CSE467: Computer Security

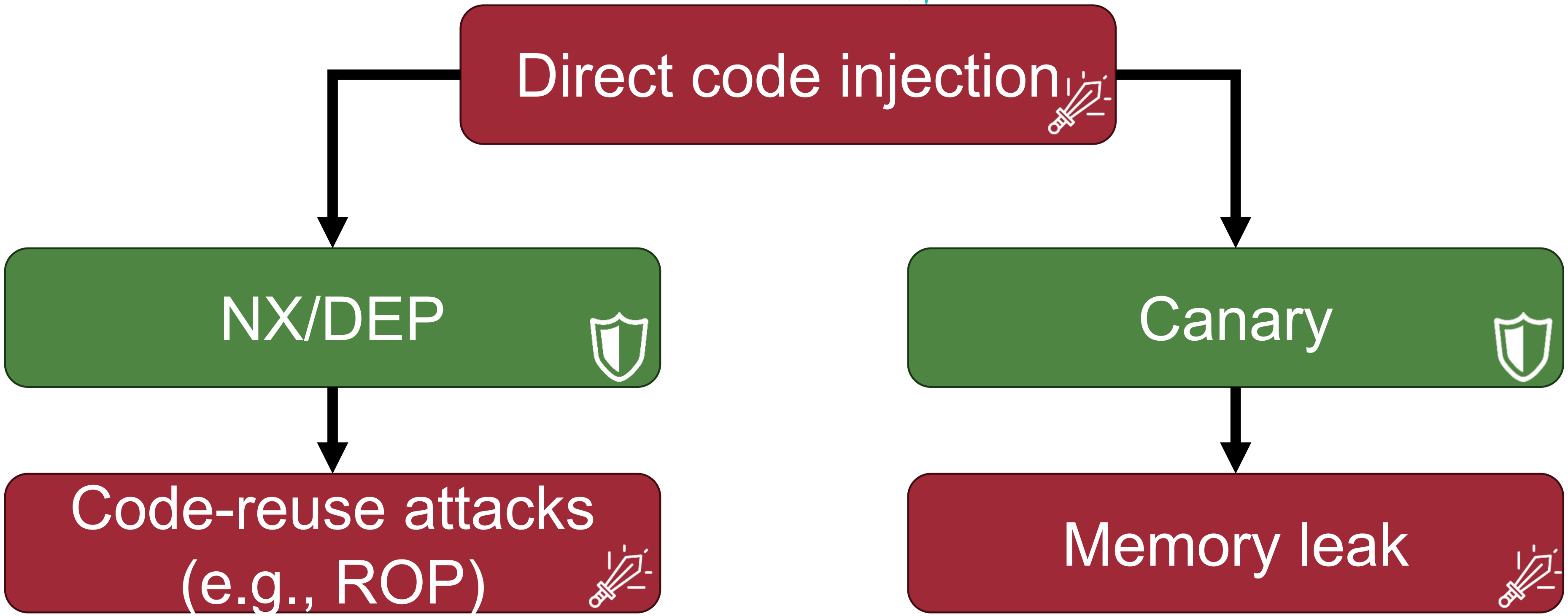
17. ASLR & Memory Disclosure

Seongil Wi

Department of Computer Science and Engineering

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

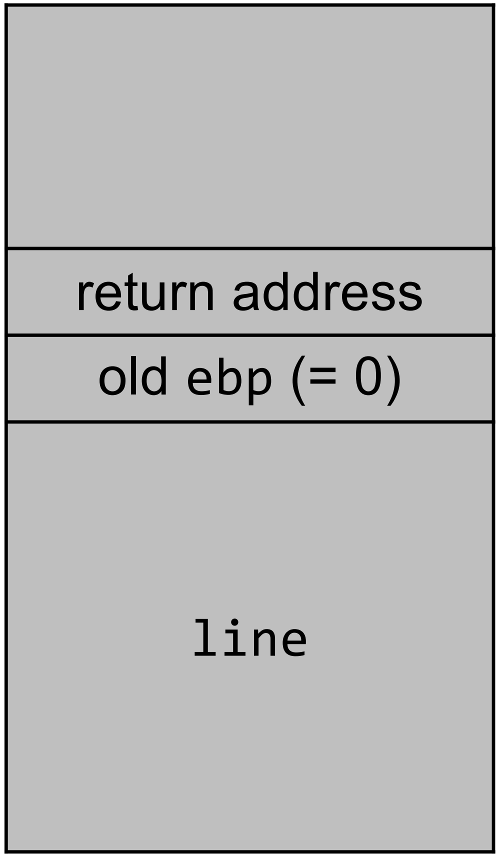
Recap: Control Hijack Attack / Defense So Far



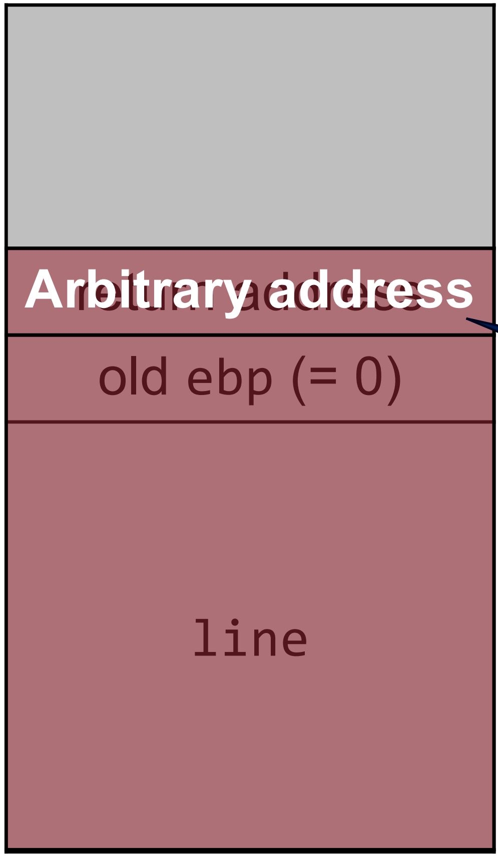
Recap: Bypassing DEP

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

Recap: Jump to Existing Code



Recap: Jump to Existing Code



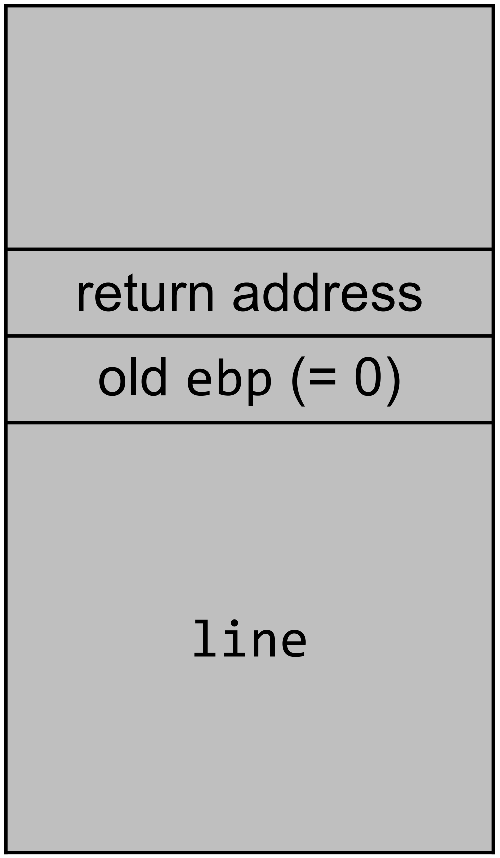
Jump to the ***existing code space***, not to the stack

Recap: Code Reuse Attack #1: Return-to-Libc

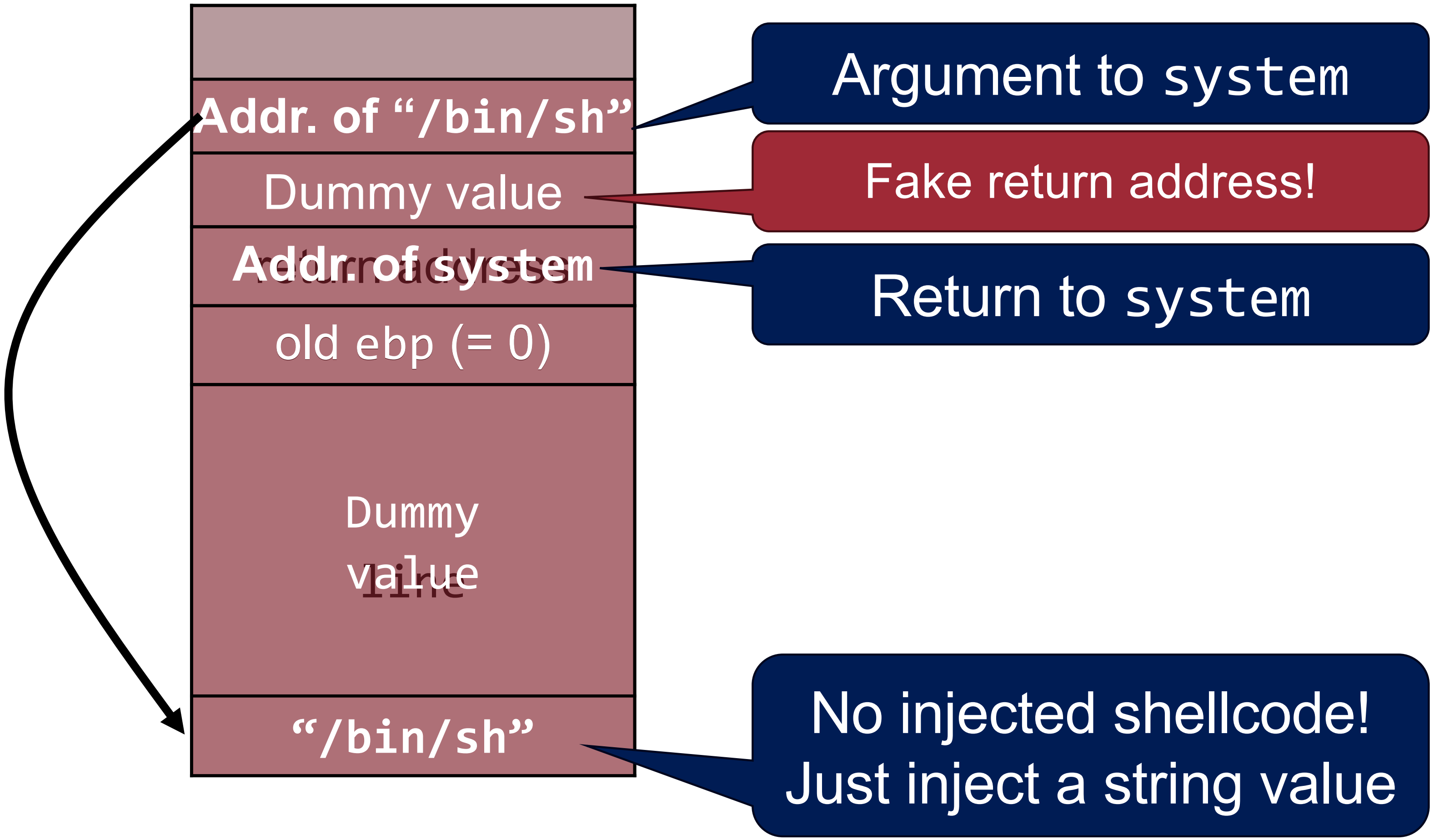
- LIBC (LIBrary C) is a standard library that most programs commonly use
  - For example, `printf` is in LIBC

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

Recap: Code Reuse Attack #1: Return-to-Libc



Recap: Code Reuse Attack #1: Return-to-Libc



Recap: 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
```

$\neq$

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

Recap: 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
```

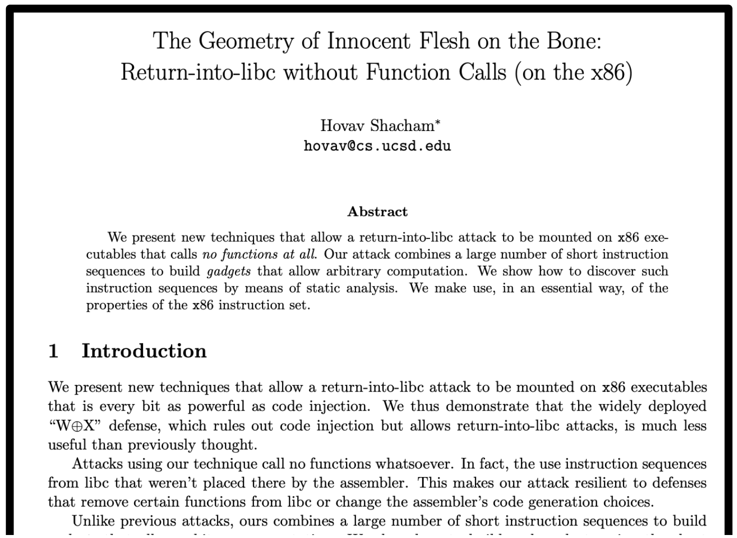
Can we spawn a shell
without the use of LIBC functions?

Recap: 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.

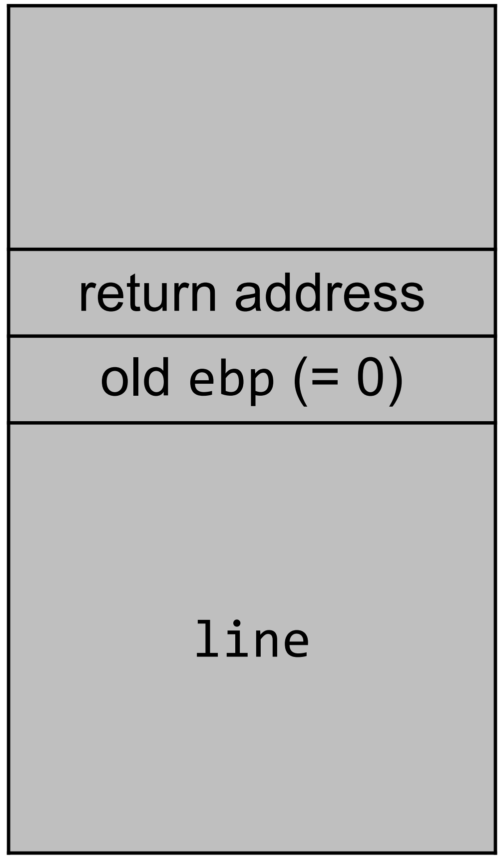
Unlike previous attacks, ours combines a large number of short instruction sequences to build

Recap: Return (ret) Chaining

Attacker's goal:

execute following instructions

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

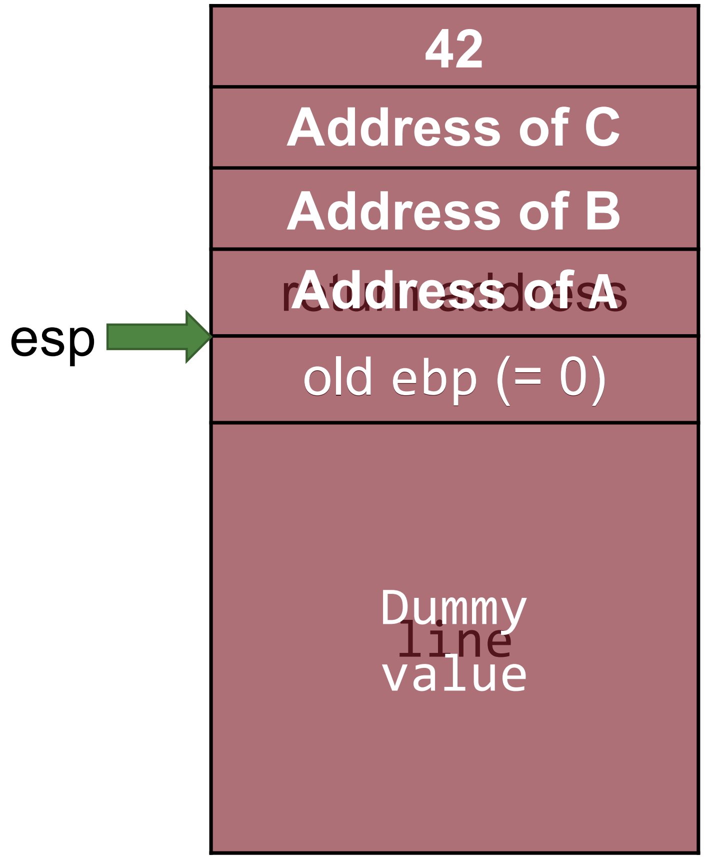


Recap: Return (ret) Chaining

Attacker's goal:

execute following instructions

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



Recap: Return (`ret`) Chaining

Attacker's goal:

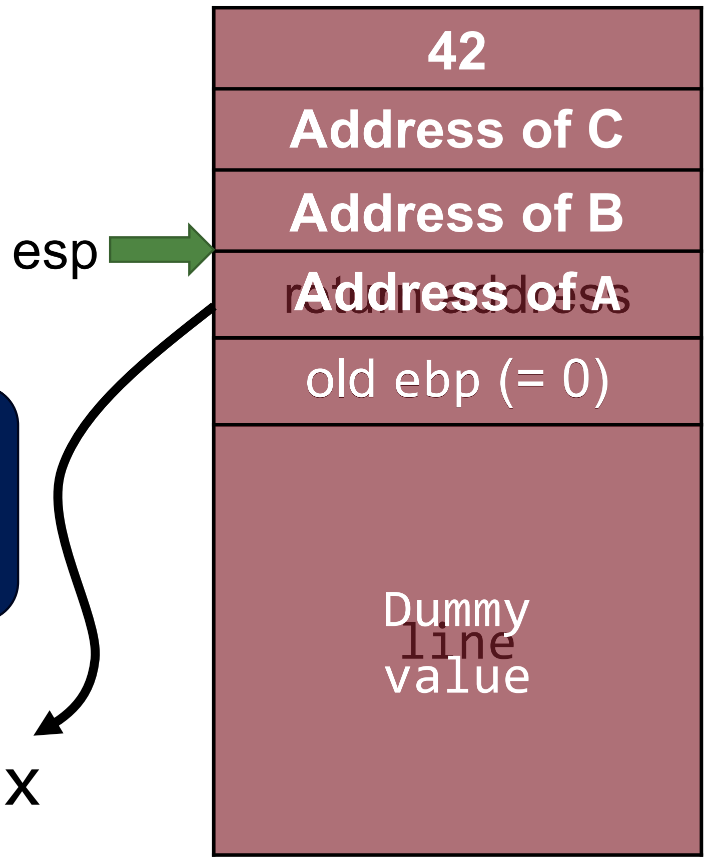
execute following instructions

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

Somewhere in the binary code

A

```
add eax, ebx
ret
```



Recap: Return (ret) Chaining

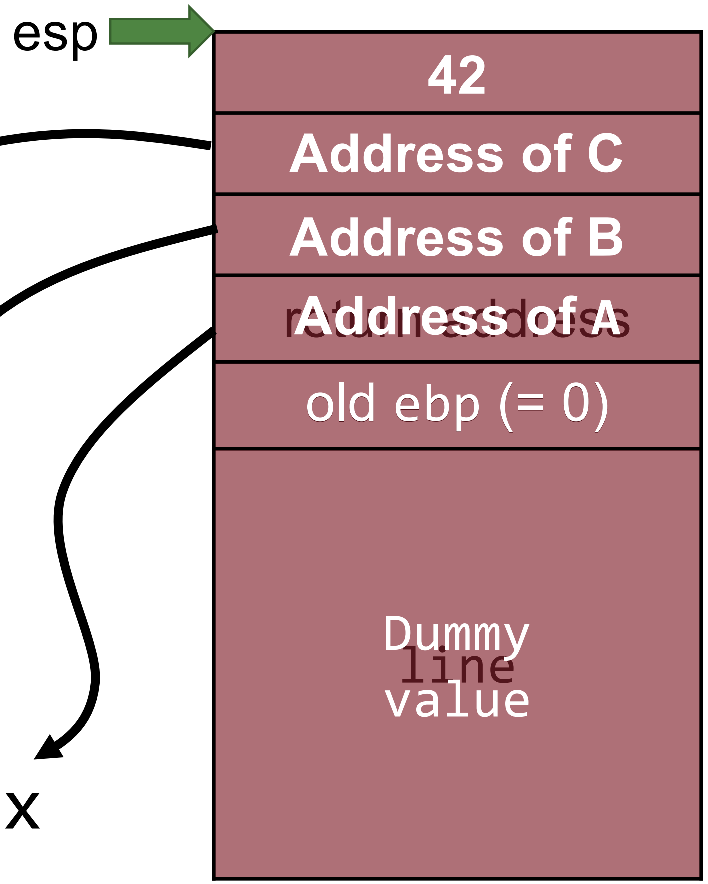
Attacker's goal:
execute following instructions

```
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
```



Recap: Return (ret) Chaining

Attacker's goal:
execute following instructions

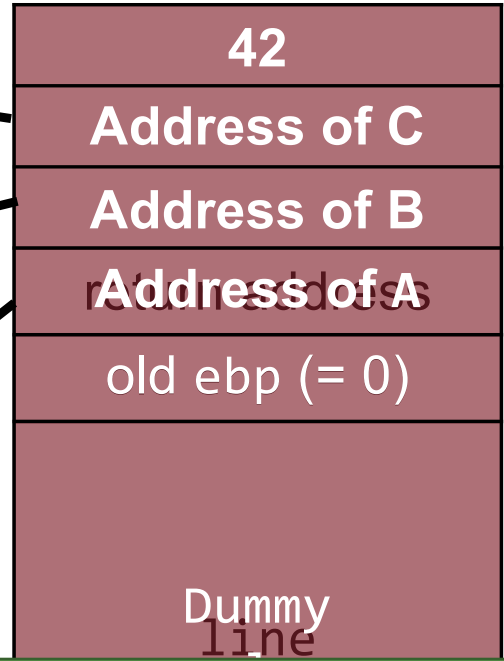
```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C

```
inc ecx
pop edx
ret
```

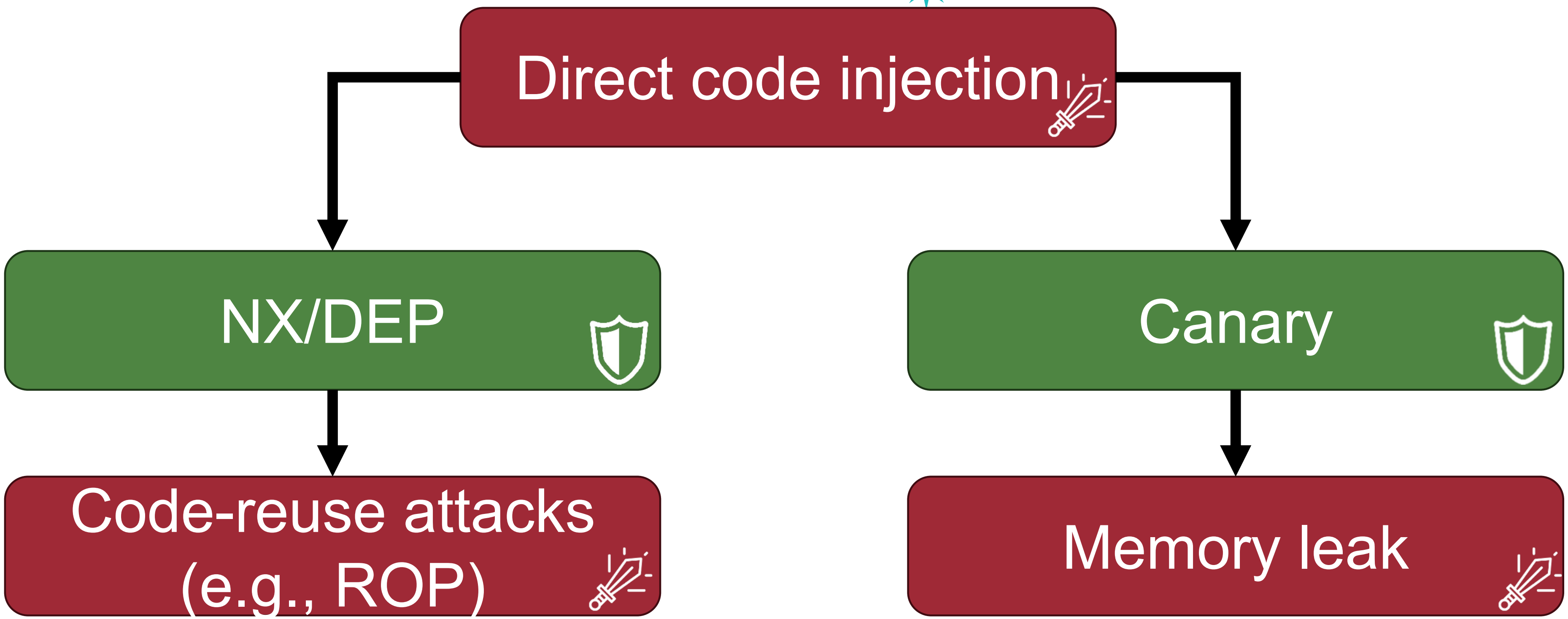
B

```
mov ecx, eax
ret
```



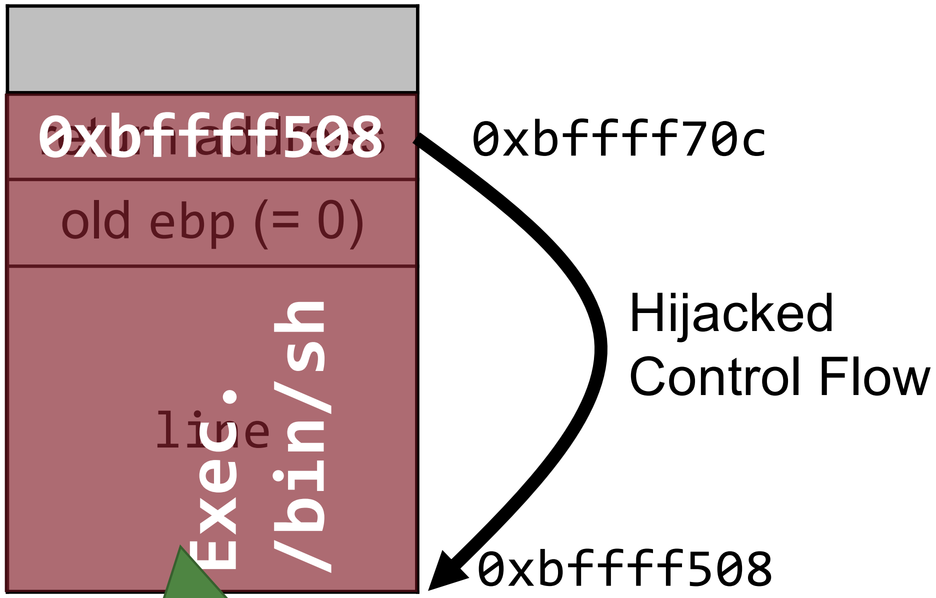
Return chaining with ROP gadgets
allows **arbitrary computation!**

Motivation


Direct code injection
NX/DEP
Canary
Code-reuse attacks
(e.g., ROP)
Memory leak

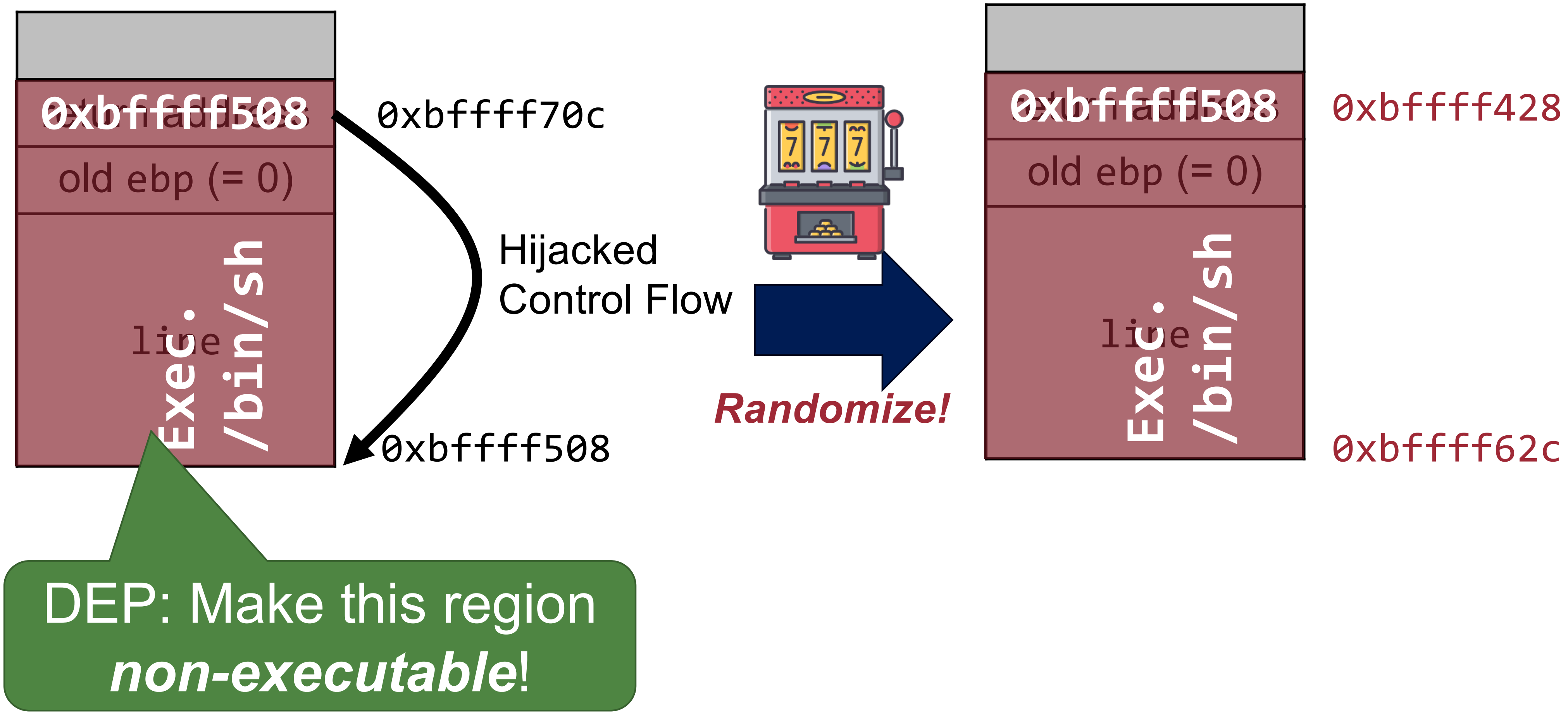
Address Space Layout Randomization (ASLR)

Control Flow Hijack Attack

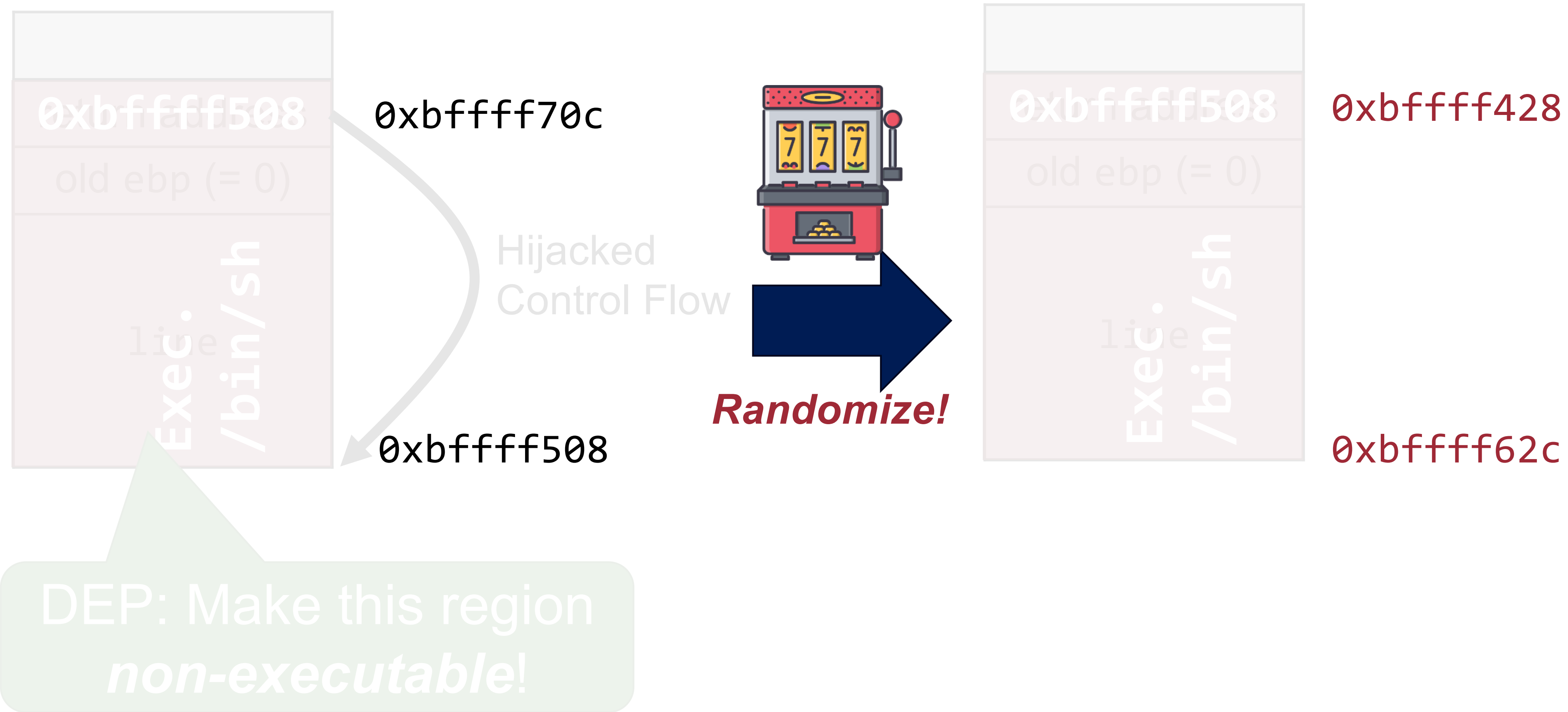


DEP: Make this region ***non-executable***!

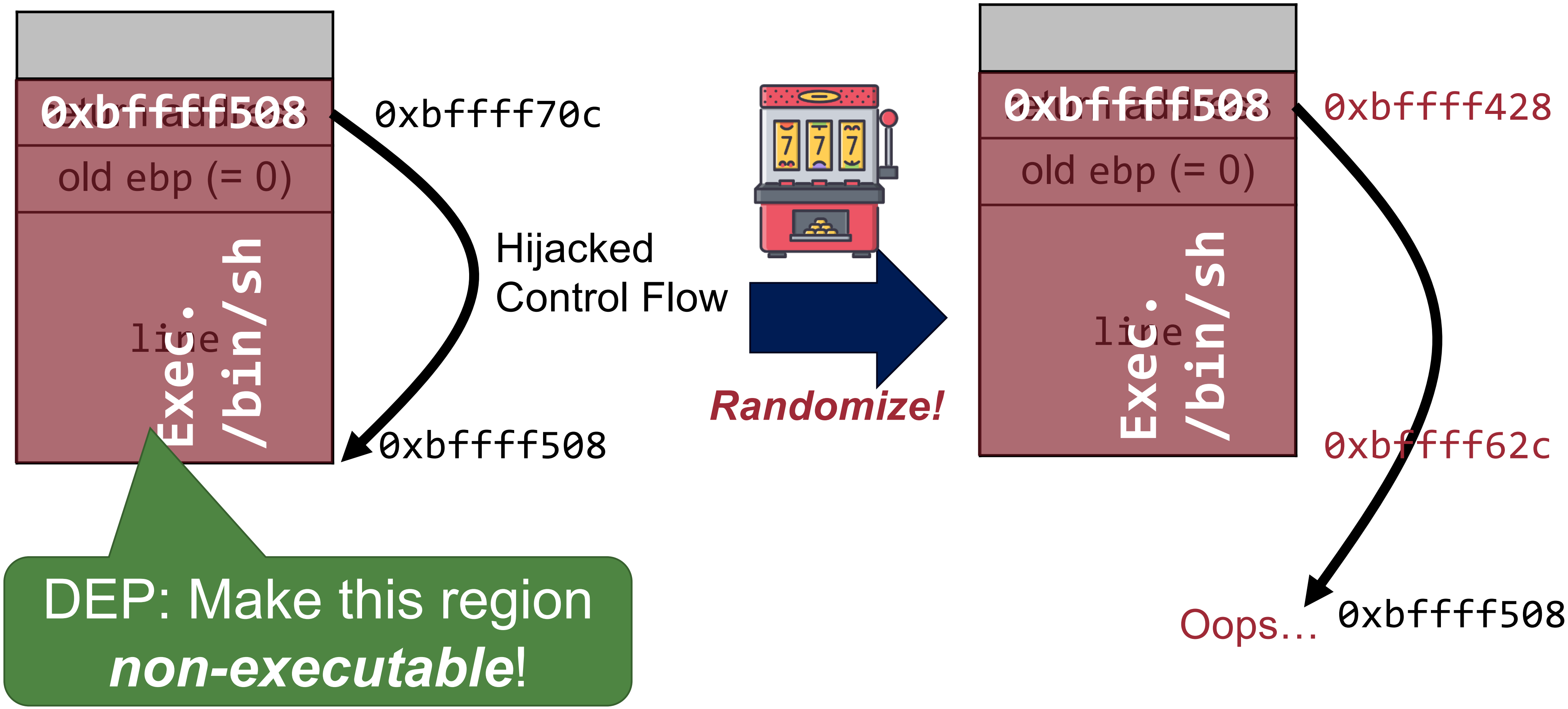
Different Perspective: ASLR



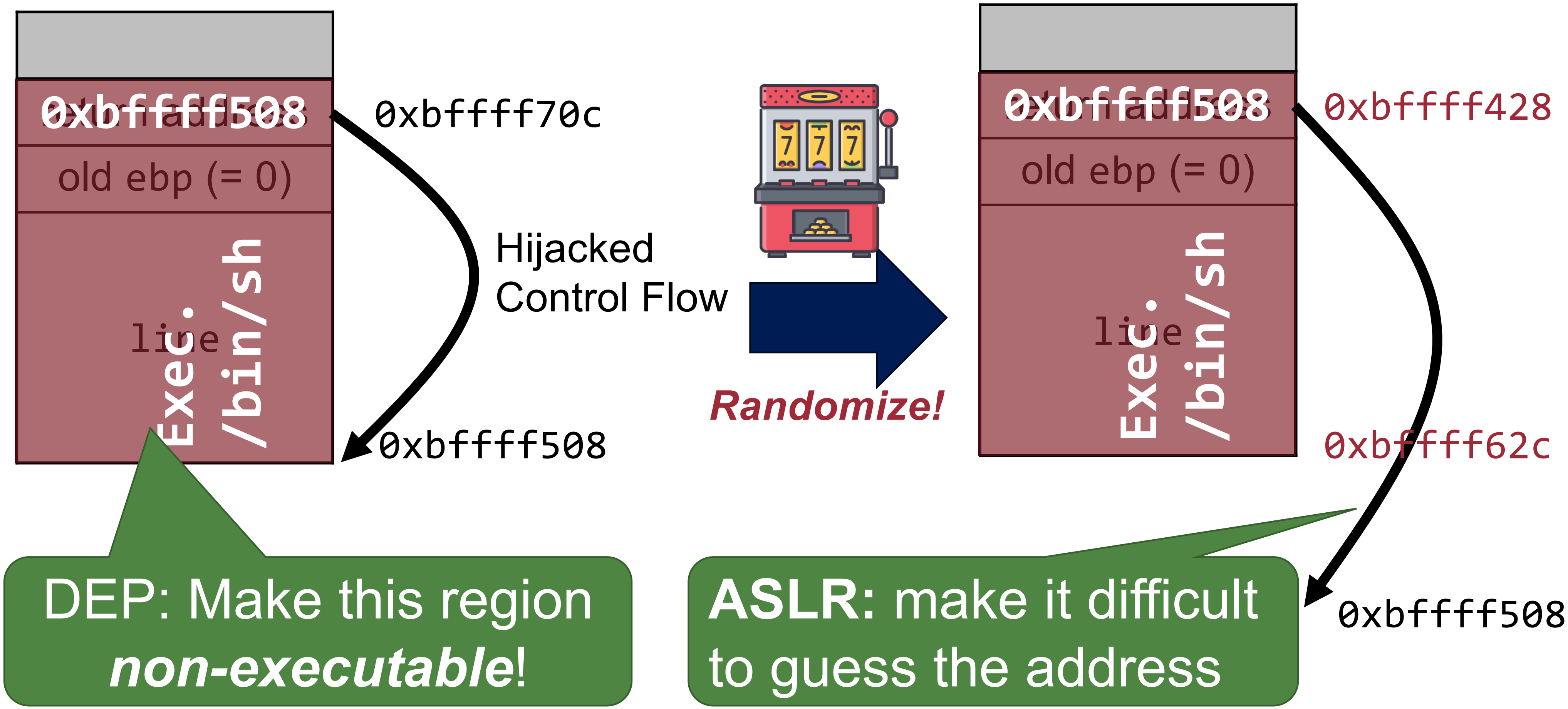
Different Perspective: ASLR



Different Perspective: ASLR



Different Perspective: ASLR



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
```

DEMO

World with ASLR

- ASLR is ON by default [Ubuntu-Security]

You can enable ASLR by:

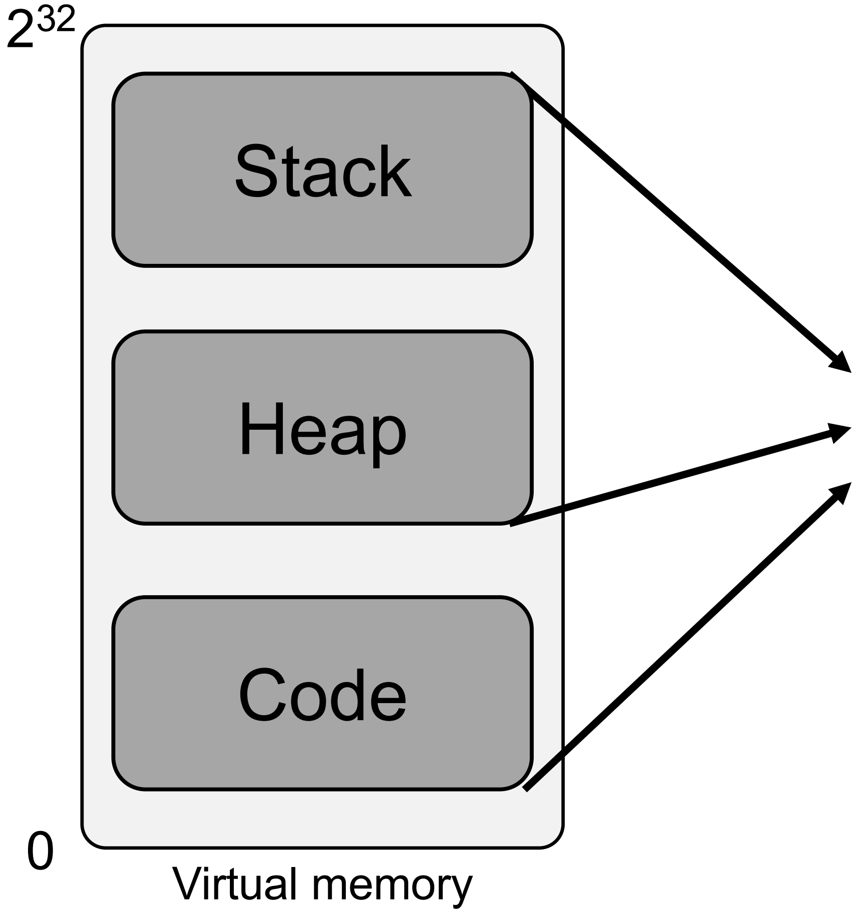
```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

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

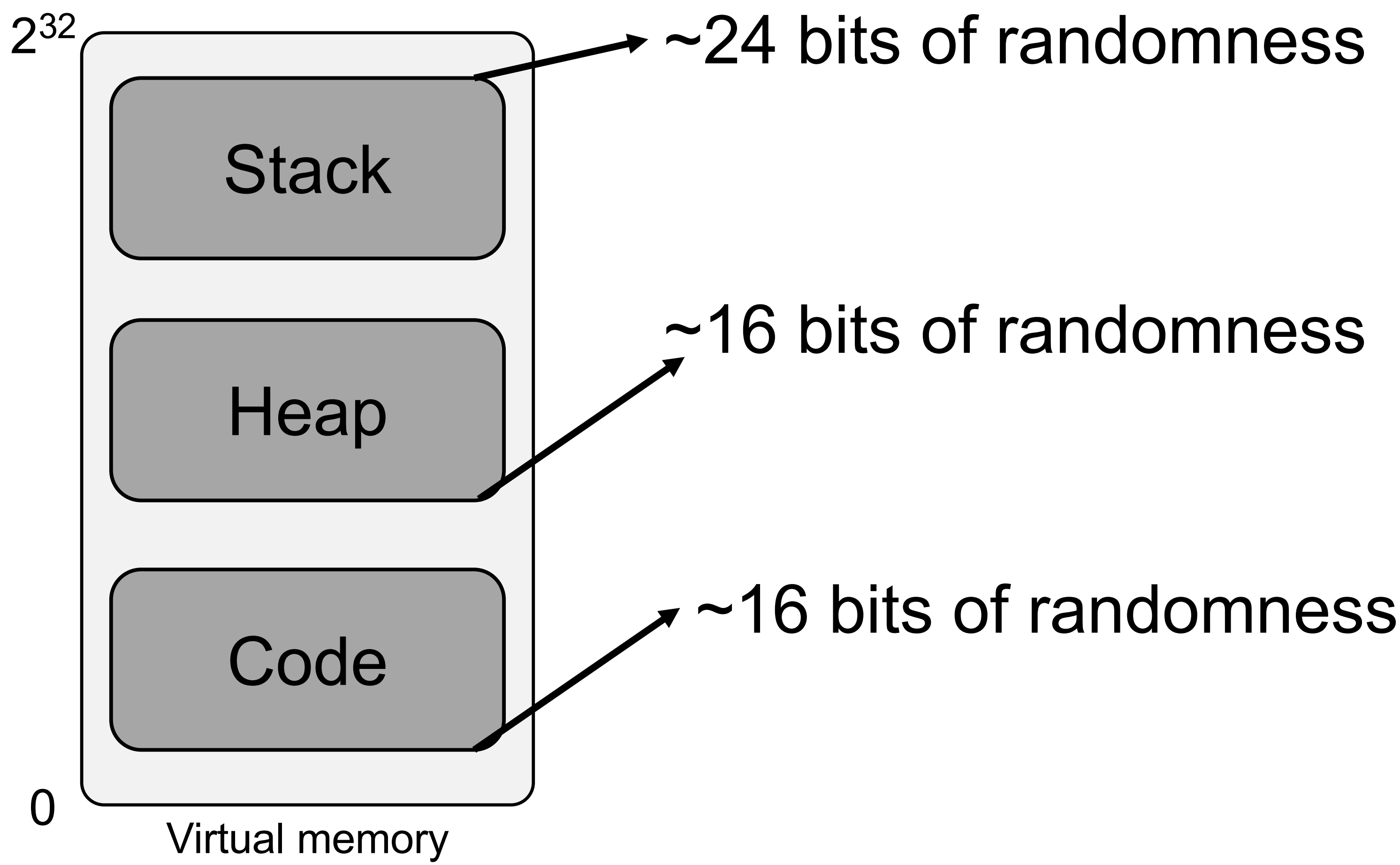
ASLR Randomizes Virtual Memory Areas



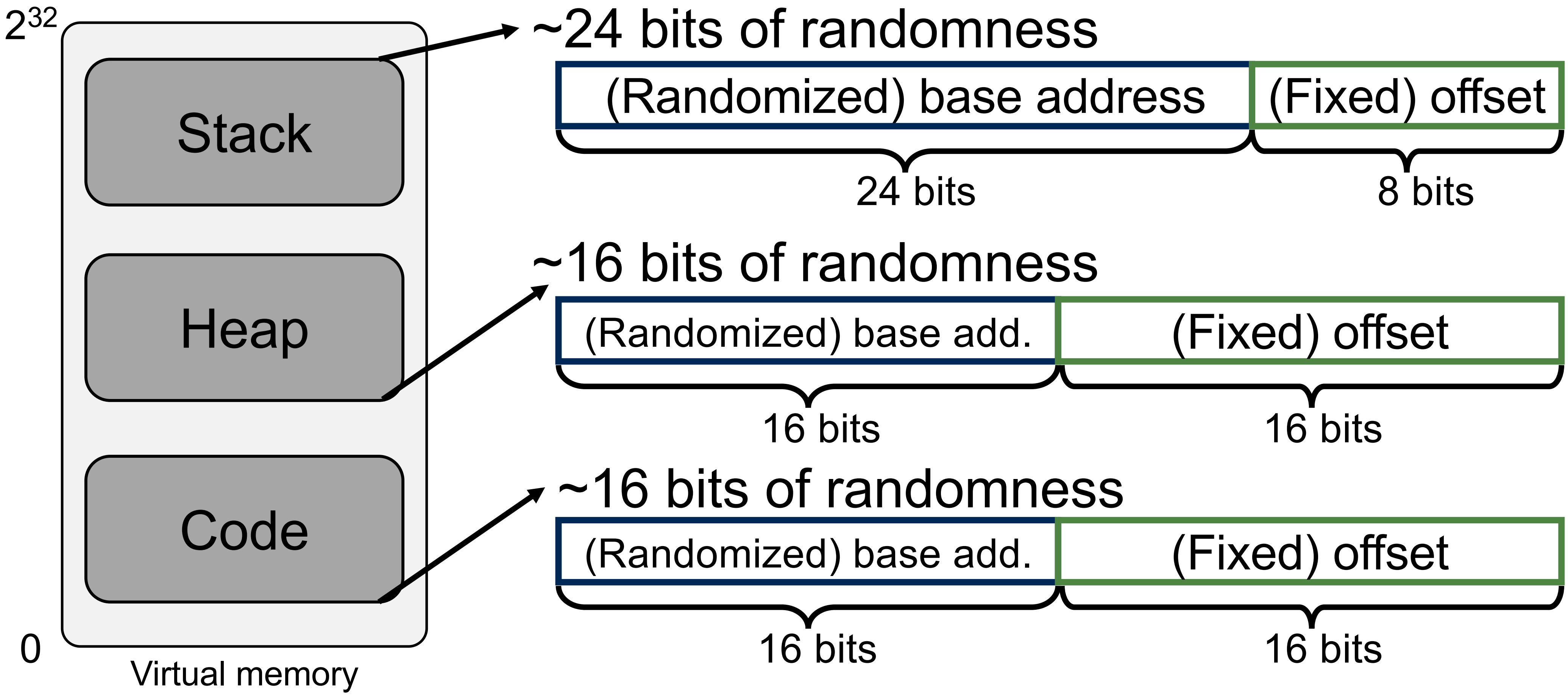
Randomized **based addresses**

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

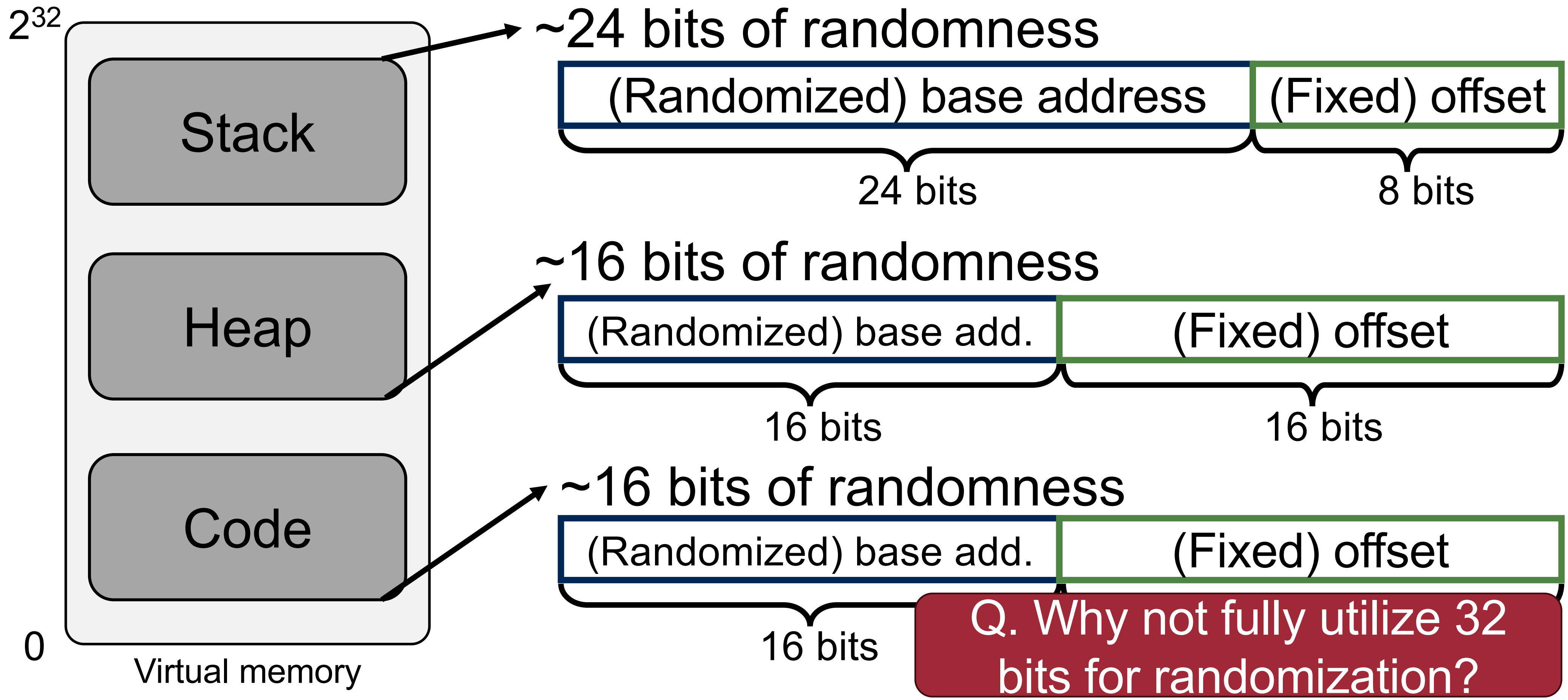
ASLR Randomizes Virtual Memory Areas



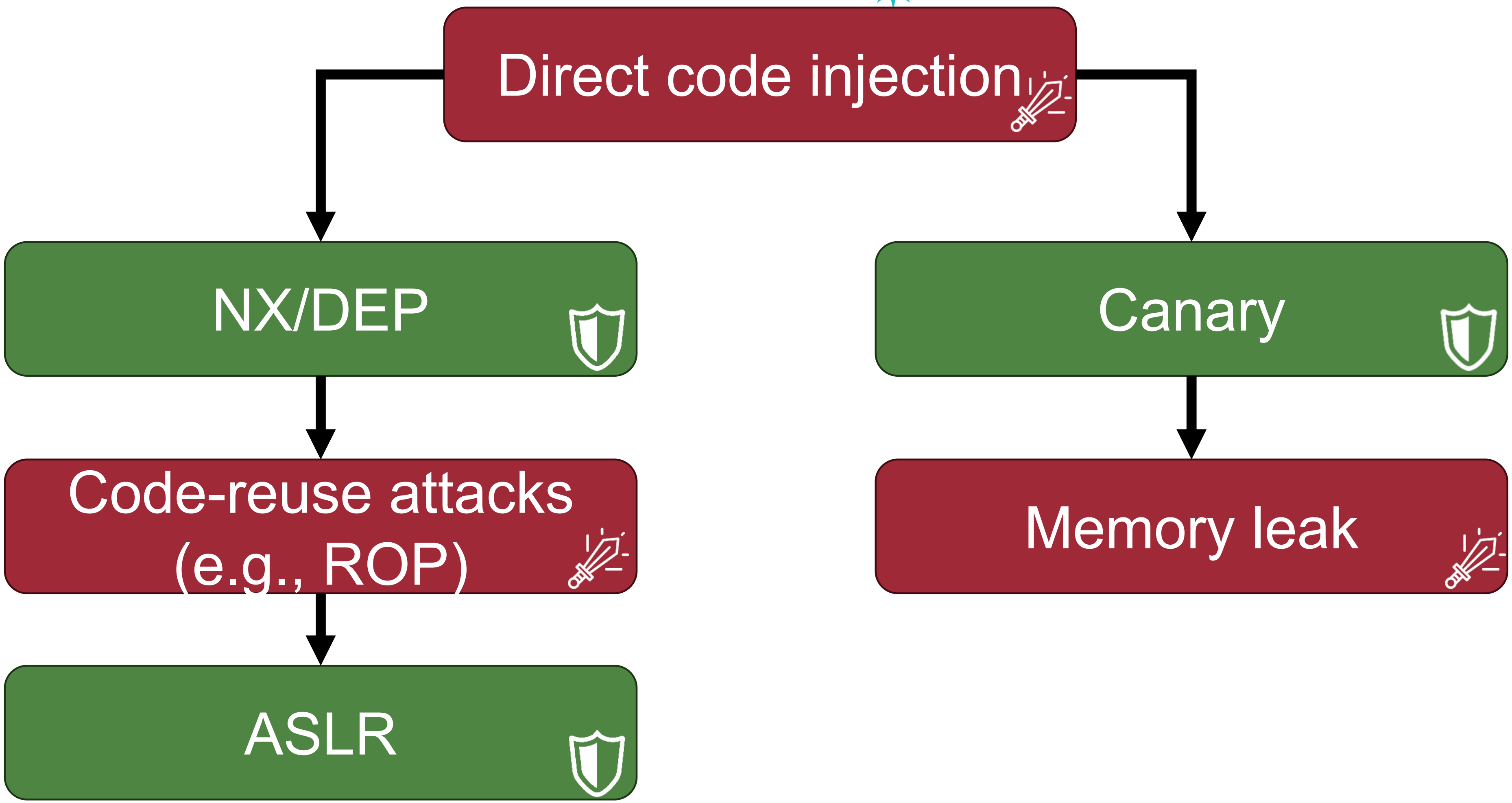
ASLR Randomizes Virtual Memory Areas



ASLR Randomizes Virtual Memory Areas



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

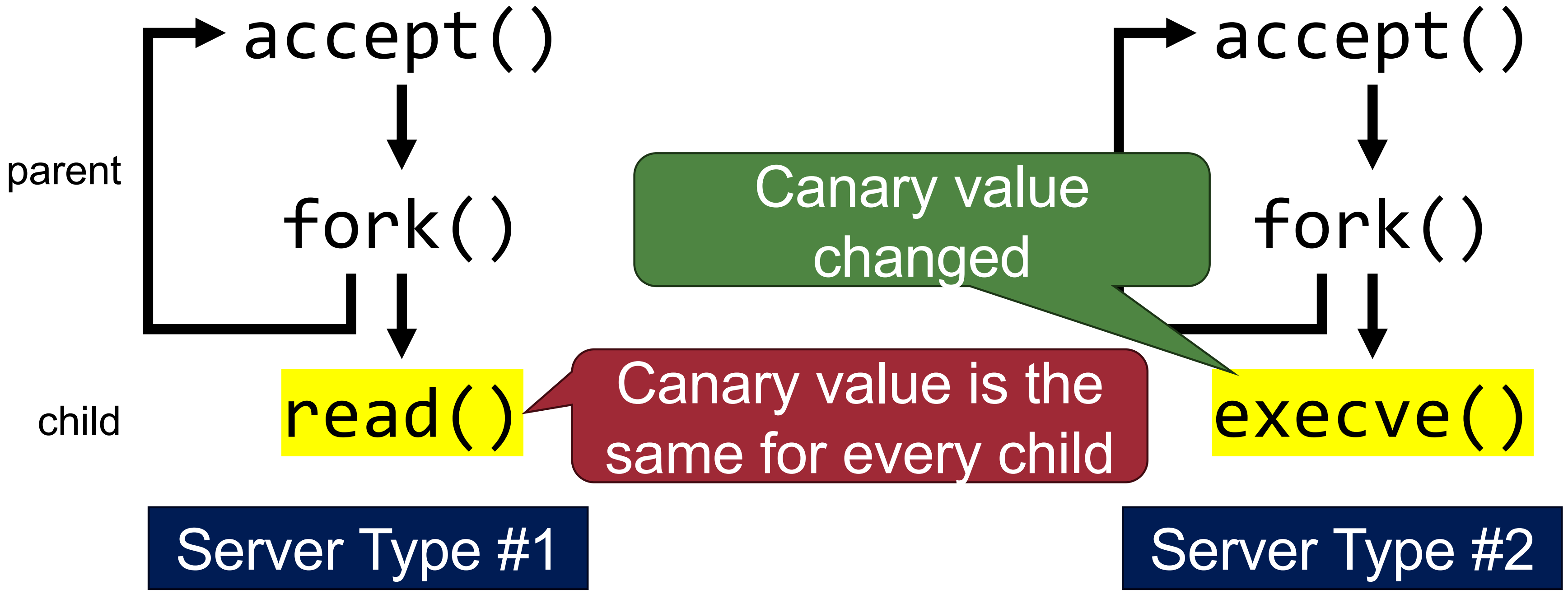
Attack #1: Entropy is Small on x86

- 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

Recap: Reused Canary Value

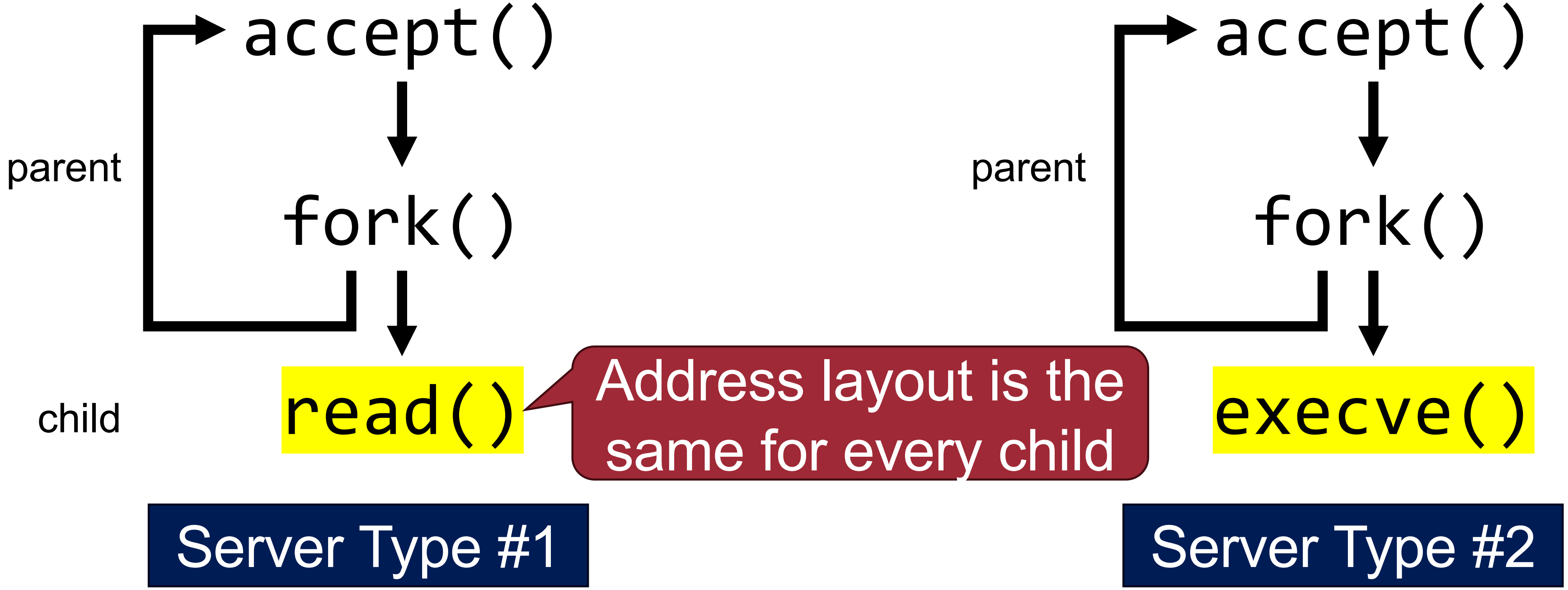
- Uses a random canary value for ***every process creation***



e.g., OpenSSH does this

Remained Address Space

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



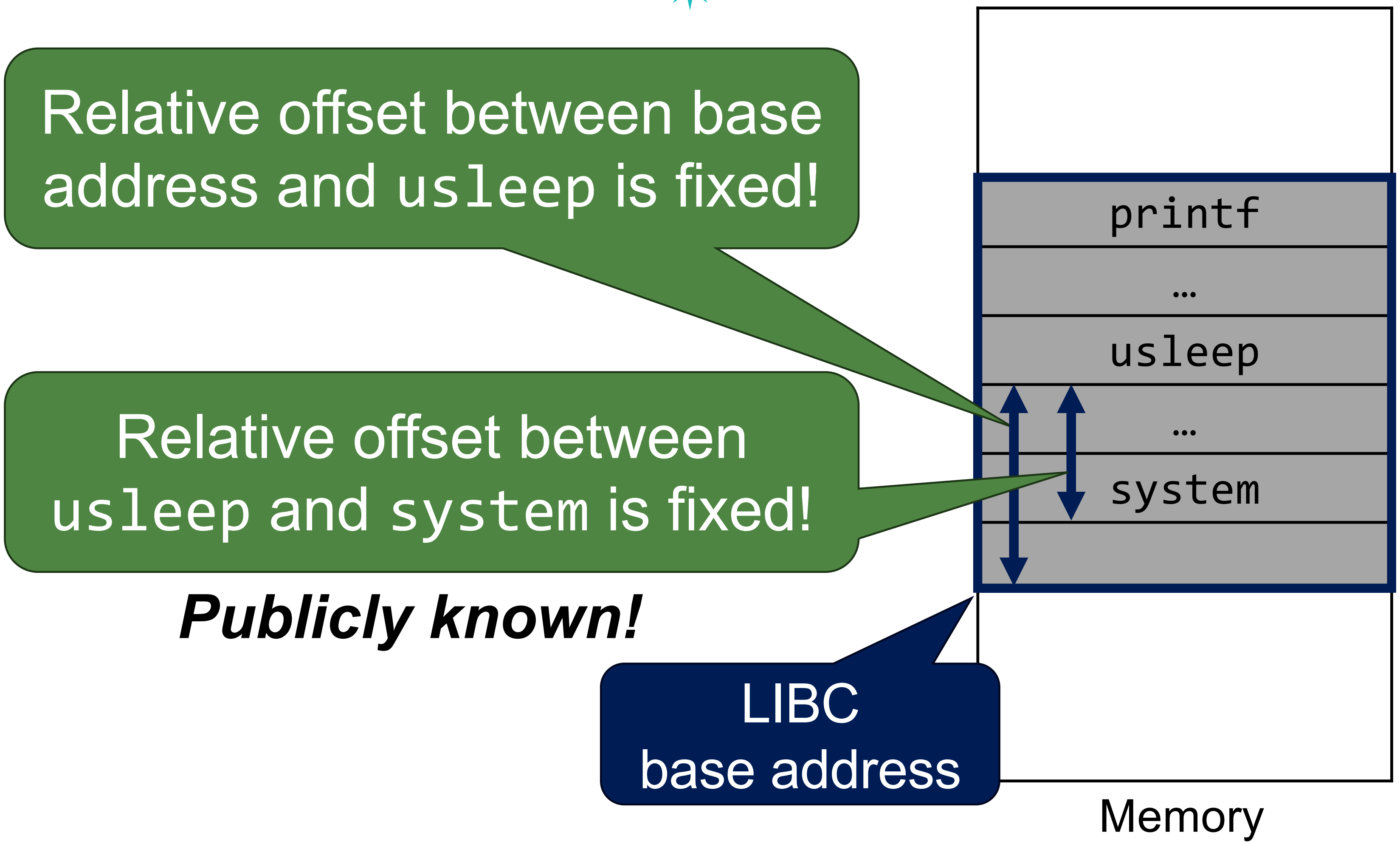
Attack #1: Entropy is Small on x86

- 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***!

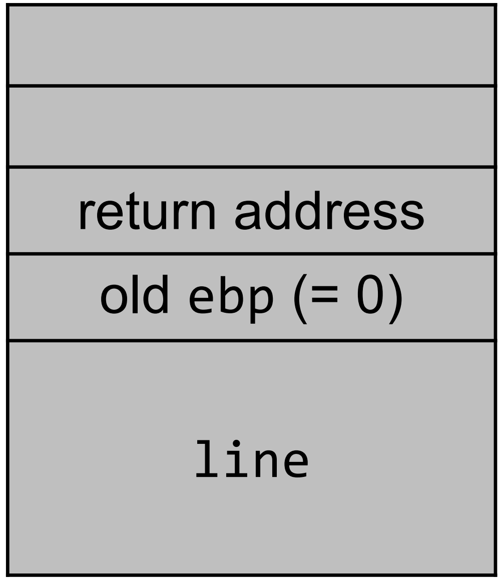
Key point: relative offsets between LIBC functions are the same regardless of ASLR

Observation: Relative Offsets are Same!



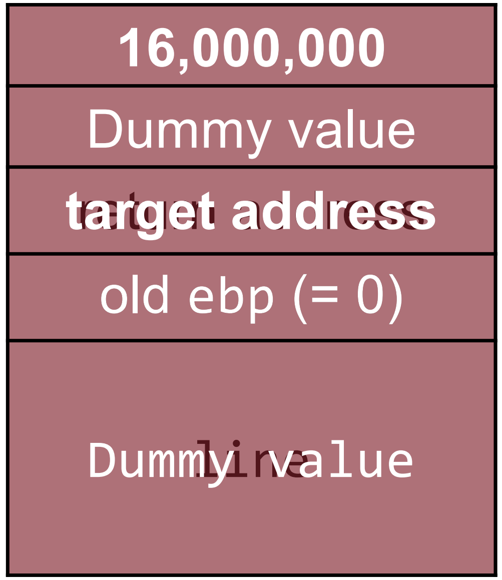
Brute-forcing Attack Example

- 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

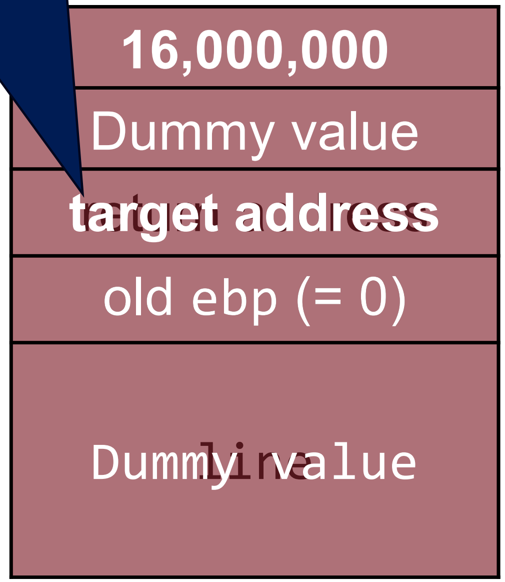


Brute-forcing Attack Example

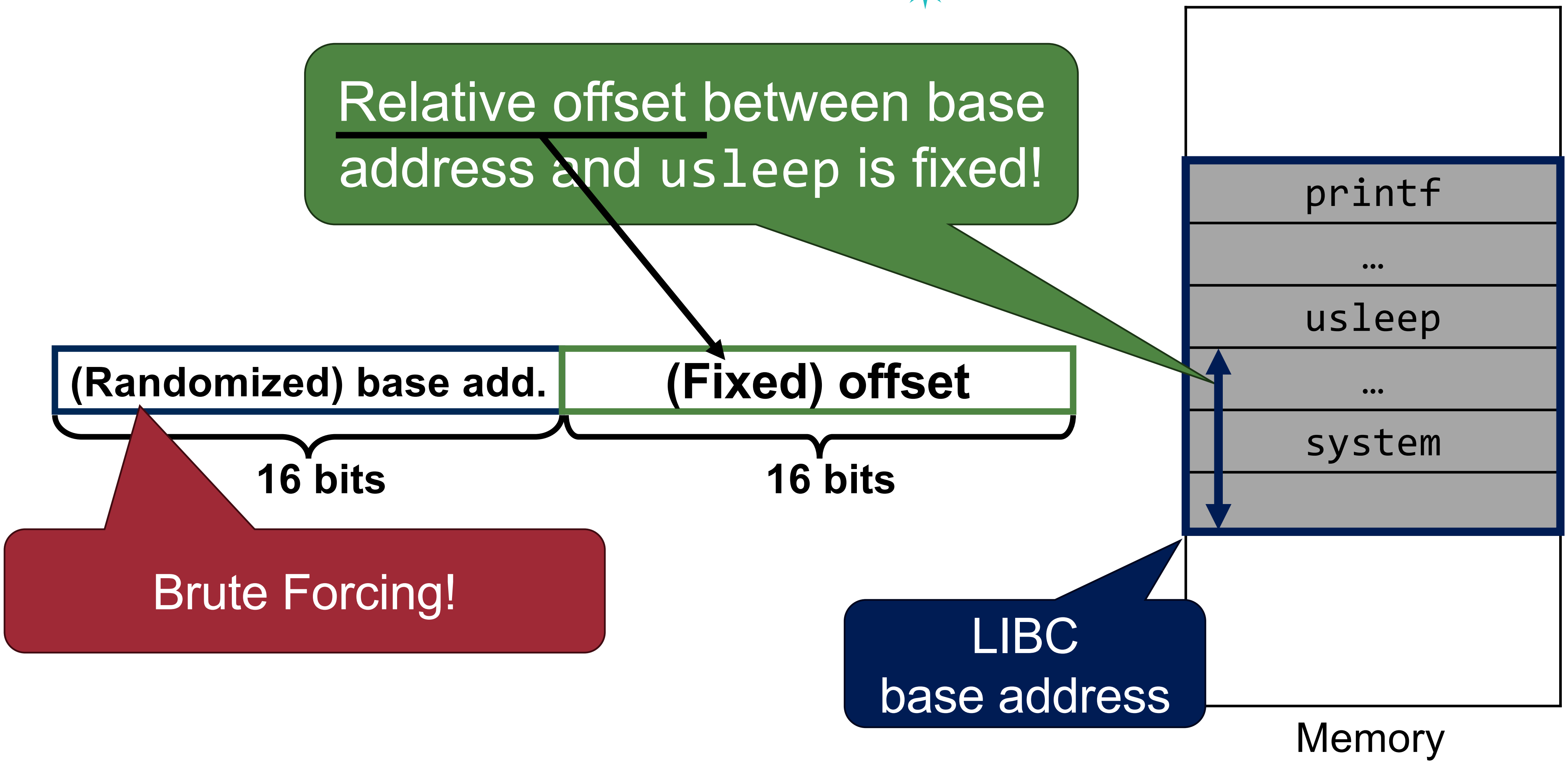
- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow

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)



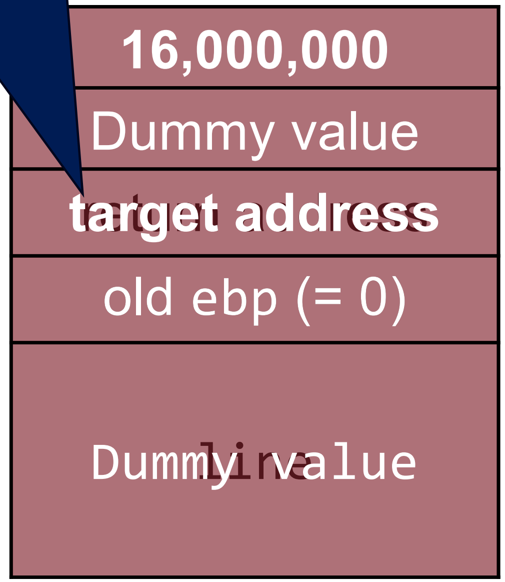
Observation: Relative Offsets are Same!



Brute-forcing Attack Example

- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow
- 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-force on 16 bits to find the address of `usleep`

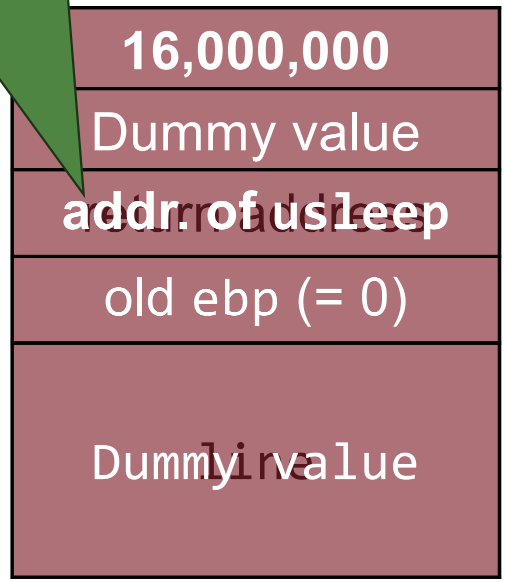


Brute-forcing Attack Example

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

If correct, the server will wait 16 seconds

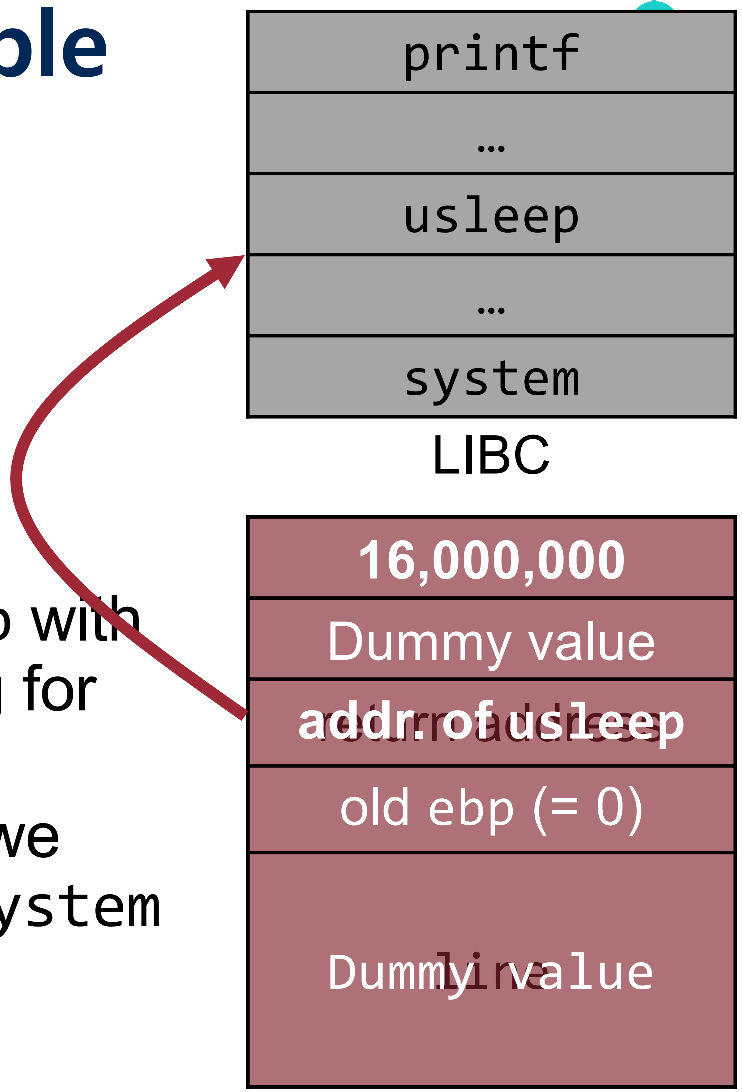
- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)



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`

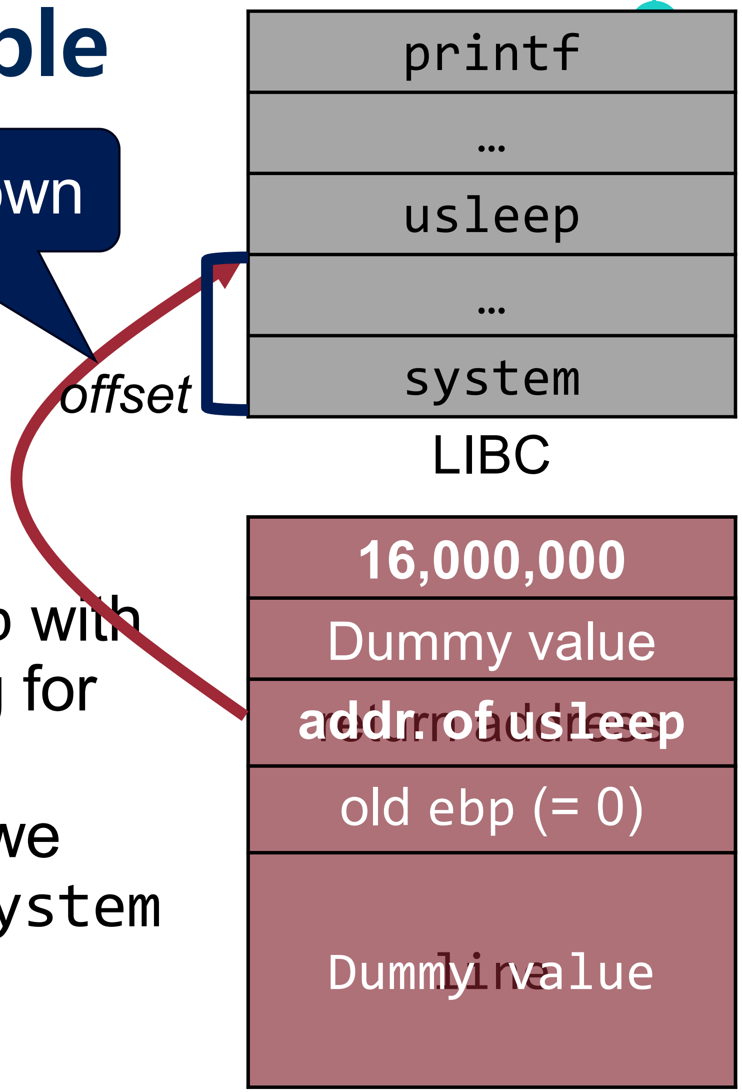


Brute-forcing Attack Example

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

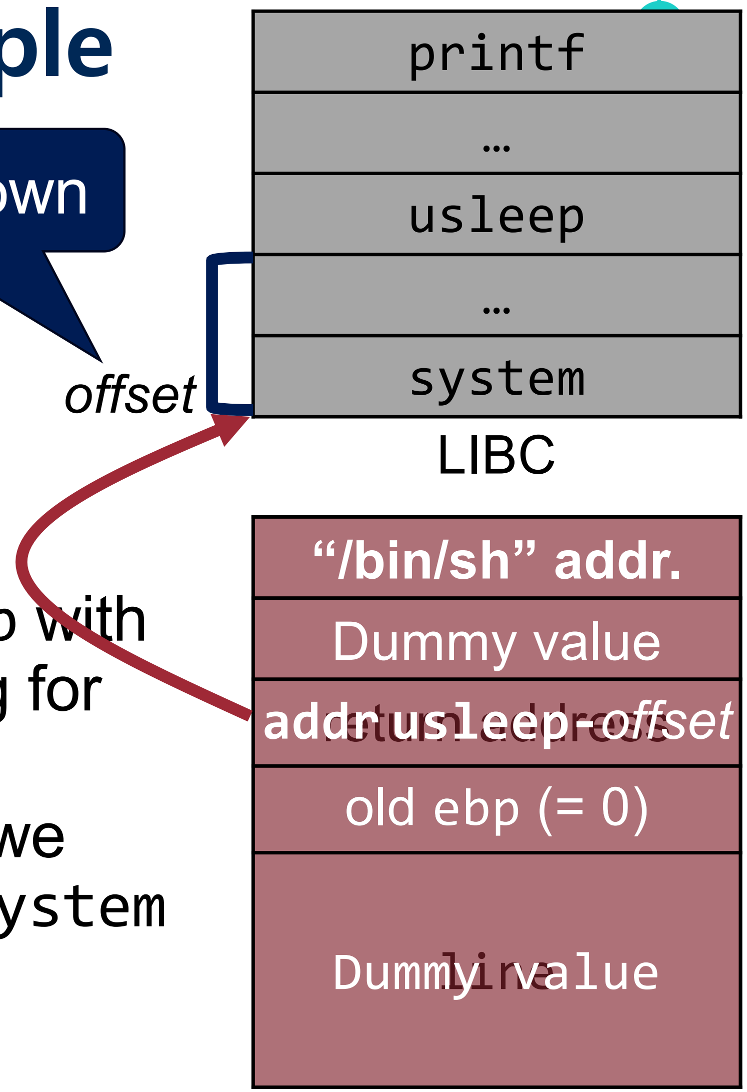
Publicly known

- 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`



Randomization Frequency on Two Major OSes

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

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

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?

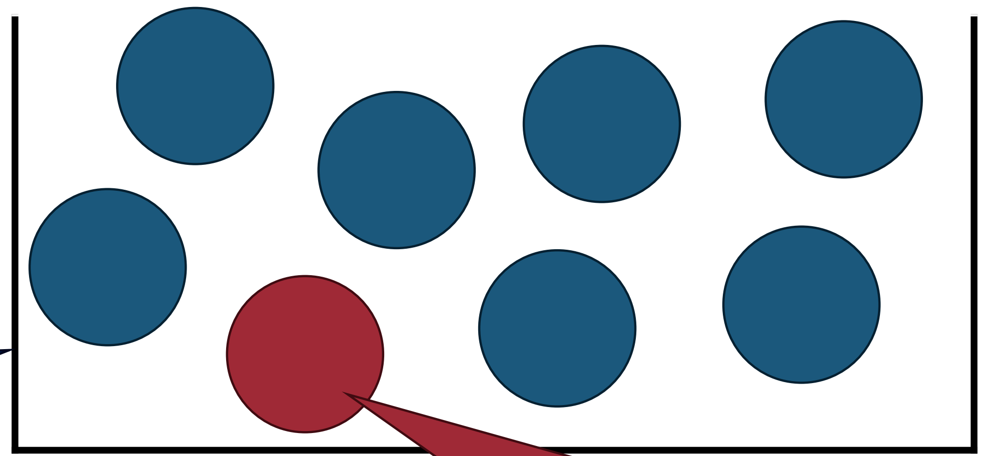
Security: Which One is Better?

- What is the <u>expected number of trials</u> to correctly guess the base address for each case?
  - Case #1: no randomization for each execution (***Windows***)
  - Case #2: re-randomization for each execution (***Linux***)

$2^N - 1$ Blue Balls and 1 Red Ball in a Jar

We have **2^N balls** in a jar

- ***N***: # of randomized bits



There a total of 2^N possible base addresses

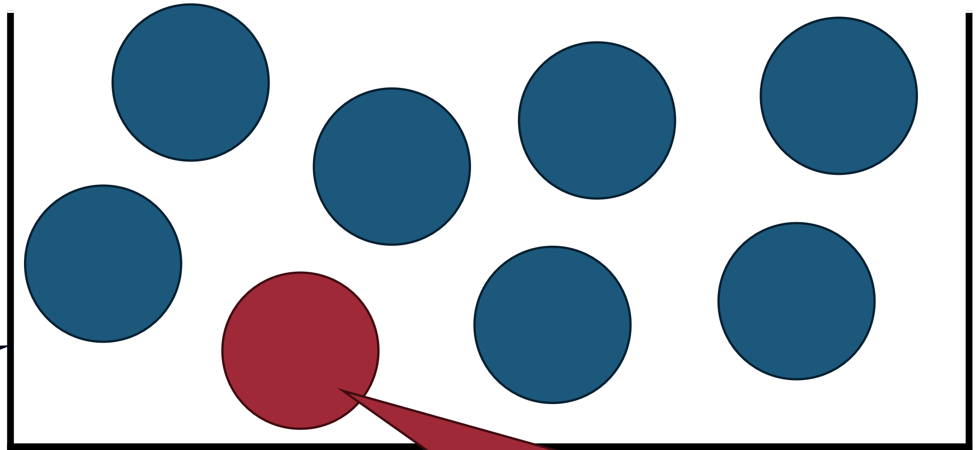
One **red ball** in a jar, which corresponds to the expected base address.

What is the probability of selecting the ***red ball***?

$2^N - 1$ Blue Balls and 1 Red Ball in a Jar

We have **2^N balls** in a jar
- ***N***: # of randomized bits

There a total of 2^N possible base addresses

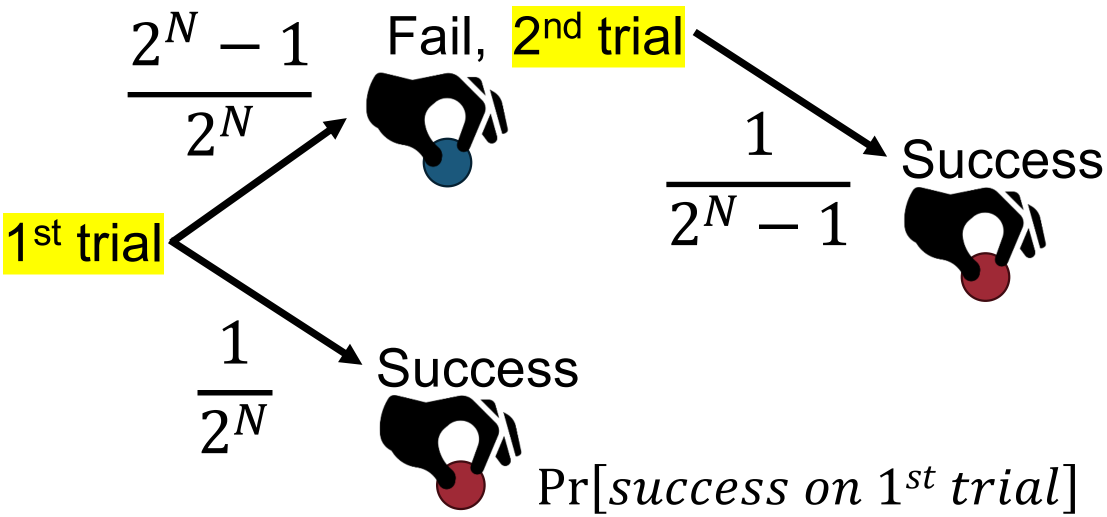


One red ball in a jar, which corresponds to the expected base address.

What is the probability of selecting the ***red ball***?

- Case 1: Select balls without replacement (***Windows***)
- Case 2: Select balls with replacement (***Linux***)

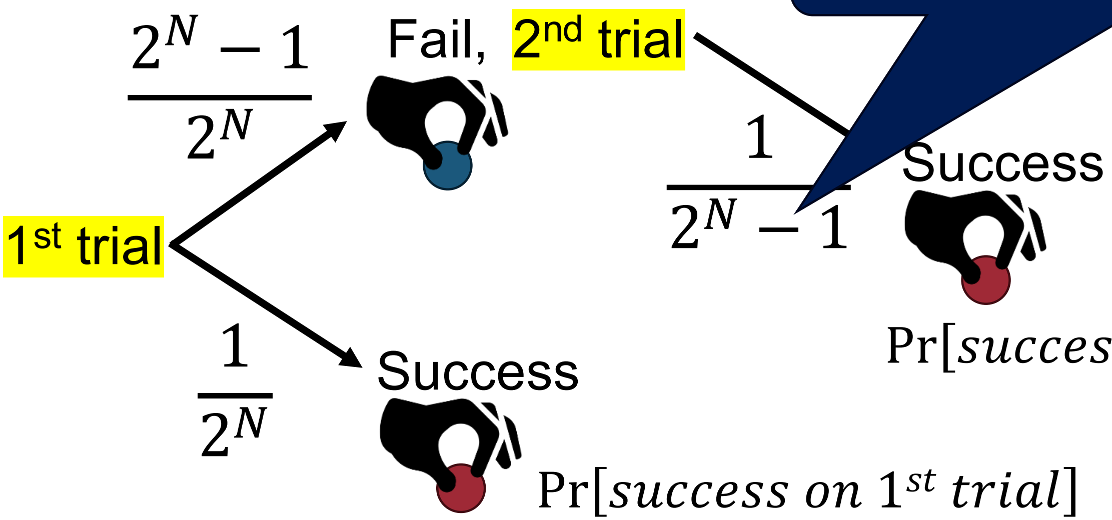
Case #1: Selecting Balls w/o Replacement
(Windows)



$$\Pr[success\ on\ 1^{st}\ trial] = \frac{1}{2^N}$$

Case #1: Selecting Balls w/o Replacement
(Windows)

Without replacement!

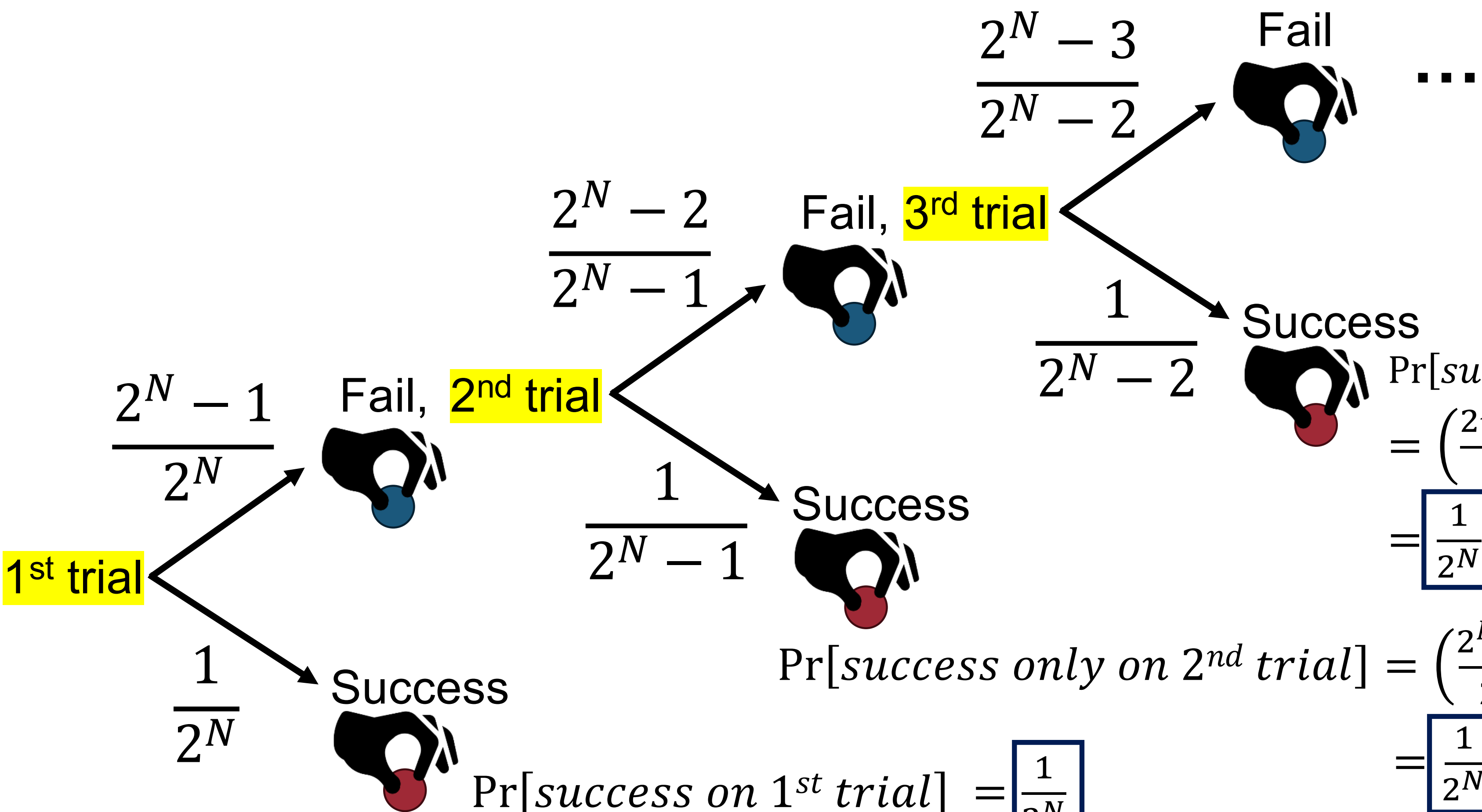


$$\Pr[success\ on\ 1^{st}\ trial] = \frac{1}{2^N}$$

$$\Pr[success\ only\ on\ 2^{nd}\ trial] = \left(\frac{2^N-1}{2^N}\right)\left(\frac{1}{2^N-1}\right) = \frac{1}{2^N}$$

Case #1: Selecting Balls w/o Replacement

(Windows)



$$Pr[success\ on\ 1^{st}\ trial] = \boxed{\frac{1}{2^N}}$$

$$Pr[success\ only\ on\ 2^{nd}\ trial] = \left(\frac{2^N-1}{2^N}\right)\left(\frac{1}{2^N-1}\right) = \boxed{\frac{1}{2^N}}$$

$$Pr[success\ only\ on\ 3^{rd}\ trial] = \left(\frac{2^N-1}{2^N}\right)\left(\frac{2^N-2}{2^N-1}\right)\left(\frac{1}{2^N-2}\right) = \boxed{\frac{1}{2^N}}$$

Case #1: Selecting Balls w/o Replacement

(Windows)

$$\Pr[success\ only\ on\ k^{th}\ trial] = \left(\frac{2^N-1}{2^N}\right) \times \cdots \times \left(\frac{2^N-k+1}{2^N-k}\right) \times \left(\frac{1}{2^N-k+1}\right) = \frac{1}{2^N}$$

- **Expected # of trials before success**

$$E[X] = \sum_{k=1}^{2^N} k \cdot Pr[\text{success only on } k\text{th trial} = \sum_{k=1}^{2^N} \frac{k}{2^N}$$

Case #1: Selecting Balls w/o Replacement

(Windows)

$$\Pr[\mathit{success\ only\ on\ k^{th}\ trial}] = \left(\frac{2^N-1}{2^N}\right) \times \cdots \times \left(\frac{2^N-k+1}{2^N-k}\right) \times \left(\frac{1}{2^N-k+1}\right) = \frac{1}{2^N}$$

- **Expected # of trials before success**

$$E[X] = \sum_{k=1}^{2^N} k \cdot Pr[\text{success only on } k\text{th trial} = \sum_{k=1}^{2^N} \frac{k}{2^N}$$

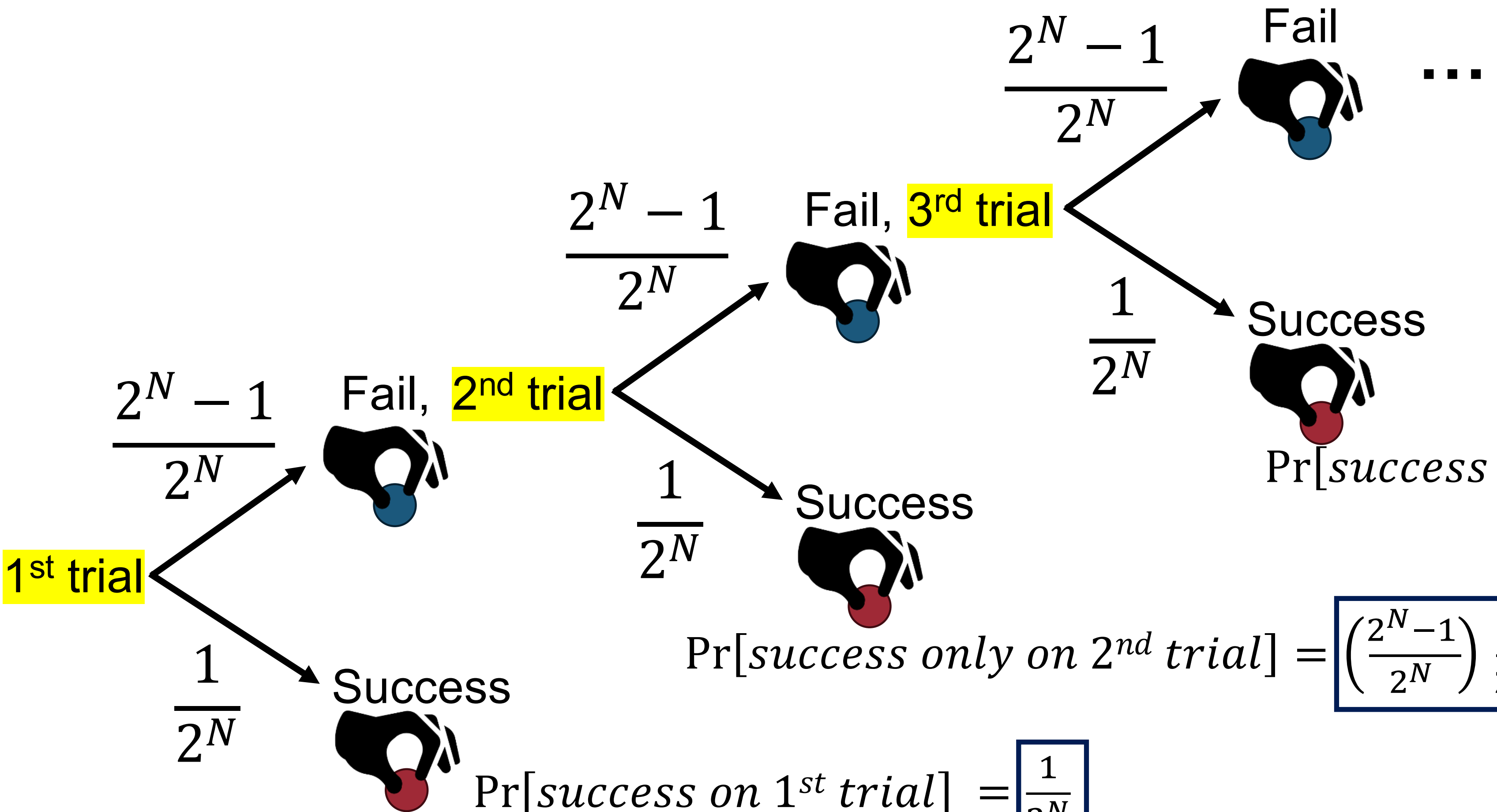
$$= \frac{1}{2^N} \sum_{k=1}^{2^N} k$$

$$= \frac{1}{2^N} \cdot \frac{2^N(2^N+1)}{2}$$

$$= \frac{2^N+1}{2}$$

Case #2: Selecting Balls w/ Replacement

(Linux)



$$\Pr[\mathit{success\ on\ } 1^{st}\ \mathit{trial}] = \boxed{\frac{1}{2^N}}$$

$$\Pr[\mathit{success\ only\ on\ } 2^{nd}\ \mathit{trial}] = \boxed{\left(\frac{2^N-1}{2^N}\right)\frac{1}{2^N}}$$

$$\Pr[\mathit{success\ only\ on\ 3rd\ trial}] = \boxed{\left(\frac{2^N-1}{2^N}\right)^2\frac{1}{2^N}}$$

Case #2: Selecting Balls w/ Replacement

(Linux)

$$\Pr[\textit{success only on } k^{th} \textit{ trial}] = \left(\frac{2^N - 1}{2^N}\right)^{k-1} \frac{1}{2^N}$$

(Classic Geometric Distribution where $p = \frac{1}{2^N}$)

- **Expected # of trials before success**

$$E[X] = \frac{1}{p}$$

$$= 2^N$$

ASLR Comparison: Windows vs. Linux

- Brute-force attack will success in

$$\frac{2^N + 1}{2} \approx 2^{N-1}$$

trials on ***Windows***

vs.

$$2^N$$

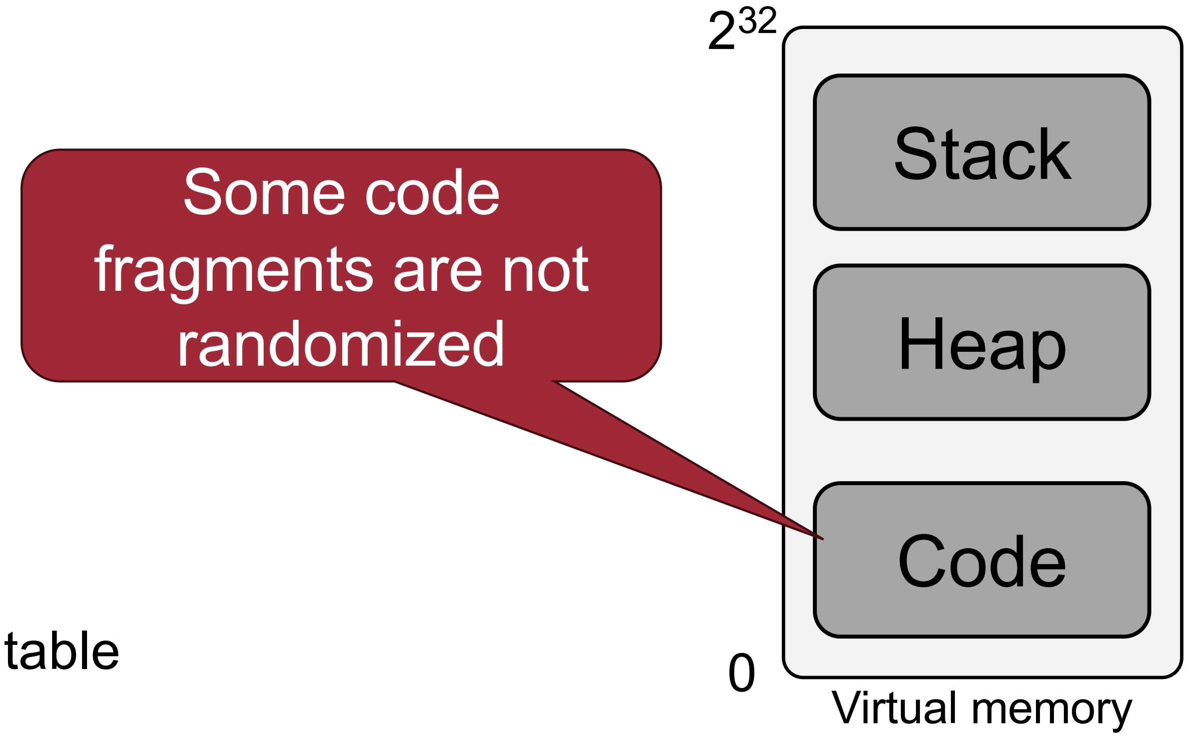
trials on ***Linux***

Linux is ≈ 2 times safer than Windows against a brute-force attack

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

Let's check the difference

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?

ROP-based Attack on Legacy Binaries

- Code sections are not randomized, hence we can use **ROP**!
- But, LIBC address is randomized (any libraries must be position-independent)! Cannot directly return to LIBC functions

But, still, relative offsets between LIBC functions are the same regardless of ASLR

Background: Library Function Call

- To understand how it is possible, you should know what really happens during the library function call
- When a program calls a library function like `open()`, `eip` does not directly transfer to the library
  - Instead, it first moves to a small code snippet called **PLT**
  - This code snippet uses a function pointer in a table called **GOT**

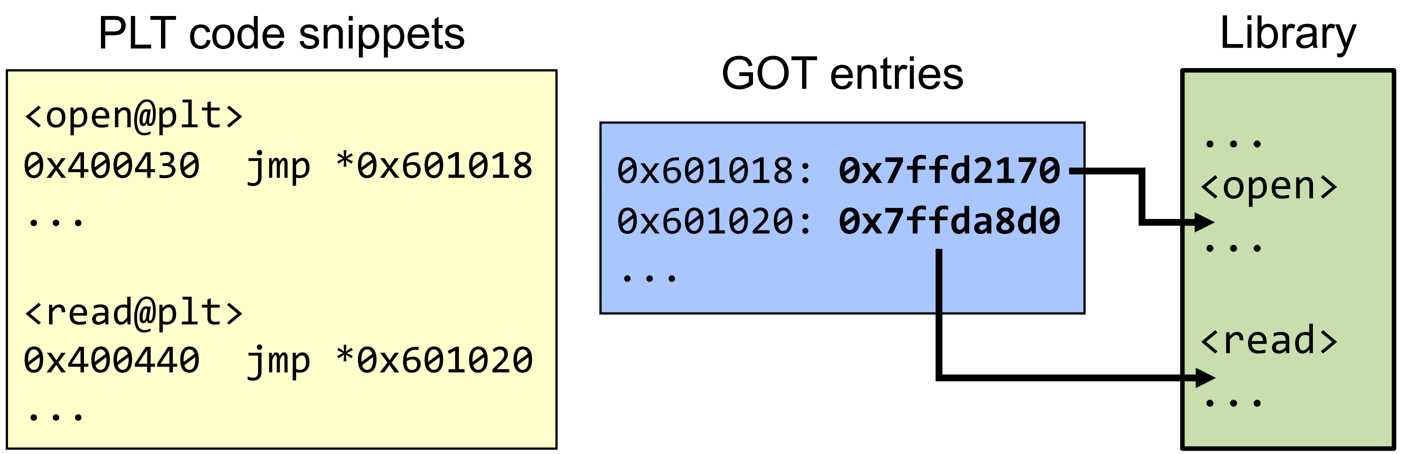
```
int main(void) {
   ...
   open(...);
   ...
}
```

```
<main>
...
0x400579:  call 0x400430 <open@plt>
...

<write@plt>
0x400430  jmp *0x601018  # GOT entry
...
```

Background: PLT and GOT

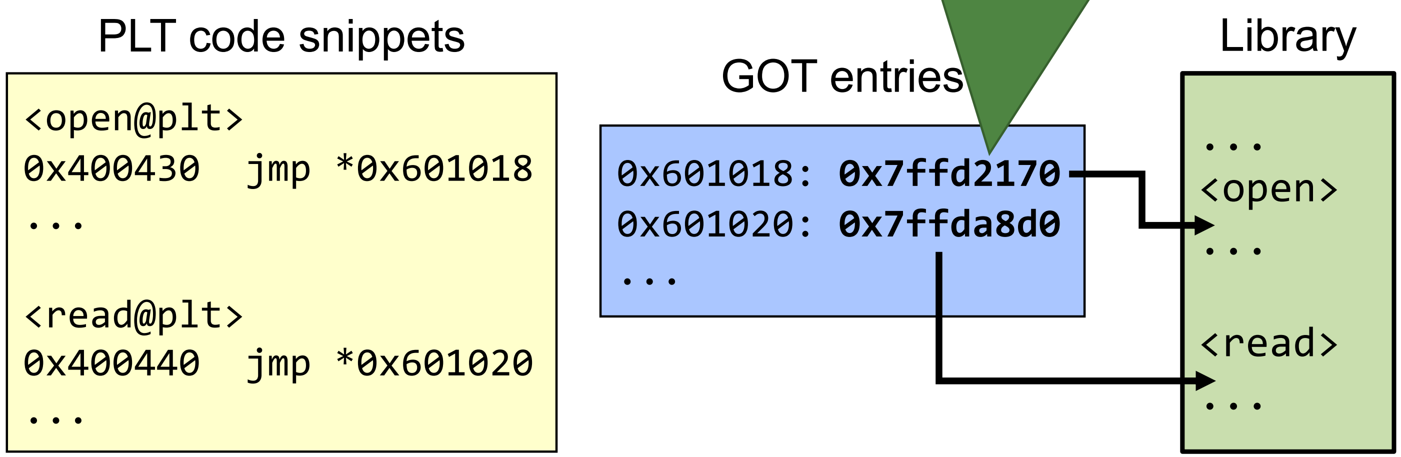
- In other words, you can think that compiler and linker implicitly generate some function pointer table and fill it
  - To enable your program to call a function in library
  - Each library function called by your program has its PLT+GOT
  - GOT is filled at runtime (cannot be determined during compile)



Background: PLT and GOT

- In other words, you can think that compiler and linker implicitly generate some function pointer table and fill it
  - To enable your program to call a func
  - Each library function called by your p
  - GOT is filled at runtime (cannot be d

We can bypass ASLR by disclosing this GOT entry!



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

x

y

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

Gadget **C**

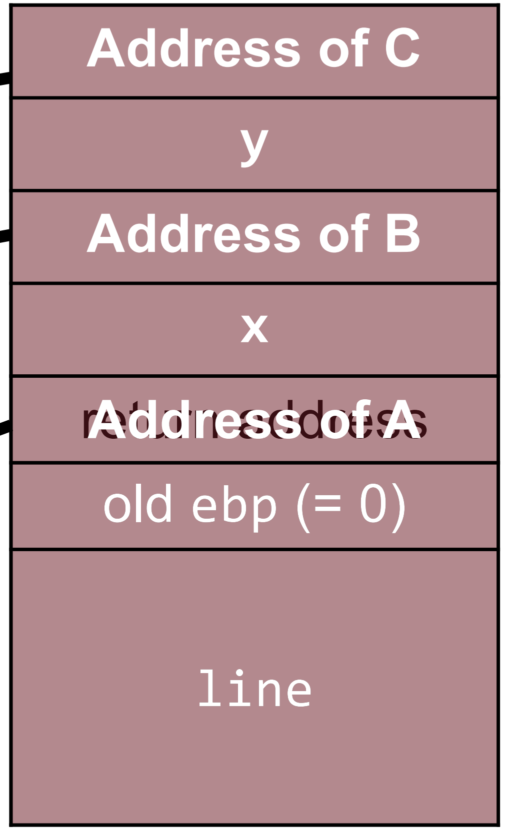
```
jmp [eax]
```

Gadget **B**

```
pop eax
add eax, edi
ret
```

Gadget **A**

```
pop edi
ret
```



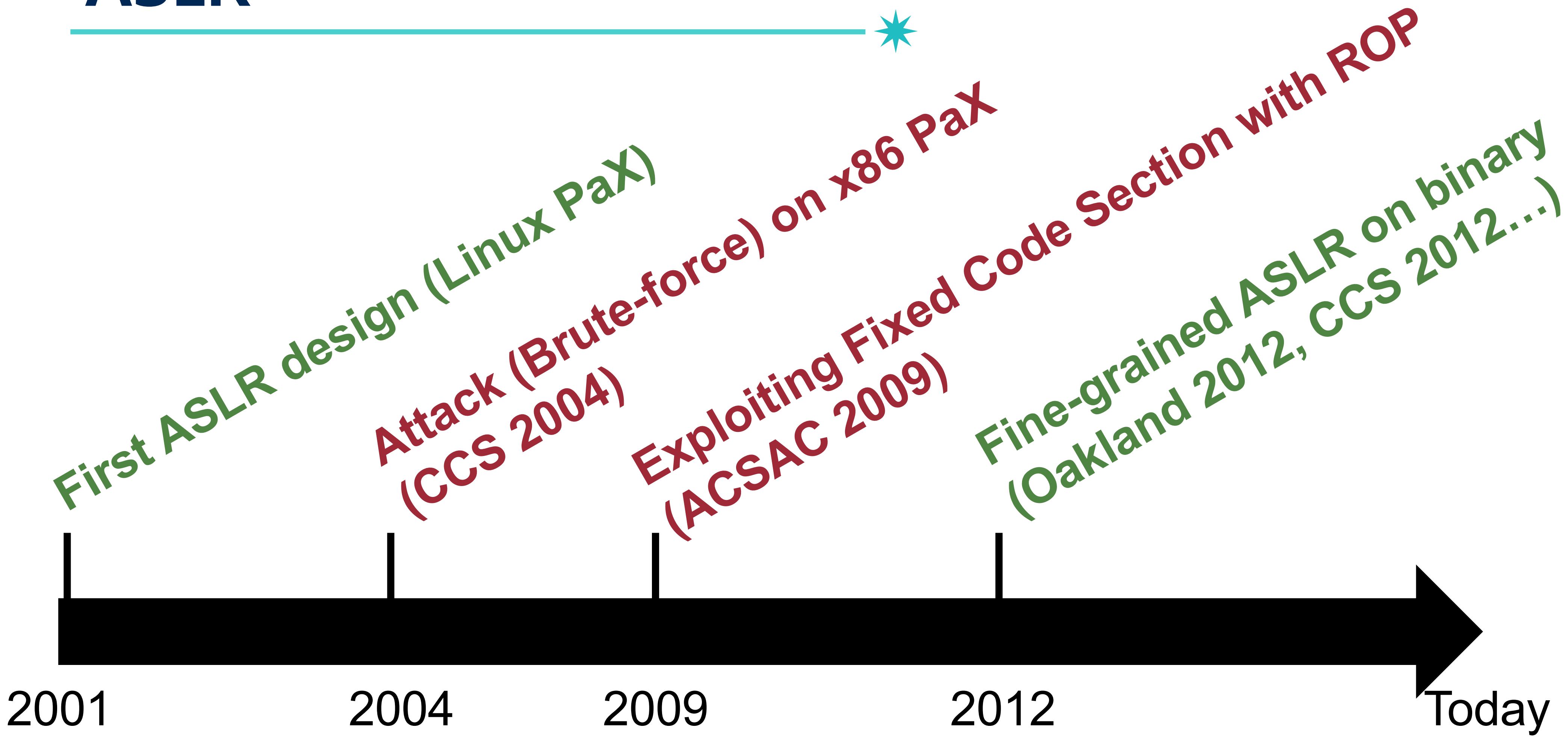
Notes on Memory Disclosure

- Is library function offset always predictable?
  - Depending on your Linux version, **`Libc`** library version will vary and the offset of each function will change, too

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)

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

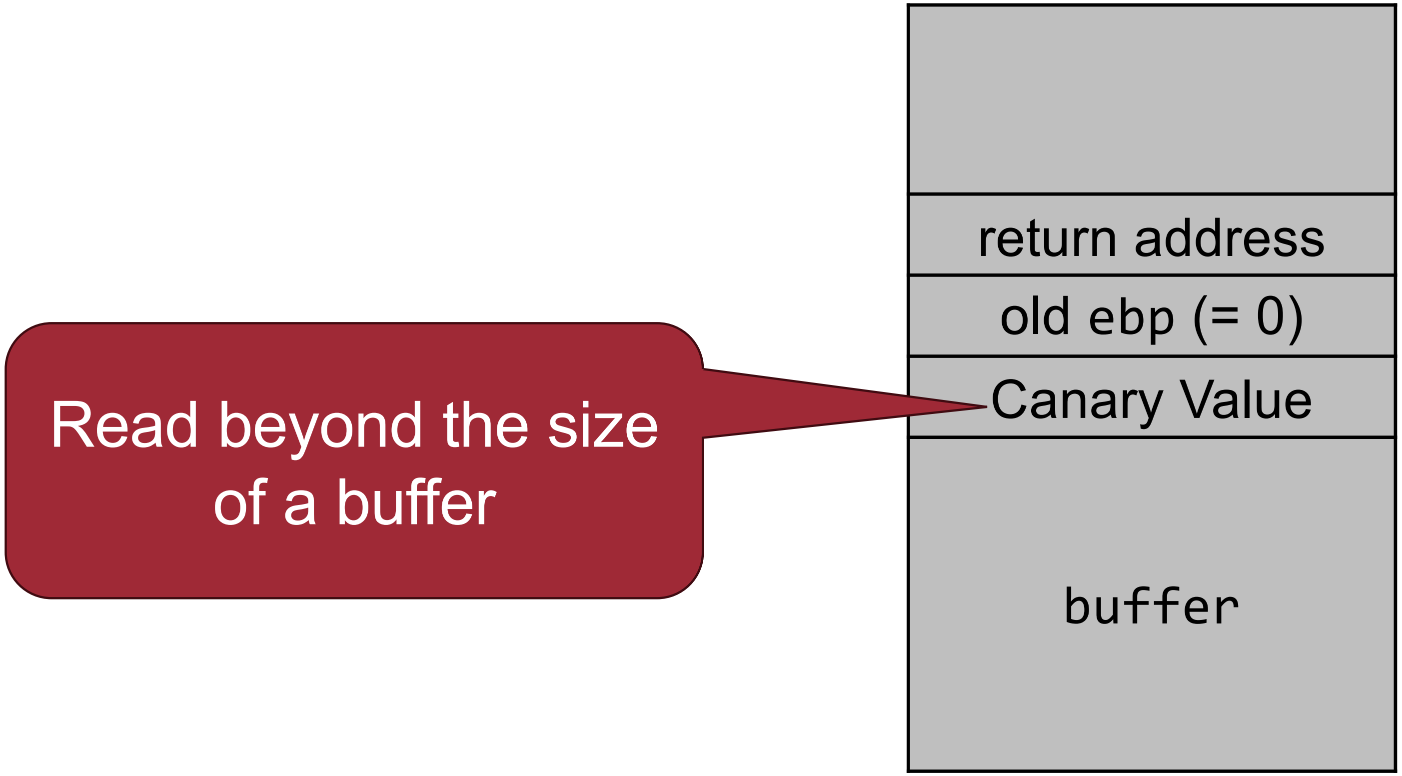
Memory Disclosure

Memory Disclosure ≠ Memory Corruption

Memory disclosure does not necessarily involve memory corruption

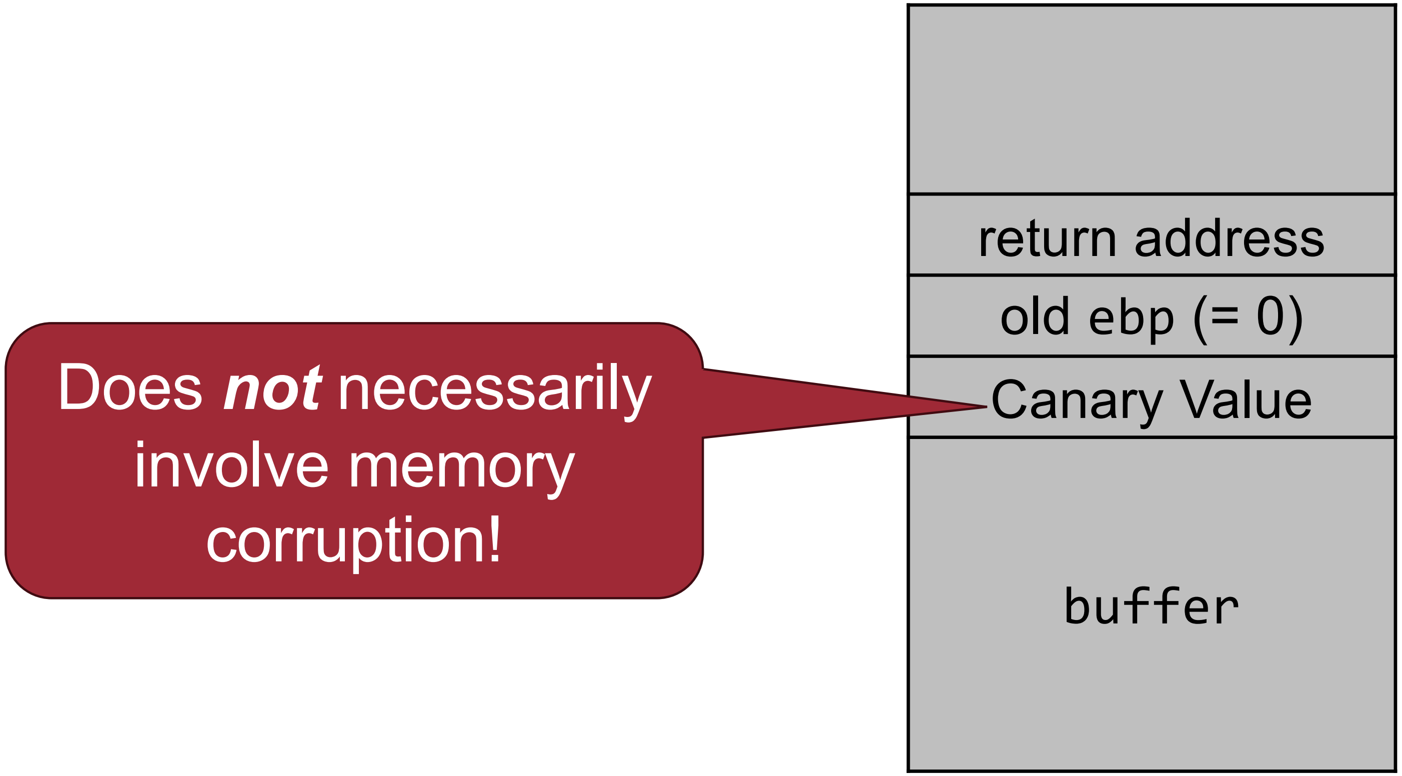
Buffer Over-Read

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



Buffer Over-Read

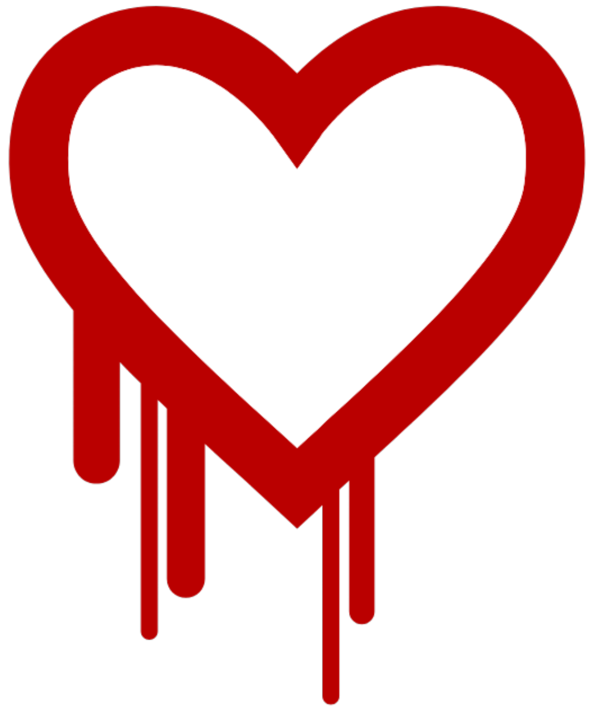
Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer



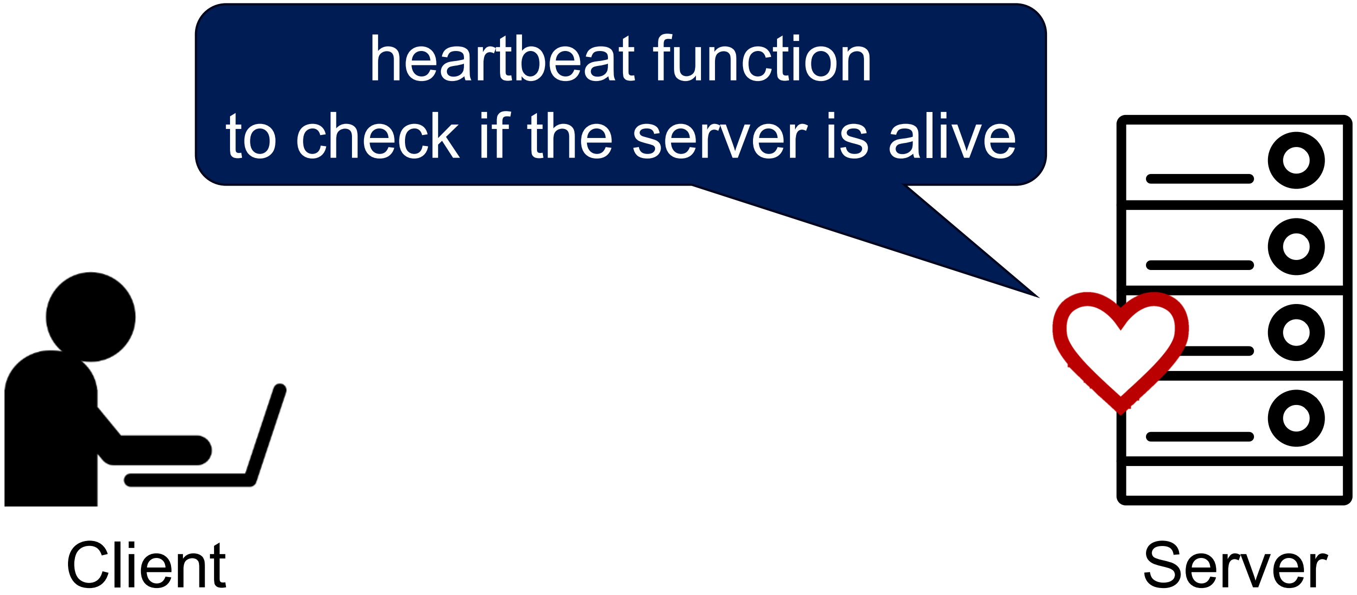
Example: Heartbleed Bug (in 2014)

- Famous bug in OpenSSL (in TLS ***heartbeat***)

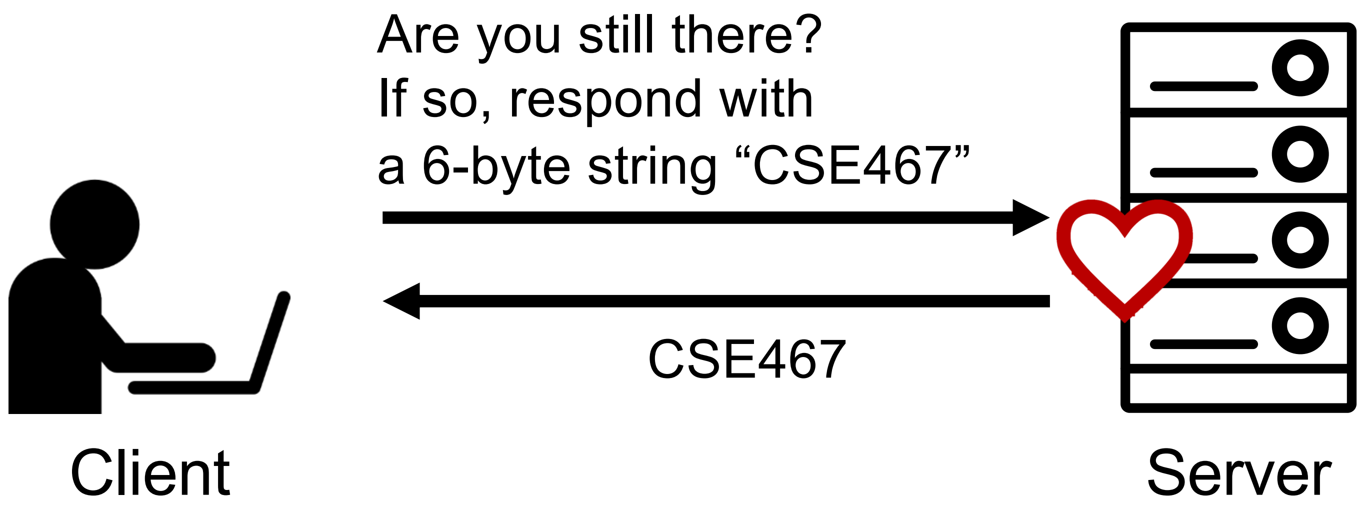
- An attacker can steal <u>private keys</u>



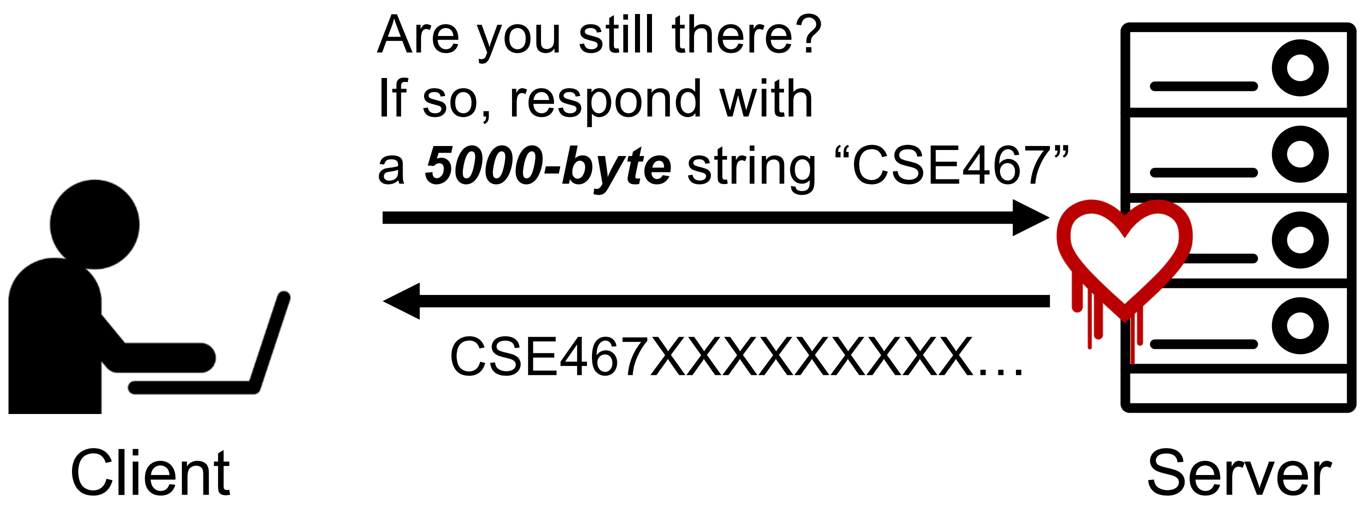
Heartbleed Bug: High-level Workflow



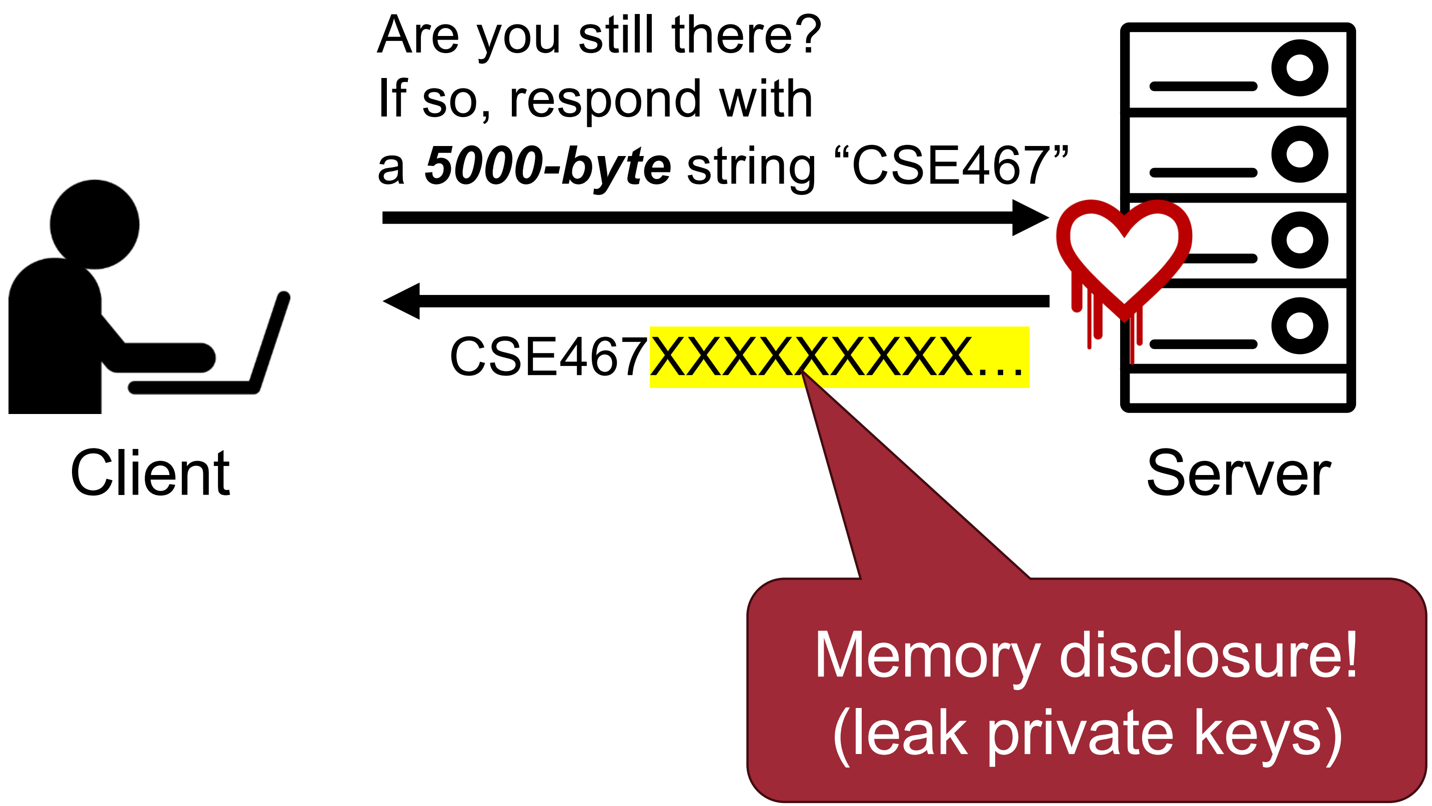
Heartbleed Bug: High-level Workflow



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
```

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

Payload obtained from `HeartbeatMessage` (i.e., CSE467)

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

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

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

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

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;

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

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

Payload obtained from HeartbeatMessage (i.e., CSE467)

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

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

Copy arbitrary memory contents of a server! TLS secret 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

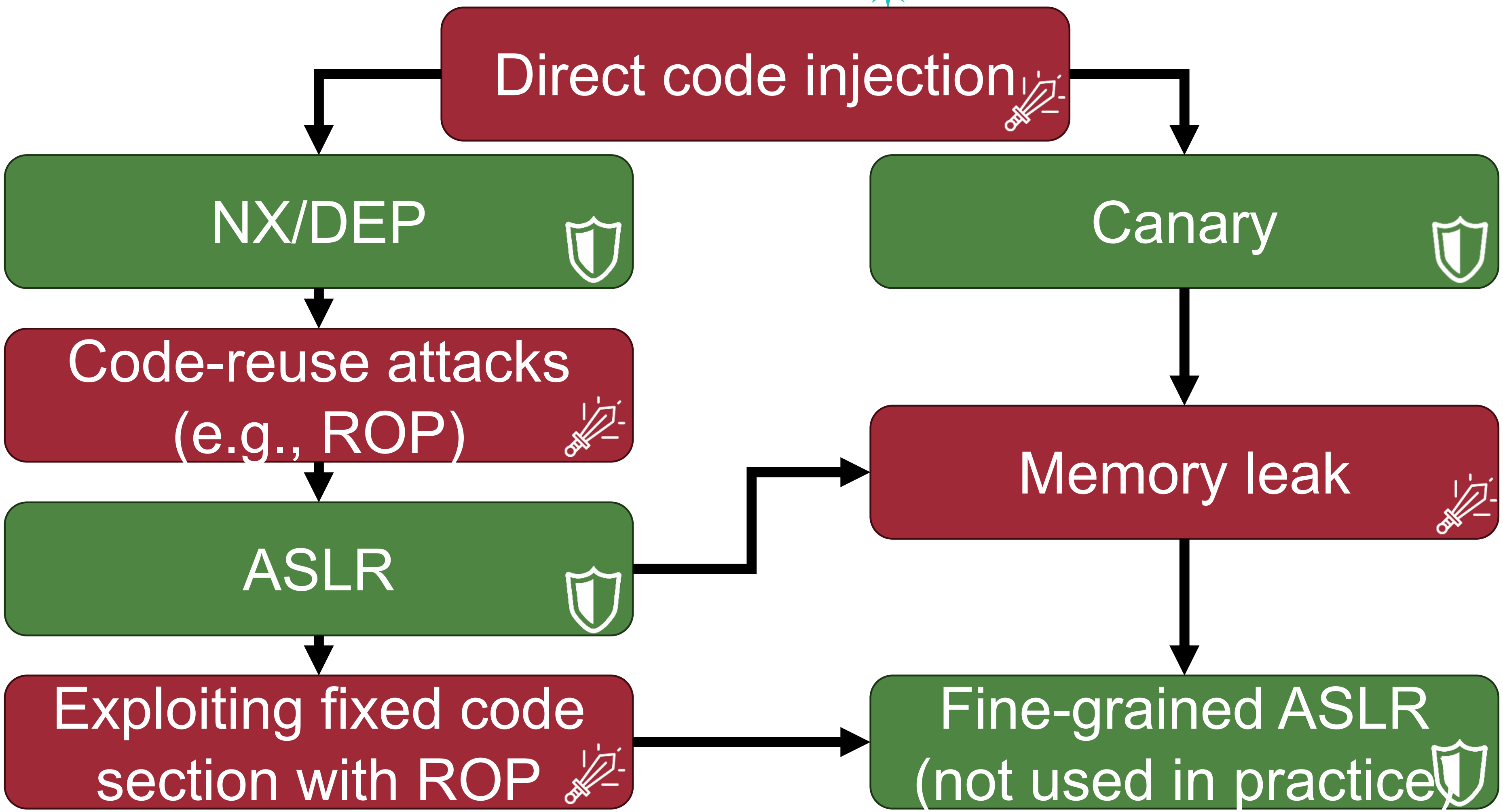
Memory Disclosure and Exploit

- 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



Summary

- 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 (≠ Memory Corruption)
- Security vs. Performance

Lessons

- Hackers are more persistent than you think
  - They often come up with creative methods to bypass mitigation

- So the impact of software vulnerabilities should not be underestimated or overlooked

- To precisely understand the outcome of a bug, it is important to know the internals of computer systems
  - Ex) If you didn’t know the existence of PLT/GOT, it would be hard to imagine how the exploit is possible

Question?