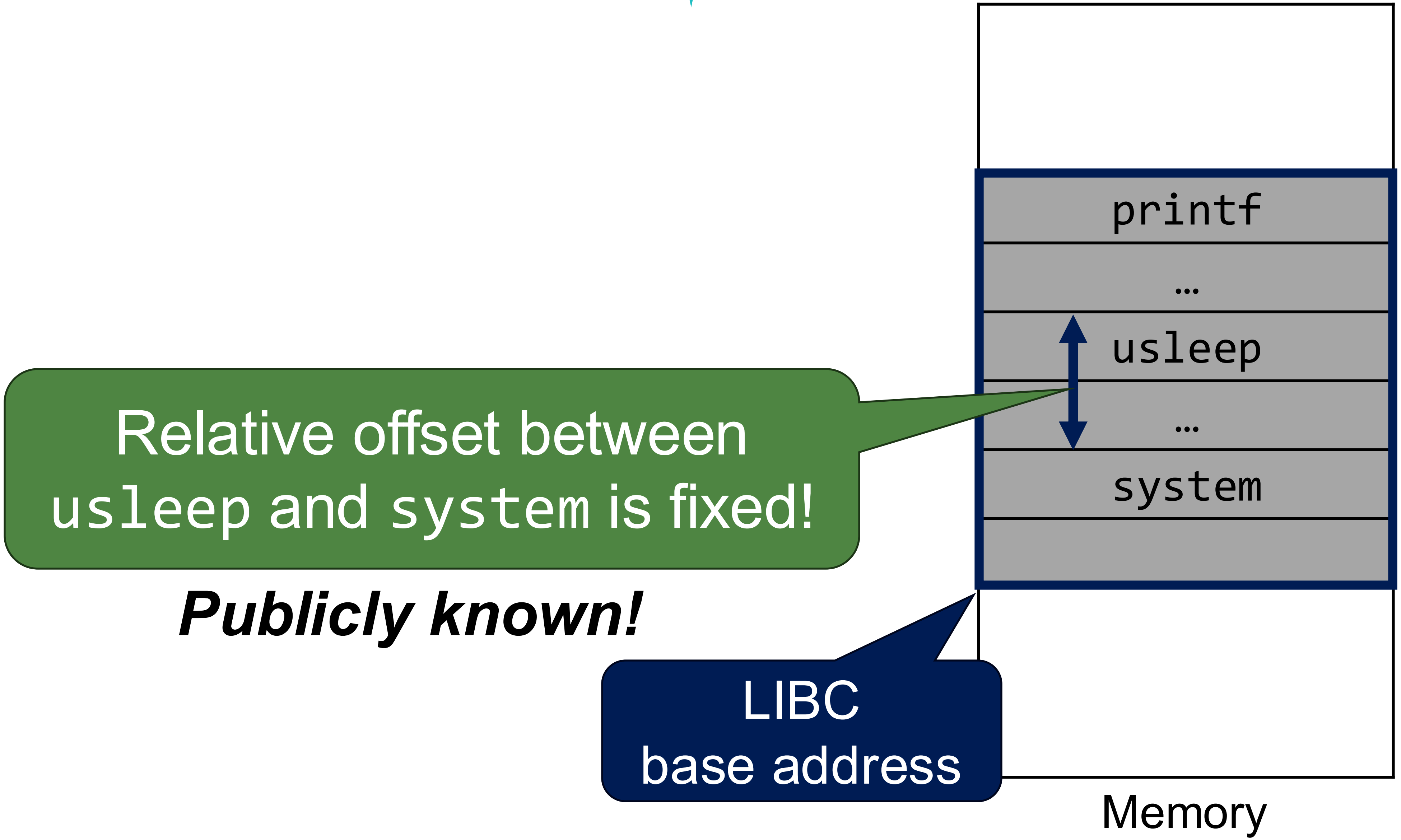
Attack #1: Entropy is Small on x86

- Just 16 bits are used for heap and libraries on x86 (Therefore, entropy is small on x86)

- **Brute-forcing** is possible for server applications that use ***fork***ing
  - Forked process has the same address space layout as its parent
  - Once we know the address of ***a function in LIBC***, we can deduce the addresses of ***all functions in LIBC***!

Key point: relative offsets between LIBC functions are the same regardless of ASLR

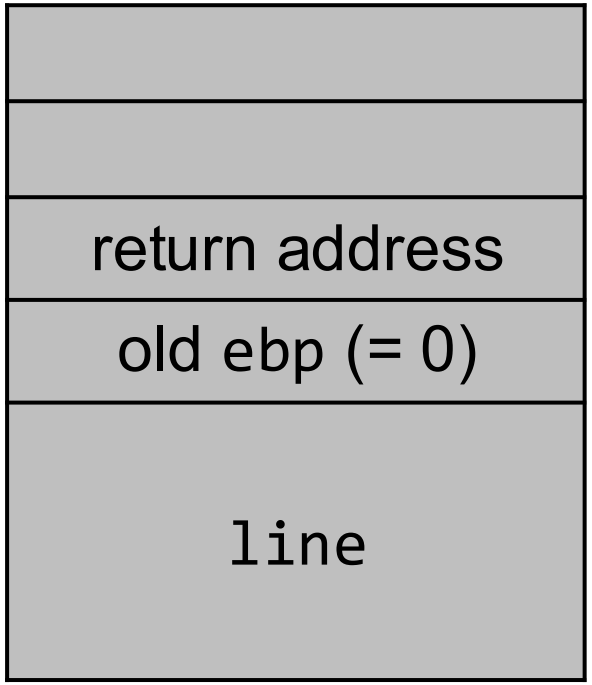
Observation: Relative Offsets are Same!



Publicly known!

Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability



Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability

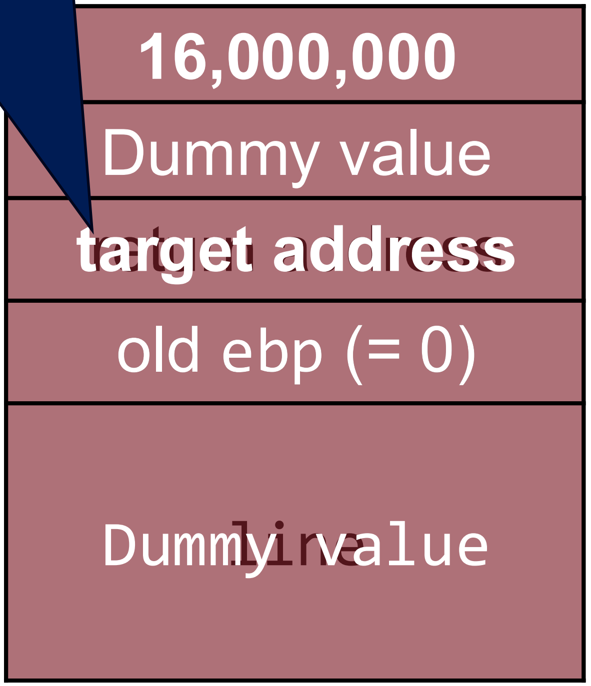
16,000,000
Dummy value
target address
old ebp (= 0)
Dummy value

Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow

Brute-force on 16 bits to find the address of `usleep`

- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)



Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow

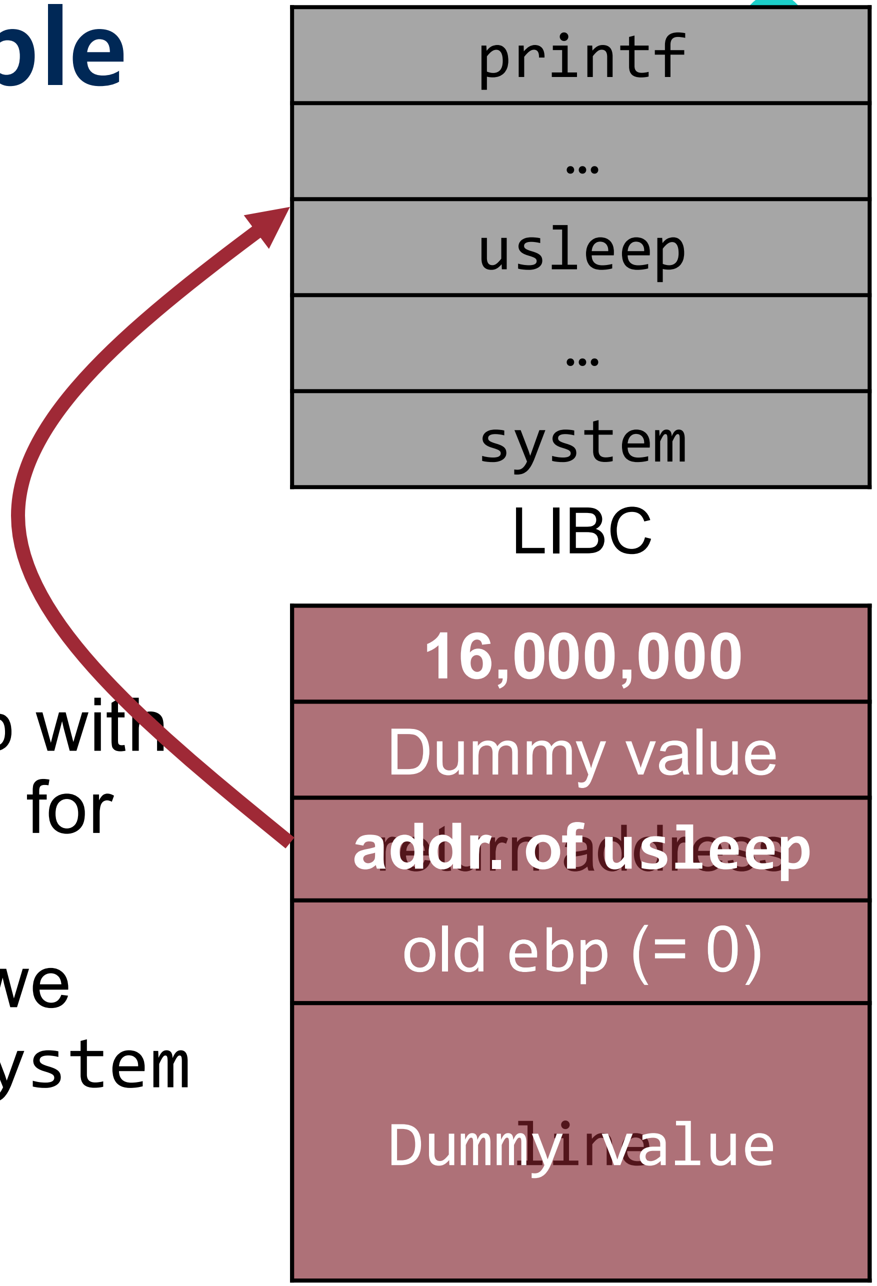
If correct, the server will wait 16 seconds

- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)

16,000,000
Dummy value
addr. of usleep
old ebp (= 0)
Dummy value

Brute-forcing Attack Example

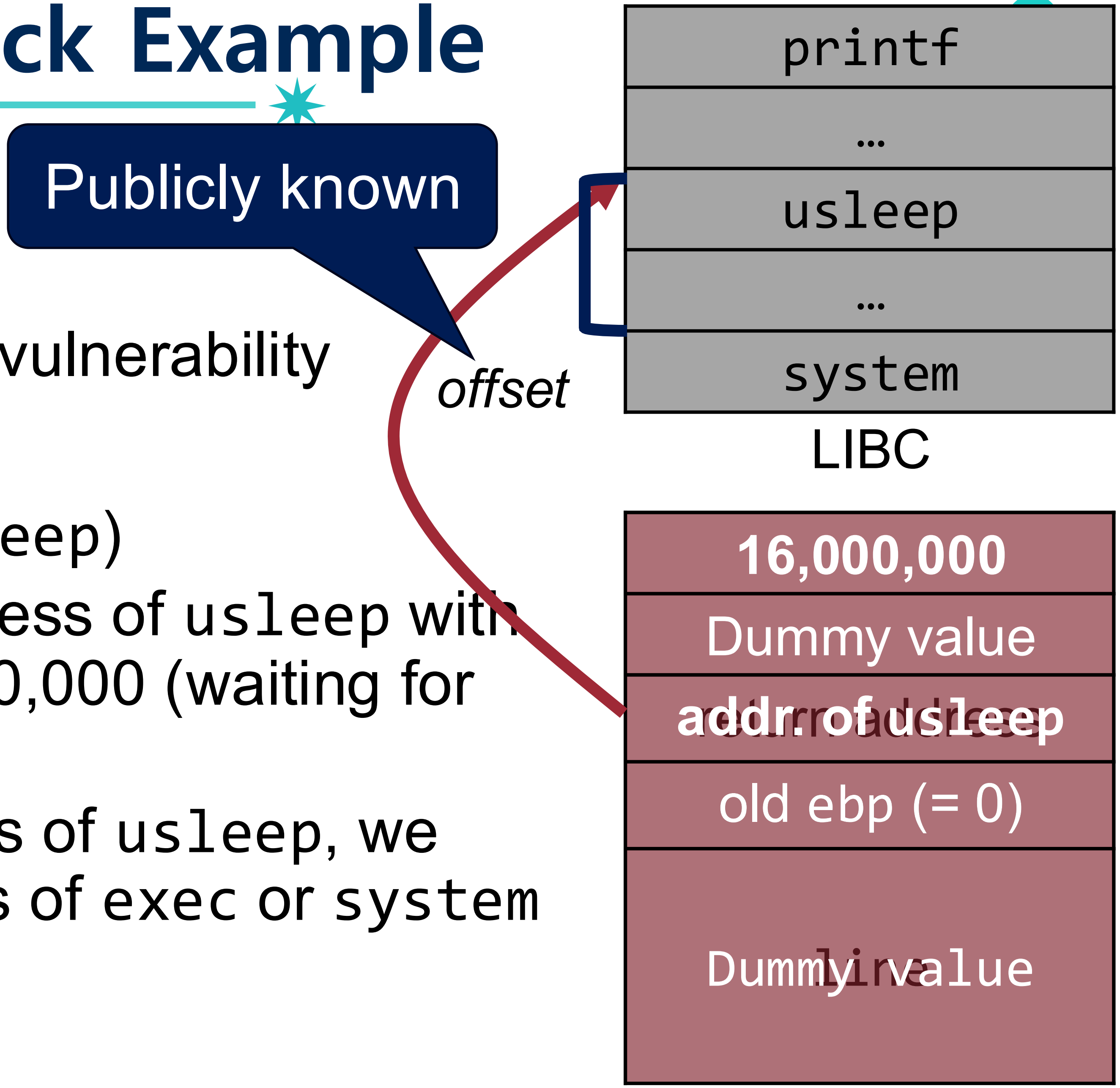
- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability
- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)
  - Once we know the address of `usleep`, we can determine the address of `exec` or `system`



Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability

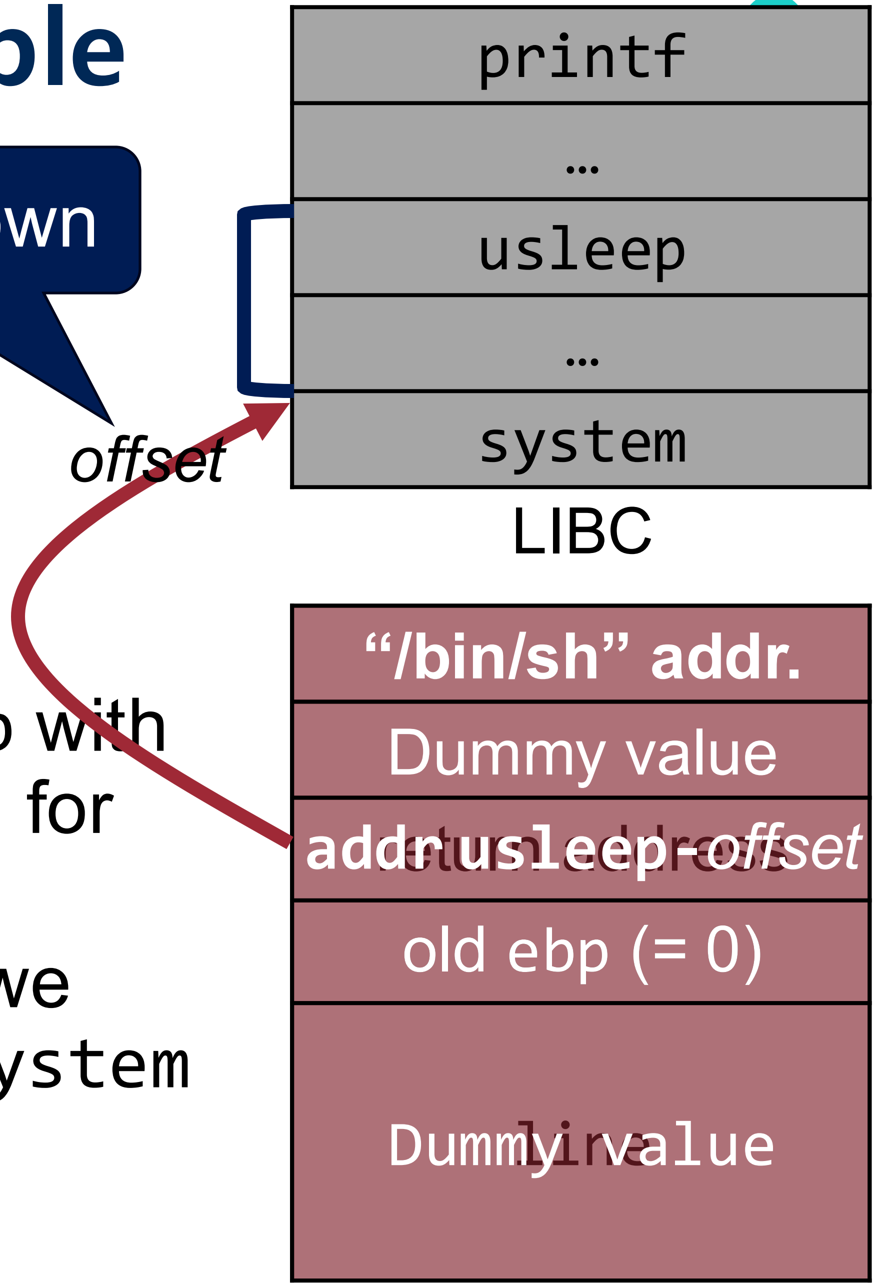
- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)
  - Once we know the address of `usleep`, we can determine the address of `exec` or `system`



Brute-forcing Attack Example

Publicly known

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability
- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)
  - Once we know the address of `usleep`, we can determine the address of `exec` or `system`



Randomization Frequency on Two Major OSes

- On ***Windows***: every time the machine starts
  - Each module will get a random address once per boot (but, stack and heap will be randomized per execution)
- On ***Linux***: every time a process loads
  - Each module will get a random address for every execution

Which one is better?

Performance: Which One is Better?

- On ***Windows***: every time the machine starts
  - Each modul will get a random address <u>once per boot</u> (but, stack an will be randomized per execution)

Faster: relocation once at boot time

- On ***Linux***: every time a process loads
  - Each modu ill get a random address <u>for every execution</u>

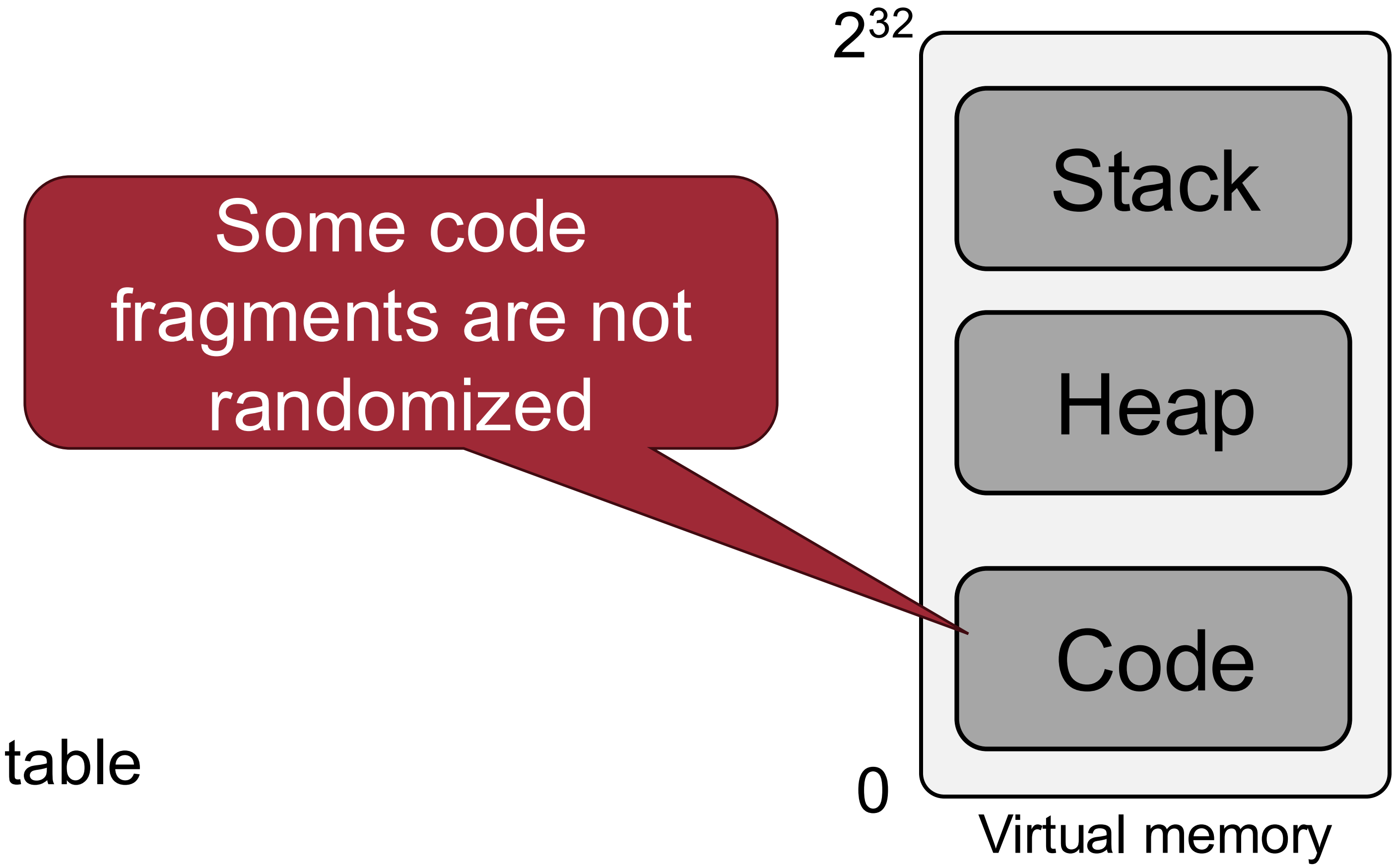
Slower: relocation fixups for every execution

How about security?

Attacking ASLR
Part 2. Exploiting Fixed Addresses

Attack 2: Exploiting Fixed Addresses

- Most binaries (before 2016) had non-randomized segments
  - Before 2016, compilers created ***non-PIE***[1] executables by default



[1]Non Position-Independent Executable

Position-Independent Executable (PIE)

- Position-Independent Code (PIC) or PIE is code that runs regardless of its location (e.g., shellcode)
  - "`gcc`" will produce a PIE by default
  - "`gcc -fno-pic -no-pie`" will produce a non-PIE

PIE vs. non-PIE

- Position-Independent Code (PIC) or PIE is code that runs regardless of its location (e.g., shellcode)
  - "`gcc`" will produce a PIE by default
  - "`gcc -fno-pic -no-pie`" will produce a non-PIE

```
080491ba <main>:
 80491ba: lea    ecx,[esp+0x4]
 80491be: and    esp,0xfffffff0
 80491c1: push   DWORD PTR [ecx-0x4]
 80491c4: push   ebp
 80491c5: mov    ebp,esp
 80491c7: push   ecx
 80491c8: sub    esp,0x14
 80491cb: mov    eax,gs:0x14
```

$ gcc -fno-pic -no-pie

```
000011f1 <main>:
    11f1: lea    ecx,[esp+0x4]
    11f5: and    esp,0xfffffff0
    11f8: push   DWORD PTR [ecx-0x4]
    11fb: push   ebp
    11fc: mov    ebp,esp
    11fe: push   ebx
    11ff: push   ecx
    1200: sub    esp,0x10
```

$ gcc (Produce a PIE)

PIE vs. non-PIE

Non-randomized segments even when ASLR is turned on

Relative addresses – randomized when ASLR is turned on

```
080491ba <main>:
 80491ba: lea    ecx,[esp+0x4]
 80491be: and    esp,0xfffffff0
 80491c1: push   DWORD PTR [ecx-0x4]
 80491c4: push   ebp
 80491c5: mov    ebp,esp
 80491c7: push   ecx
 80491c8: sub    esp,0x14
 80491cb: mov    eax,gs:0x14
```

$ gcc -fno-pic -no-pie

```
000011f1 <main>:
    11f1: lea    ecx,[esp+0x4]
    11f5: and    esp,0xfffffff0
    11f8: push   DWORD PTR [ecx-0x4]
    11fb: push   ebp
    11fc: mov    ebp,esp
    11fe: push   ebx
    11ff: push   ecx
    1200: sub    esp,0x10
```

$ gcc (Produce a PIE)

Legacy Binaries Are Not a PIE

- 93% of Linux binaries were not a PIE (in 2009)
- Thus, the code sections were not randomized

But, why?

Security vs. Performance

- Relative-addressing instructions are slower than absolute-addressing instructions

- Performance overhead of PIE on x86 is 10% on average

  (Too much PIE is bad for performance, ETH Techreport, 2012)

- Most applications on current x86 are still not PIEs

ROP-based Attack on Legacy Binaries

- Some code fragments are not randomized!

→ Why not use ROP on them

Fact: relative offsets between LIBC functions are the same regardless of ASLR

Exploitation Idea

- If a LIBC function has been invoked at least once, GOT should contain a concrete address of the function in LIBC
- Therefore, we will read the GOT entry using ROP and compute the address of `system` by using the relative offset between the LIBC function and `system`

Suppose we can get the address of open function from the GOT

(addr of `system`) = (addr of `open`)
+ (offset from `open` to `system` in LIBC)

Example ROP

(addr of `system`) = (addr of `open`)
 + (offset from `open` to `system` in LIBC)

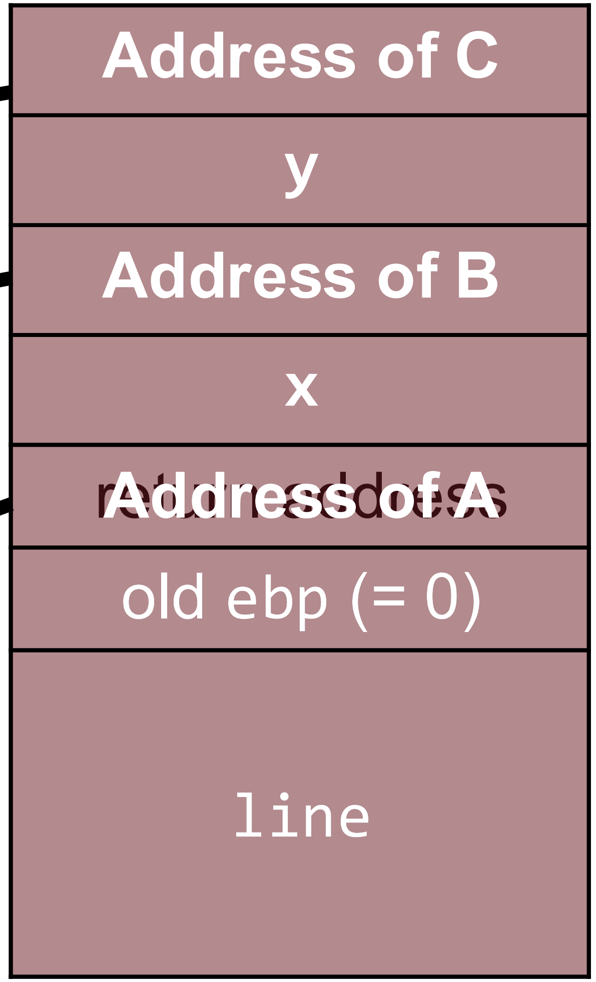
x

y

Gadget **C**
```
jmp [eax]
```

Gadget **B**
```
pop eax
add eax, edi
ret
```

Gadget **A**
```
pop edi
ret
```



Possible Defenses?

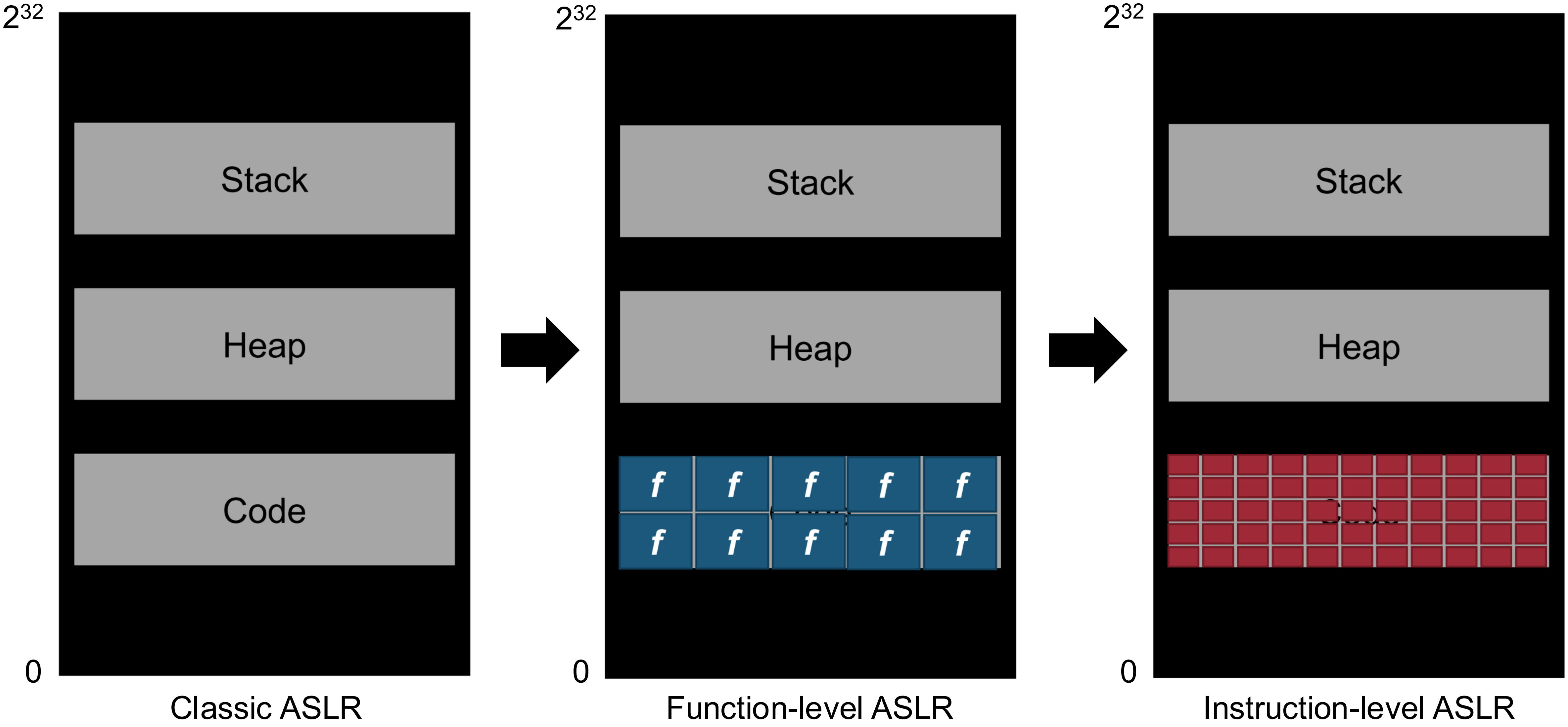
- Use PIEs
- Use 64-bit CPU: lots of entropy
- Detect brute-forcing attacks
  - Many crashes in a short amount of time
- Use non-forking servers
- Code randomization (a.k.a. fine-grained ASLR)

Code Randomization

Motivation

- ASLR only changes base addresses of VMAs
  - A single pointer leak can reveal the entire memory layout of a VMA

Fine-Grained ASLR (Not Used in Practice)

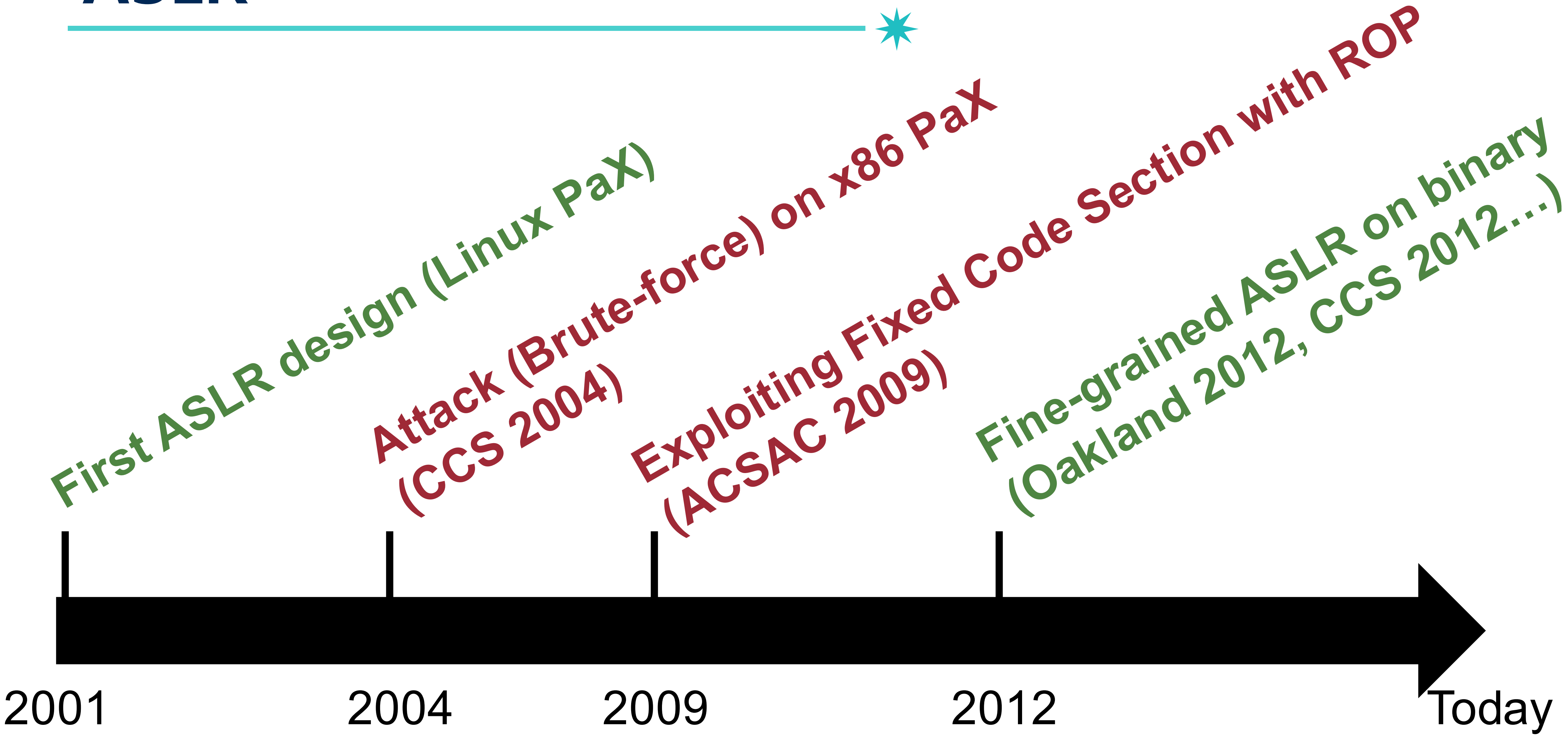


Fine-Grained ASLR (Not Used in Practice)

- Source-based approaches:
  - Efficient Techniques for Comprehensive Protection from Memory Error Exploits, ***USENIX Security 2005***
  - Enhanced Operating System Security through Efficient and Fine-grained Address Space Randomization, ***USENIX Security 2012***

- Binary-based approaches:
  - Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization, **S&P 2012**
  - ILR: Where'd My Gadgets Go?, **S&P 2012**
  - Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code, **CCS 2012**

ASLR


First ASLR design (Linux PaX)
Attack (Brute-force) on x86 PaX
(CCS 2004)
Exploiting Fixed Code Section with ROP
(ACSAC 2009)
Fine-grained ASLR on binary
(Oakland 2012, CCS 2012...)
2001
2004
2009
2012
Today

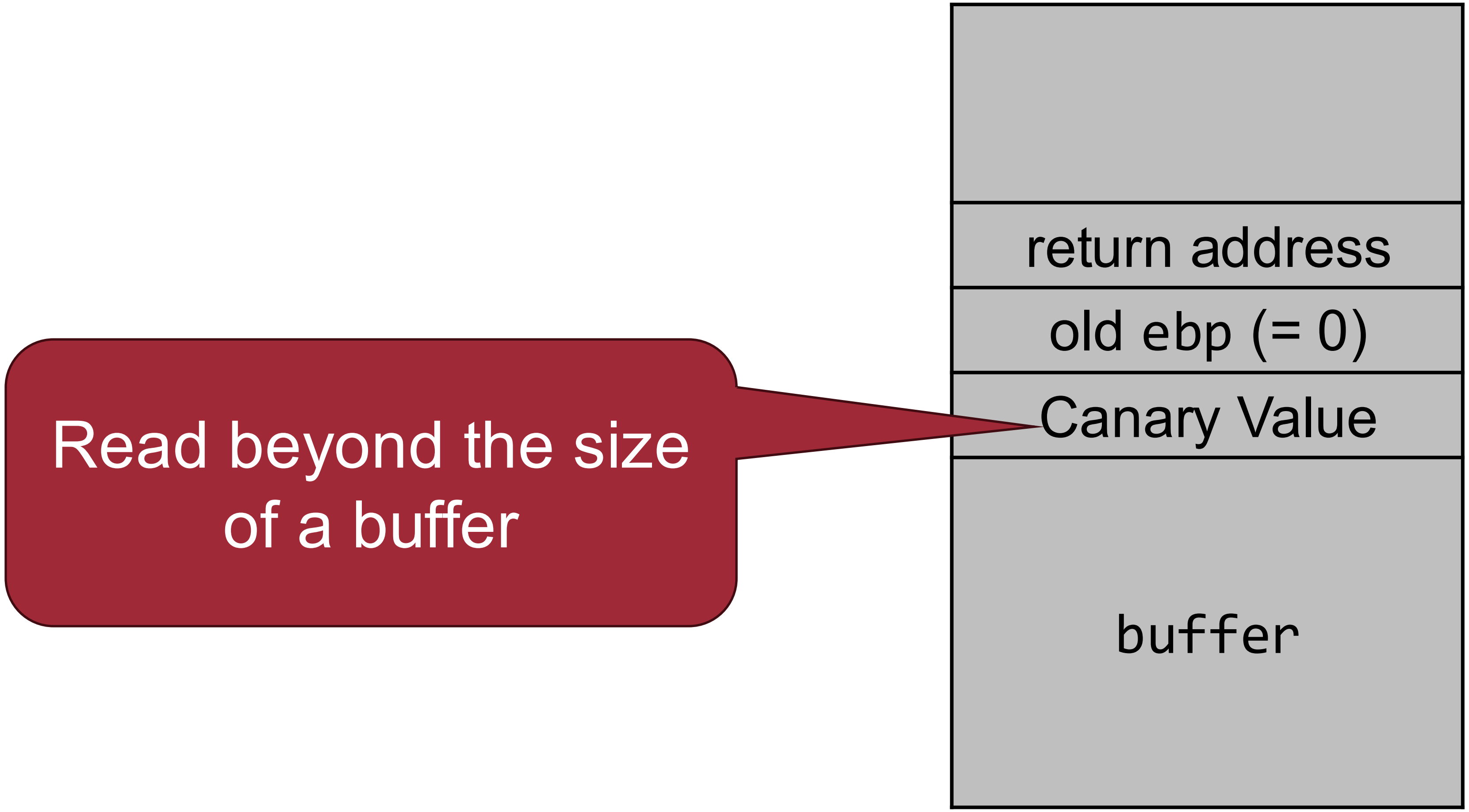
Memory Disclosure

Memory Disclosure ≠ Memory Corruption

Memory disclosure does not necessarily involve memory corruption

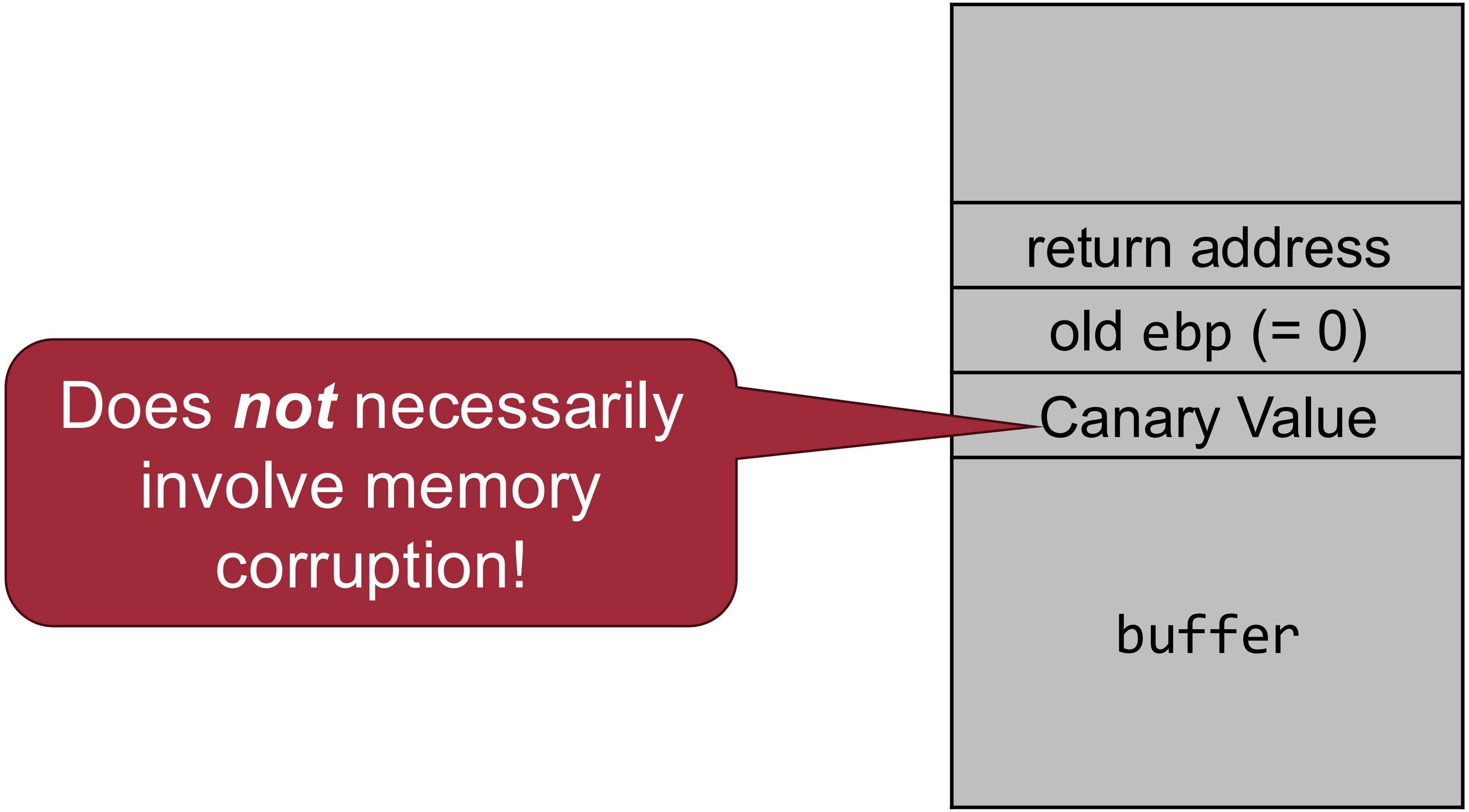
Buffer Over-Read

Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer



Buffer Over-Read

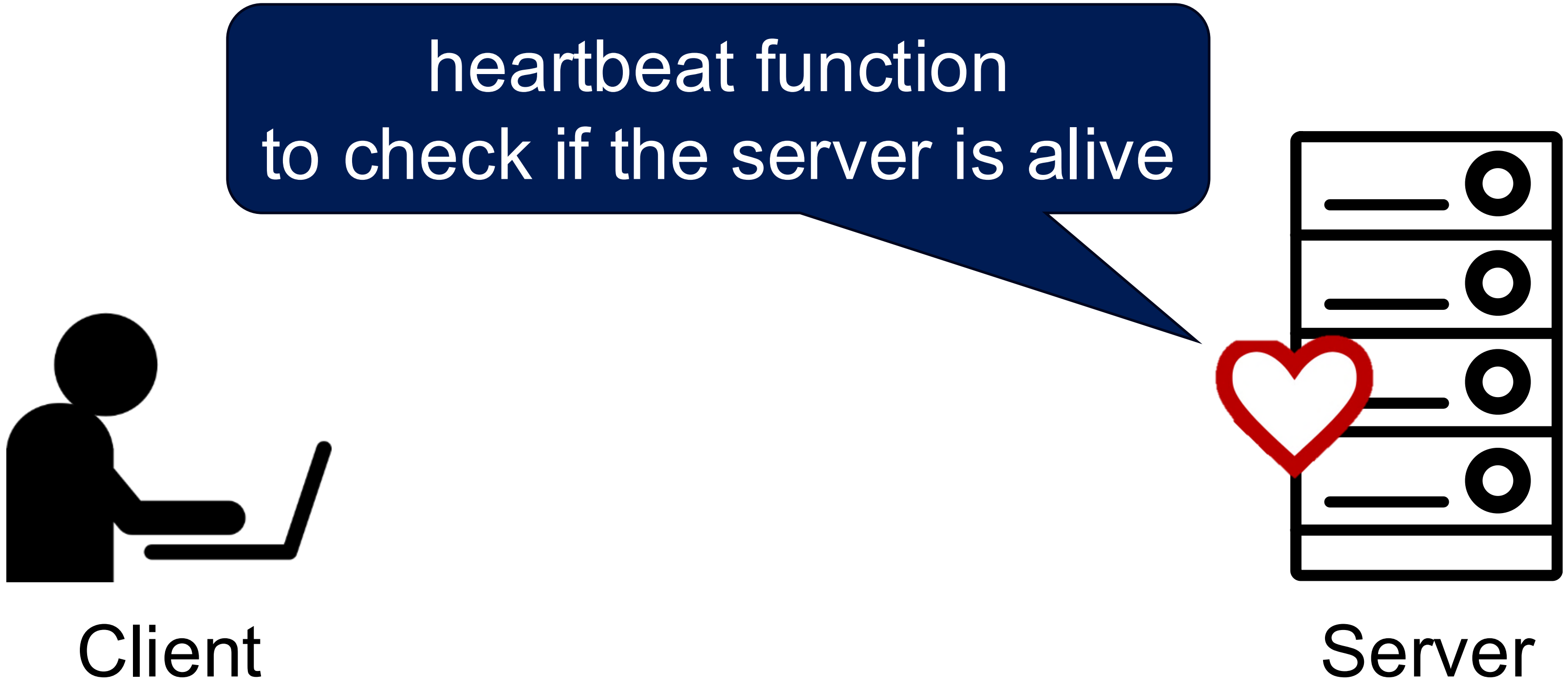
Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer



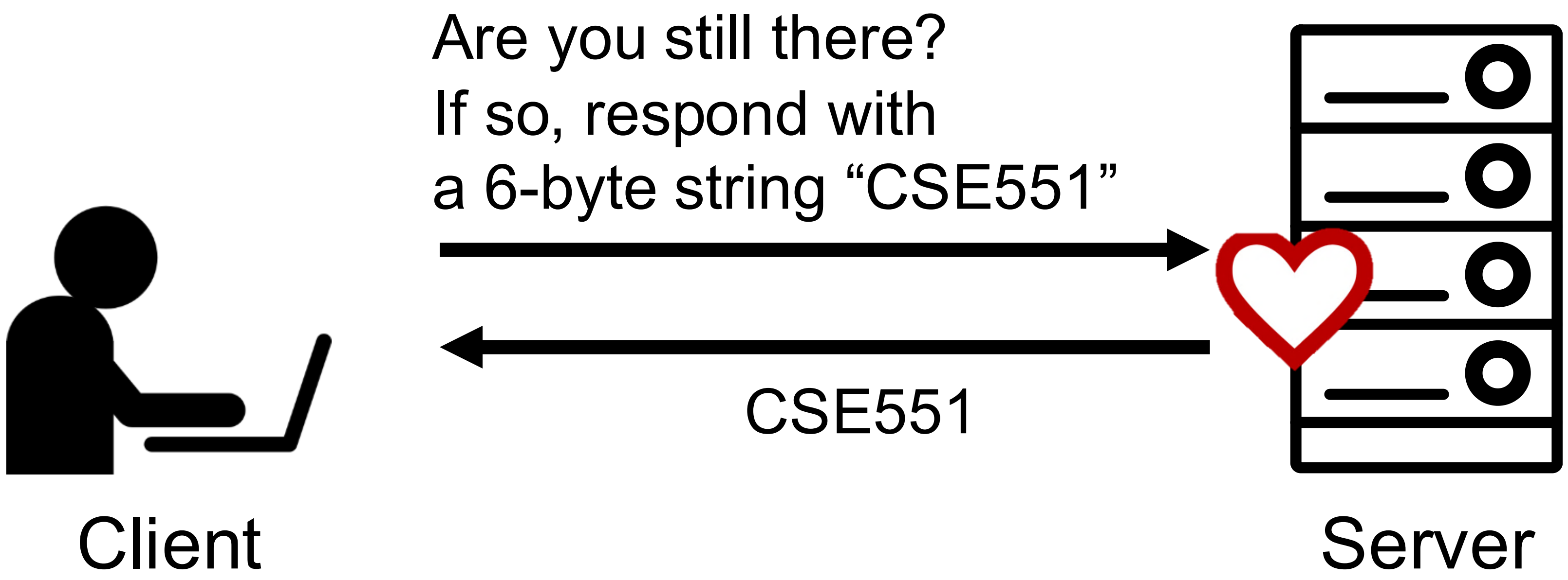
Example: Heartbleed Bug (in 2014)

- Famous bug in OpenSSL (in TLS ***heartbeat***)
- An attacker can steal private keys

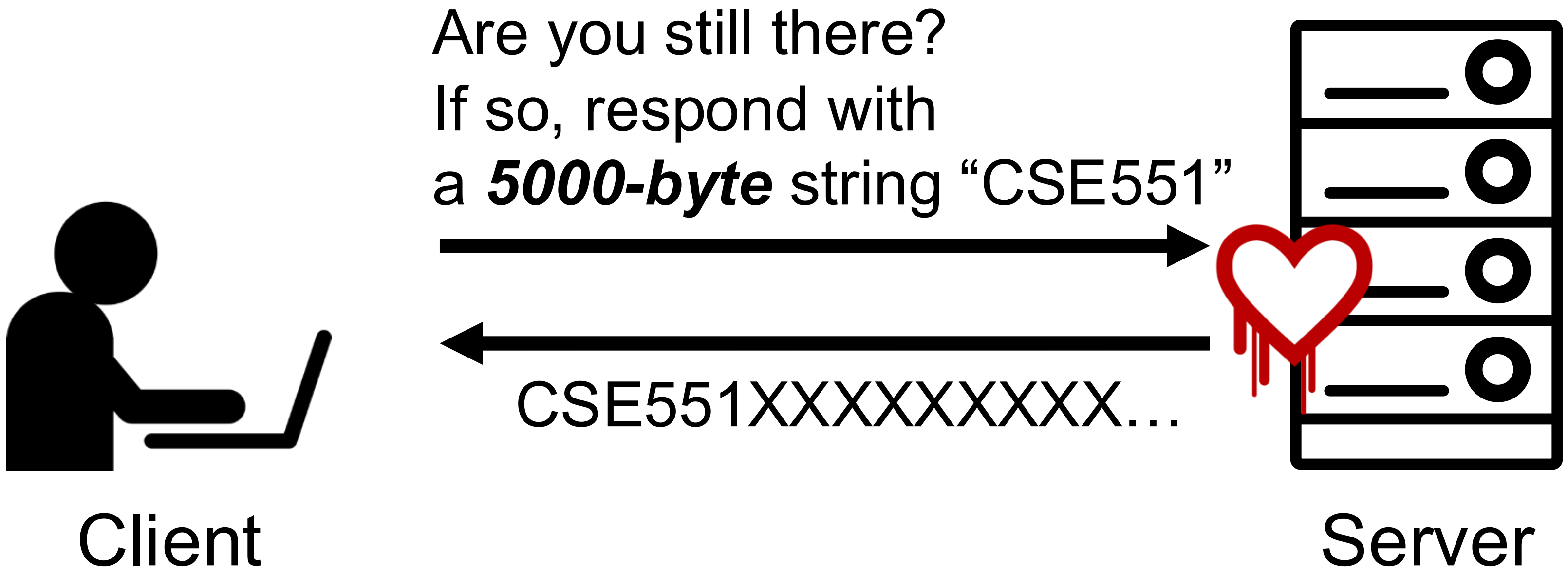
Heartbleed Bug: High-level Workflow



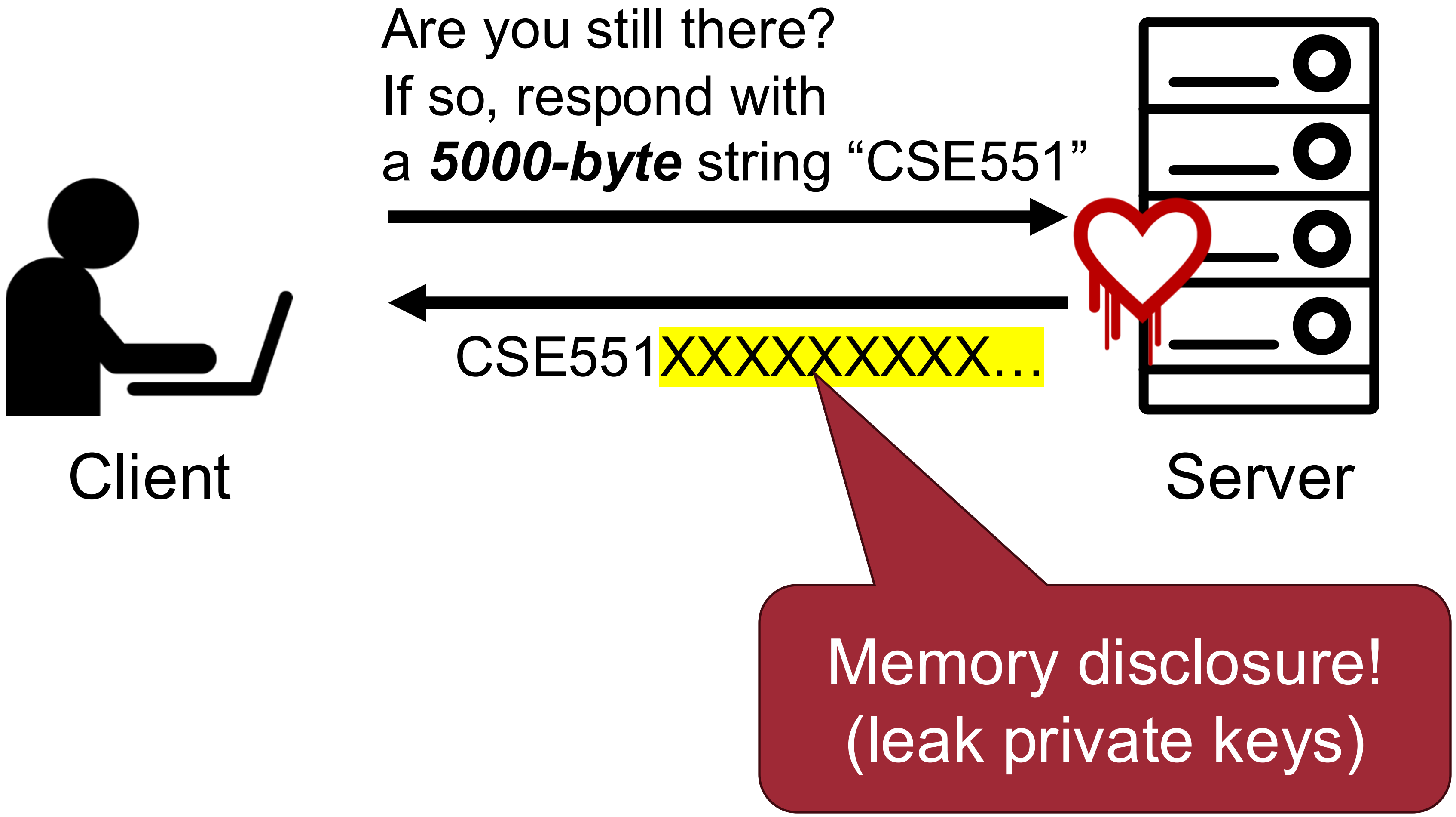
Heartbleed Bug: High-level Workflow



Heartbleed Bug: High-level Workflow



Heartbleed Bug: High-level Workflow



The Bug

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;

struct {
    unsigned int length;
    unsigned char *data;
    ...
} SSL3_RECORD;
```

The Bug

```
struct {
    HeartbeatMessageType typ
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque paddi
} HeartbeatMessa

struct {
    unsigned int length;
    unsigned char *data;
    ...
} SSL3_RECORD;

memcpy(bp, pl, length);  // vulnerable spot!
```

Calculated from the user's payload (i.e., 6)

Payload obtained from HeartbeatMessage (i.e., CSE551)

Obtained from the user's input (i.e., 5000)

Copy arbitrary memory contents of a server! TLS private key may be available

The Bug

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;
```

Calculated from the user's payload (i.e., 6)

Payload obtained from HeartbeatMessage (i.e., CSE551)

```
struct {
    unsigned int length;
    unsigned char *data;
    ...
} SSL3_RECORD;
```

Obtained from the user's input (i.e., 5000)

Root cause:
Did not check the consistency of the values of the two variables!

```
memcpy(bp, pl, length);  // vulnerable spot!
```

Copy arbitrary memory contents of a server! TLS private key may be available

Other Memory Disclosure

- Format string vulnerability also leaks memory info
  - “%08x.%08x.%08x…”

- Memory corruption bugs may allow memory leak
  - E.g., overwriting the length field of a string object

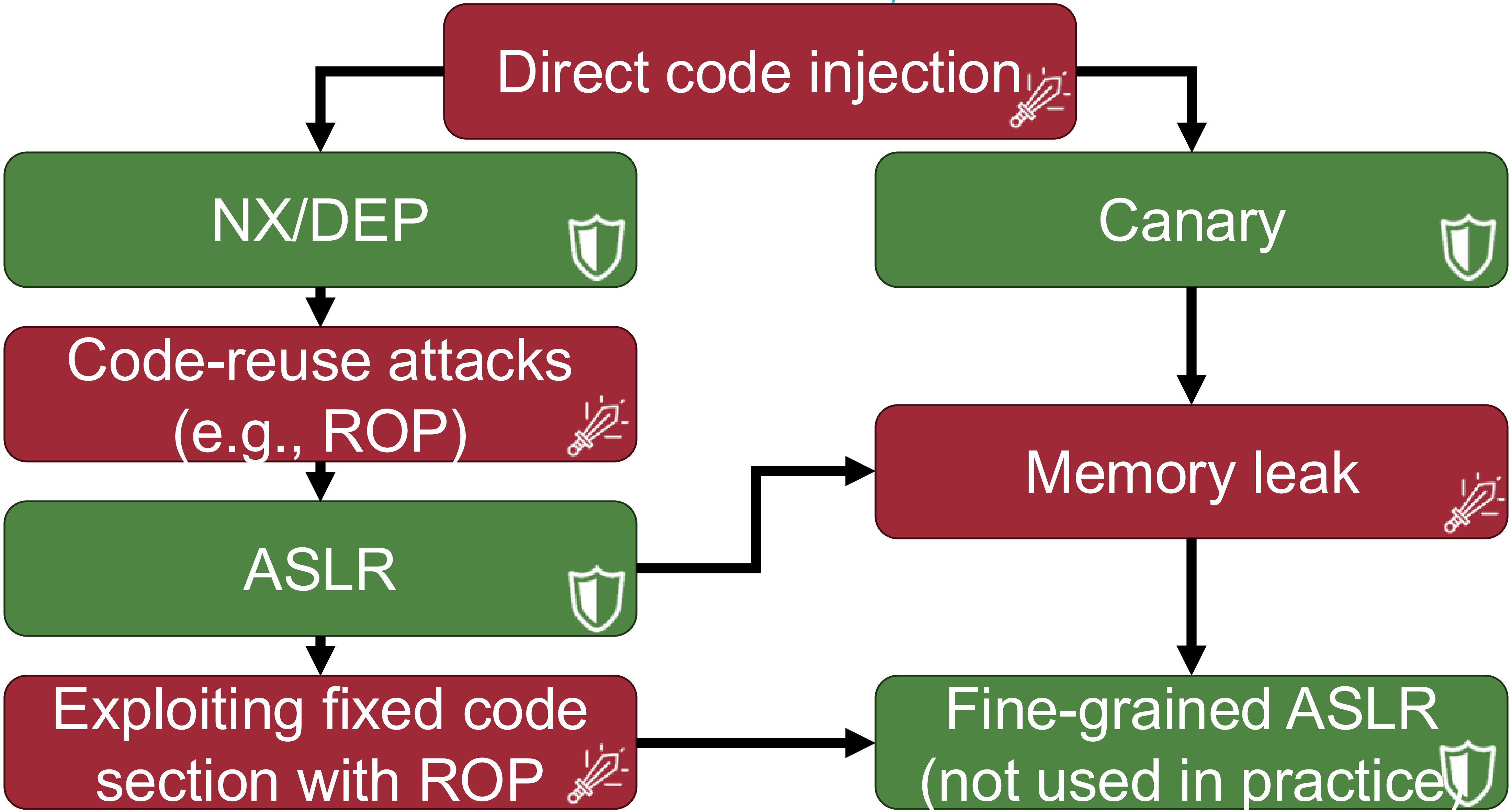
Memory Disclosure and Exploit

- It is possible that a program may have more than a single vulnerability
  - For example, one memory corruption and one memory disclosure

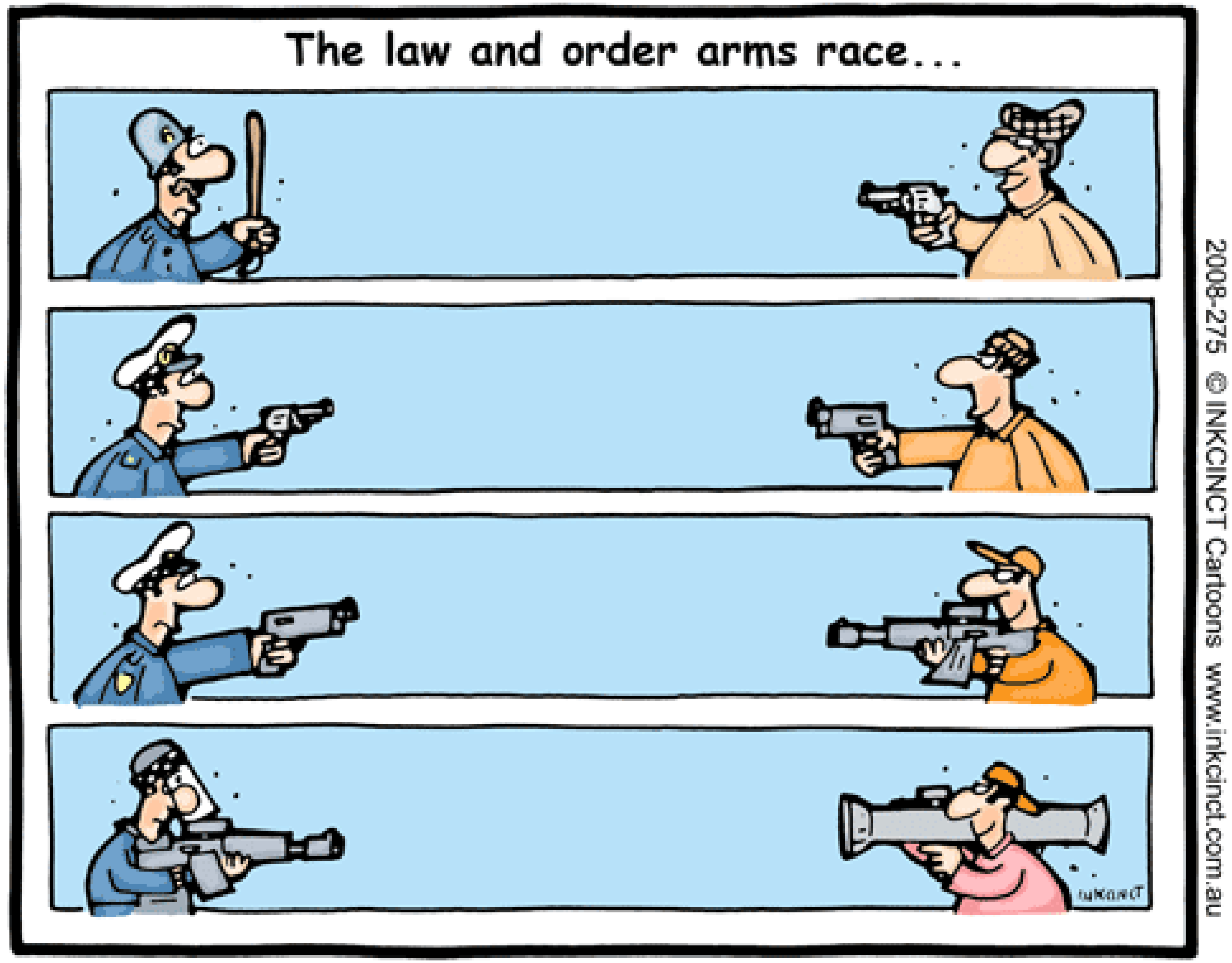
- In such a case, we can bypass existing defenses
  - ***Canary bypass***: canary value could be leaked
  - ***ASLR bypass***: code/stack pointers could be leaked

Caveat: we should be able to leak memory contents and trigger the memory corruption **within the same process**

Attack / Defense So Far



Arms Race in Security



Summary

- Code reuse attacks allow an attacker to bypass DEP
- Many mitigation techniques are proposed for code reuse attacks, which will be covered next

- ASLR: one of the mitigation techniques against code-reuse attacks
  - Brute-forcing attacks and ROP with fixed code section allow an attacker to bypass ASLR

- Memory disclosure ($\neq$ Memory Corruption)

- Security vs. Performance

Question?