CSE467: Computer Security

11. ASLR & Memory Disclosure & Type Confusion

Seongil Wi

Department of Computer Science and Engineering

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

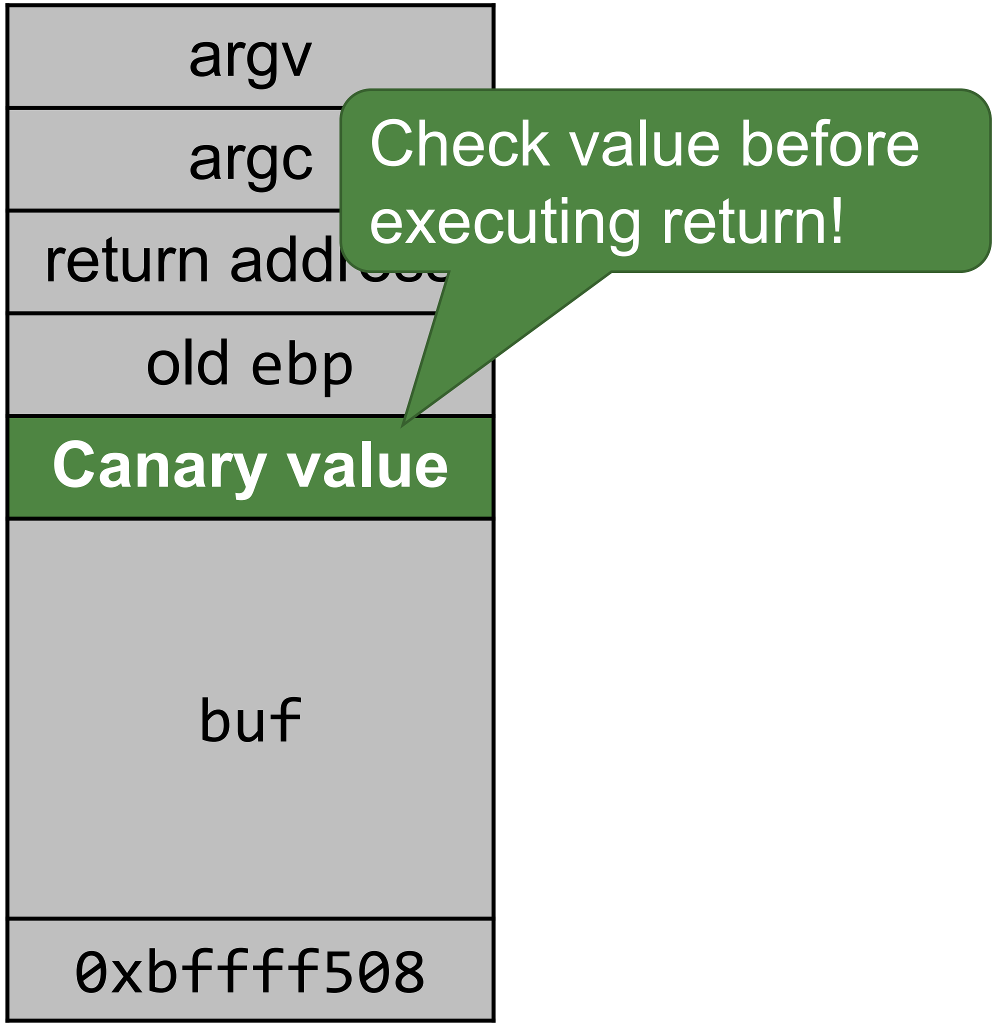
Notice

- There will be Q&A session for HW2
  - Oct. 24
    - 30 minutes lecture (It is okay to leave the room after the lecture is end)
    - 45 minutes Q&A session

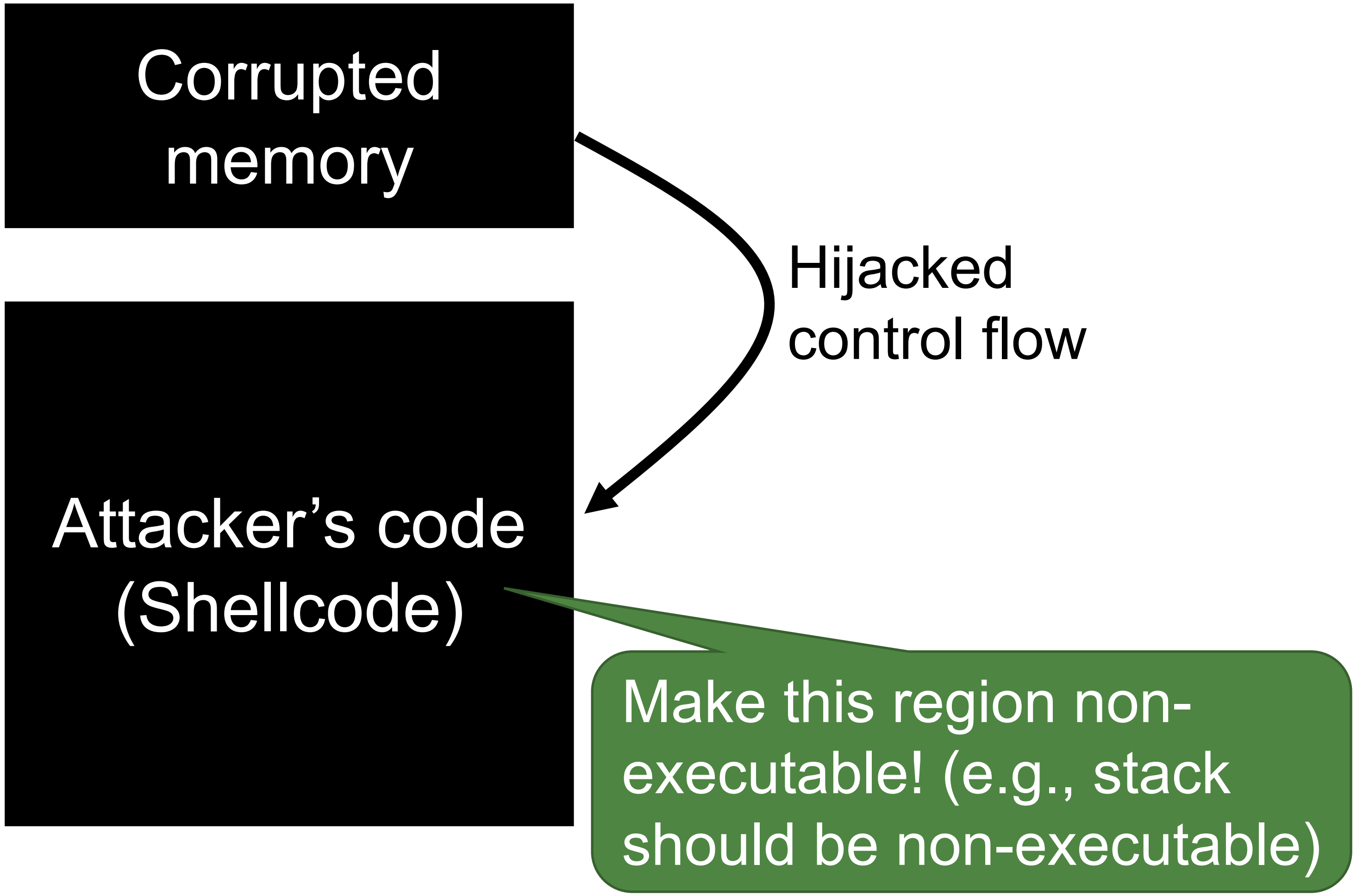
10/17/2020	Midterm weak (No exam, no class)	
10/19/2020	Midterm weak (No exam, no class)	
10/24/2023	Web Security #1	HW2 Q&A session ~~HW2 due (11:59PM)~~
10/26/2023	Web Security #2	HW2 due (11:59PM)

Recap: Mitigating Memory Corruption Bugs

Mitigation #1: Canary



Mitigation #2: NX (No eXcute)



Recap: Stack Canary (a.k.a. Stack Cookie)

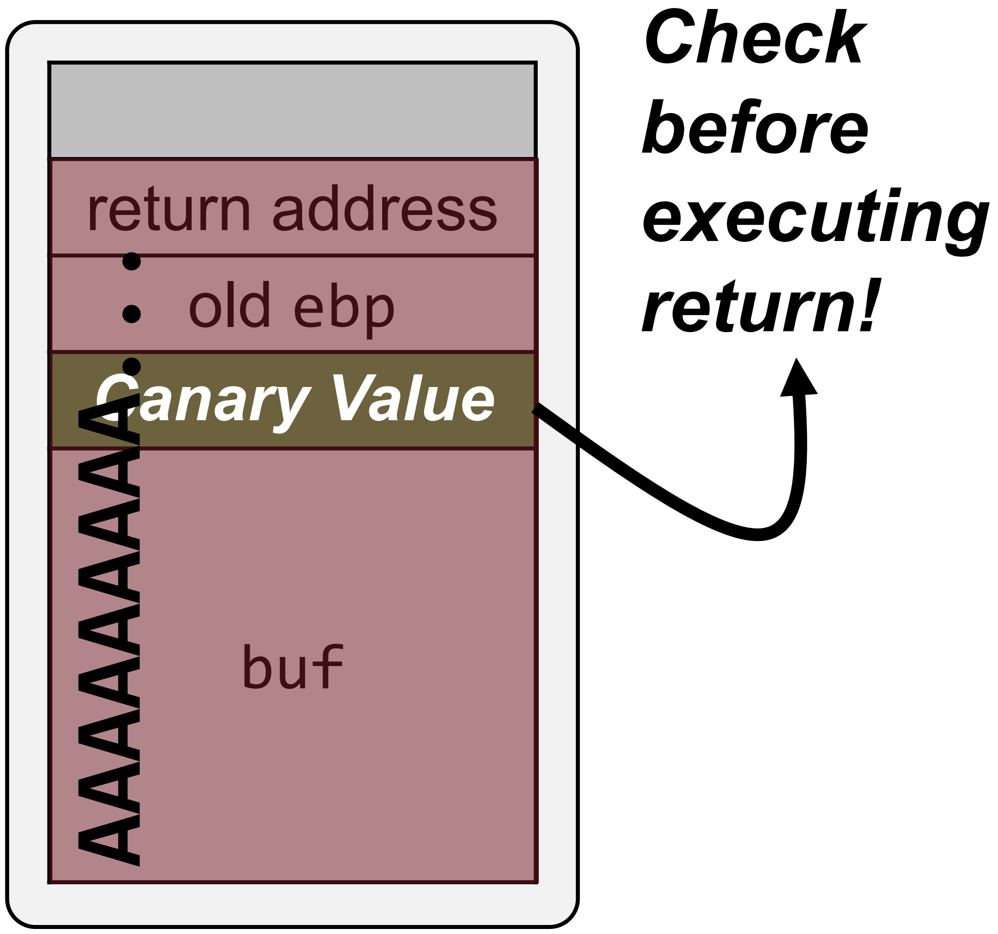
- Key idea: insert a <u>checking value</u> before the return address

Before executing return, check…

(Inserted canary value) (Current canary value)

Canary Value $\neq$ **0x41414141**

Overflow is occurred!
Stop the program

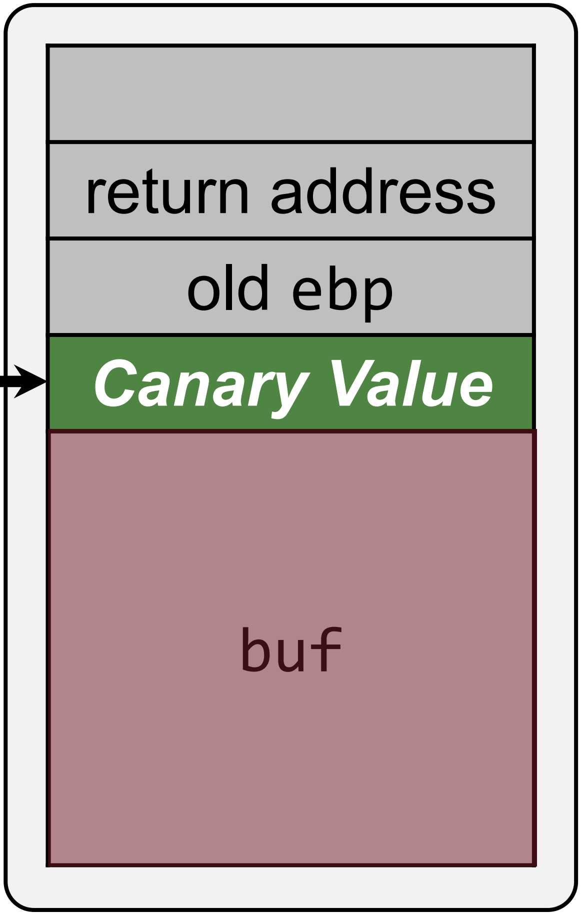


With stack canary

Recap: Byte-by-Byte Brute Forcing

Try to overwrite only 1 byte with a character from `\x00` to `\xff` until the program does not crash

Random canary:
`0x429af70c`



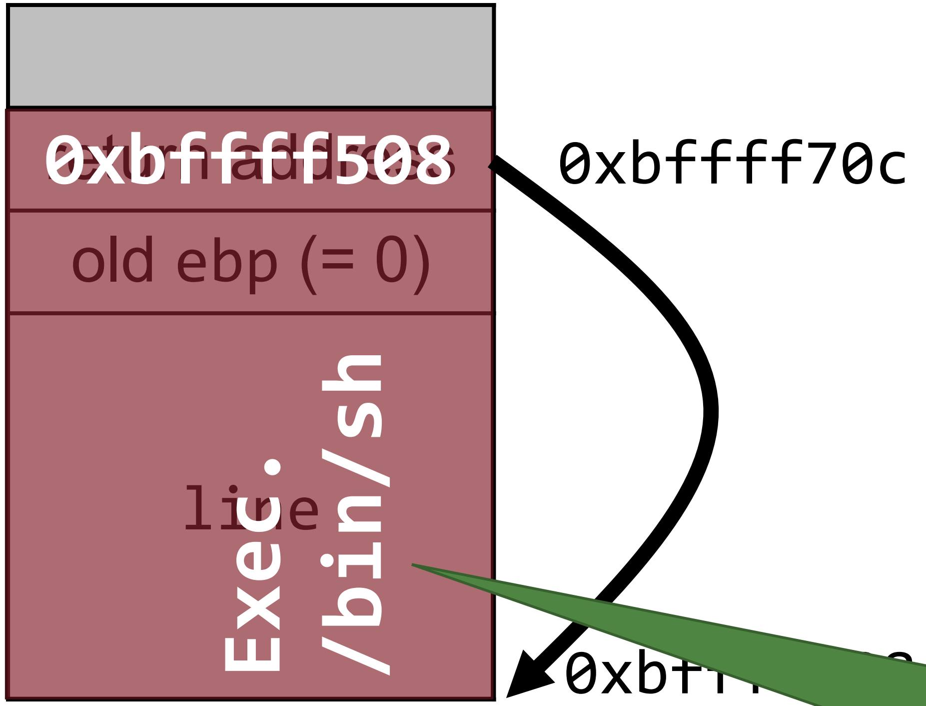
With stack canary

Recap: Leaking Canary Value

- If there is another vulnerability that allows us to ***leak*** stack contents, then we can easily bypass the canary check

- Canary is inherently vulnerable to *format string attacks*

Recap: DEP



Make this region ***non-executable***!
(e.g., stack should be non-executable)

Recap: Code Reuse Attacks

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

Recap: Return-to-Libc

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

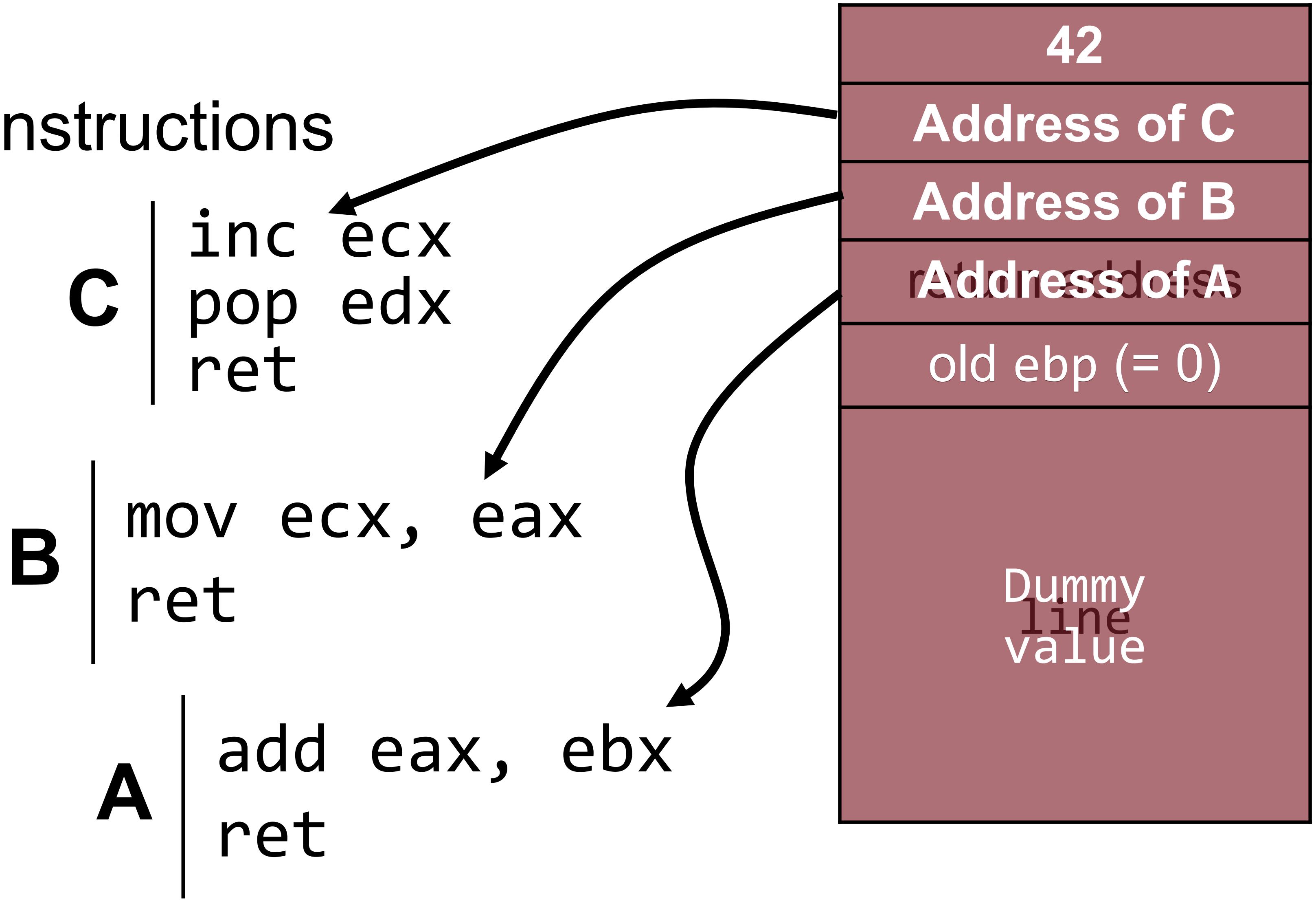
- Many useful functions in LIBC to execute
  - `exec` family: `execl`, `execlp`, `execle`, …
  - `system`
  - `mprotect`
  - `mmap`

Recap: ROP

Attacker's goal:

execute following instructions

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



Recap: ROP

Attacker's goal:
execute following instructions

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

C

```
inc ecx
pop edx
ret
```

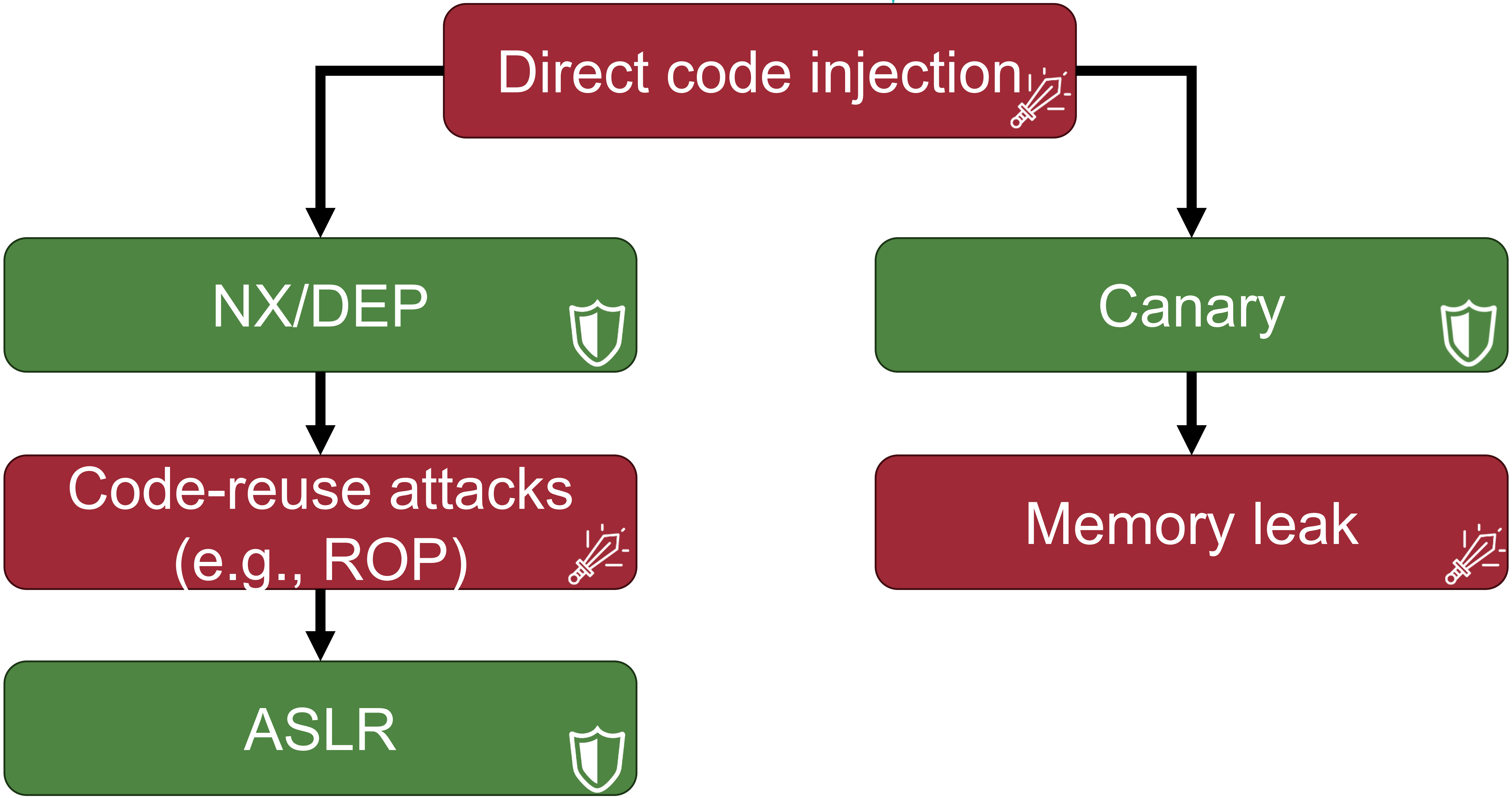
B

```
mov ecx, eax
ret
```

42
Address of C
Address of B
Address of A
old ebp (= 0)
Dummy line

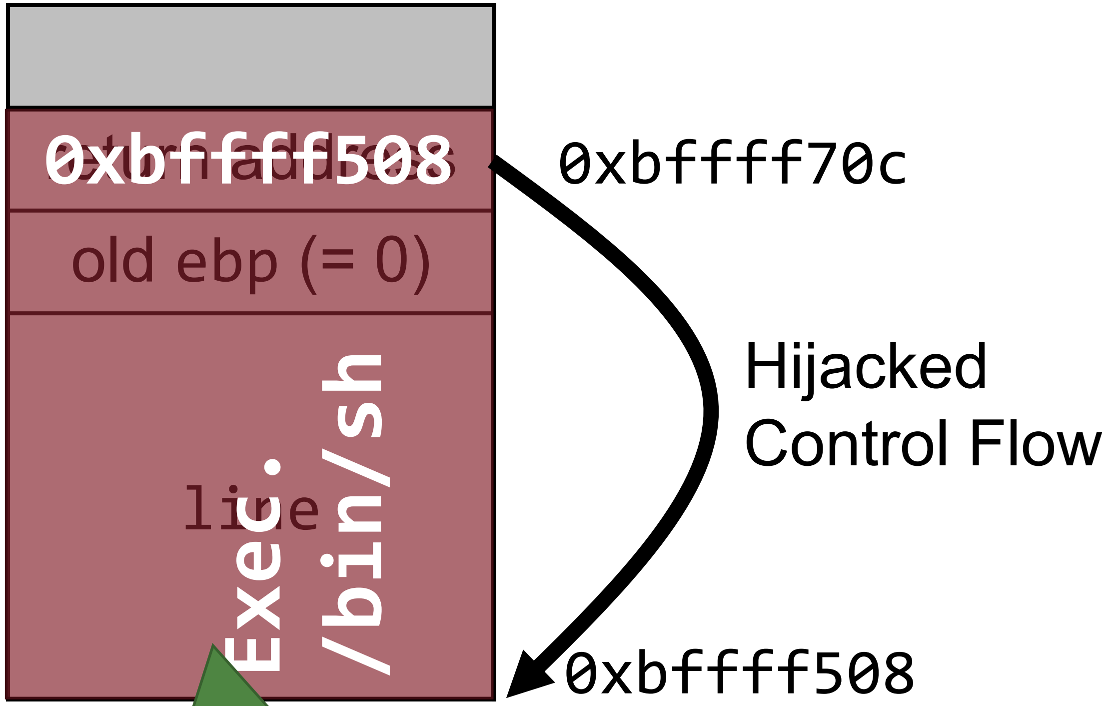
Return chaining with ROP gadgets
allows **arbitrary computation!**

Control Hijack Attack / Defense So Far



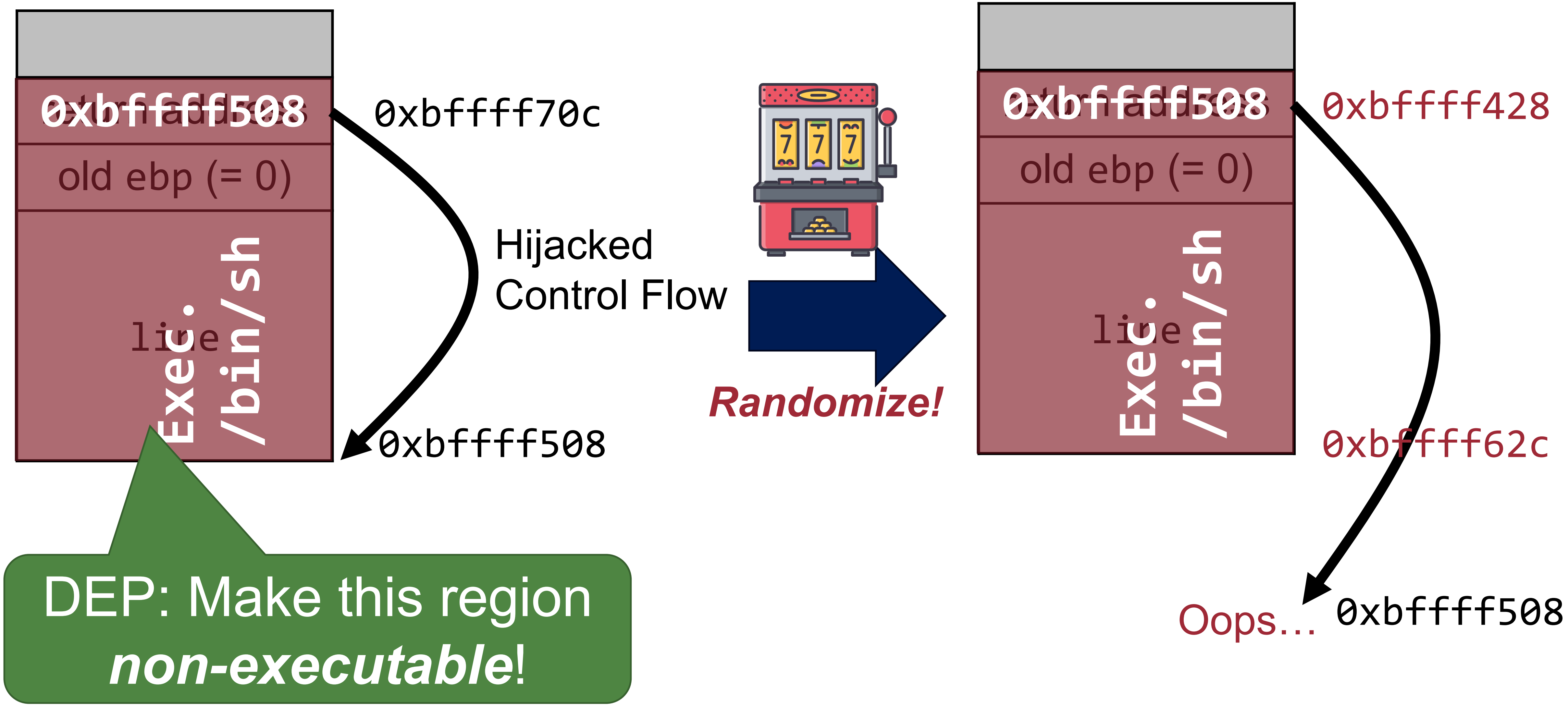
Address Space Layout Randomization (ASLR)

Control Flow Hijack Attack

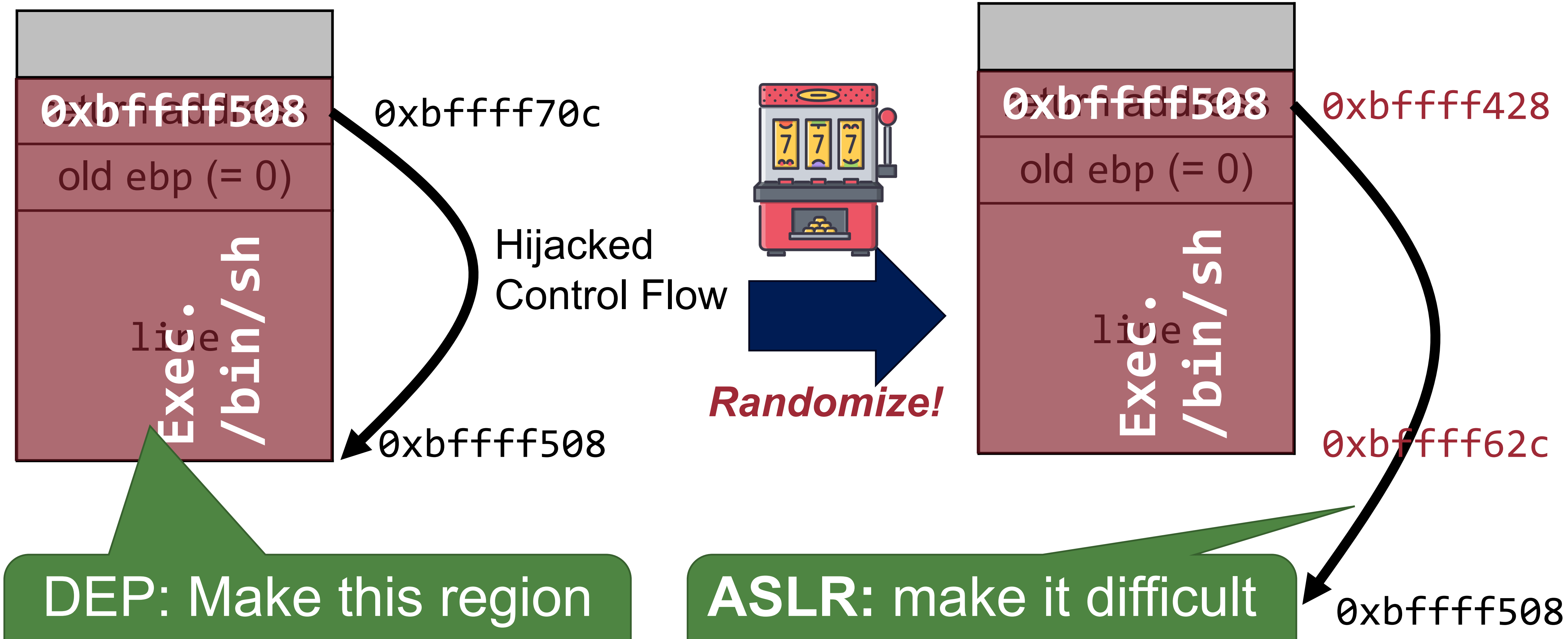


DEP: Make this region
non-executable!

Different Perspective: ASLR



Different Perspective: ASLR



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
```

DEMO

World with ASLR

- ASLR is ON by default [Ubuntu-Security]

You can enable ASLR by:

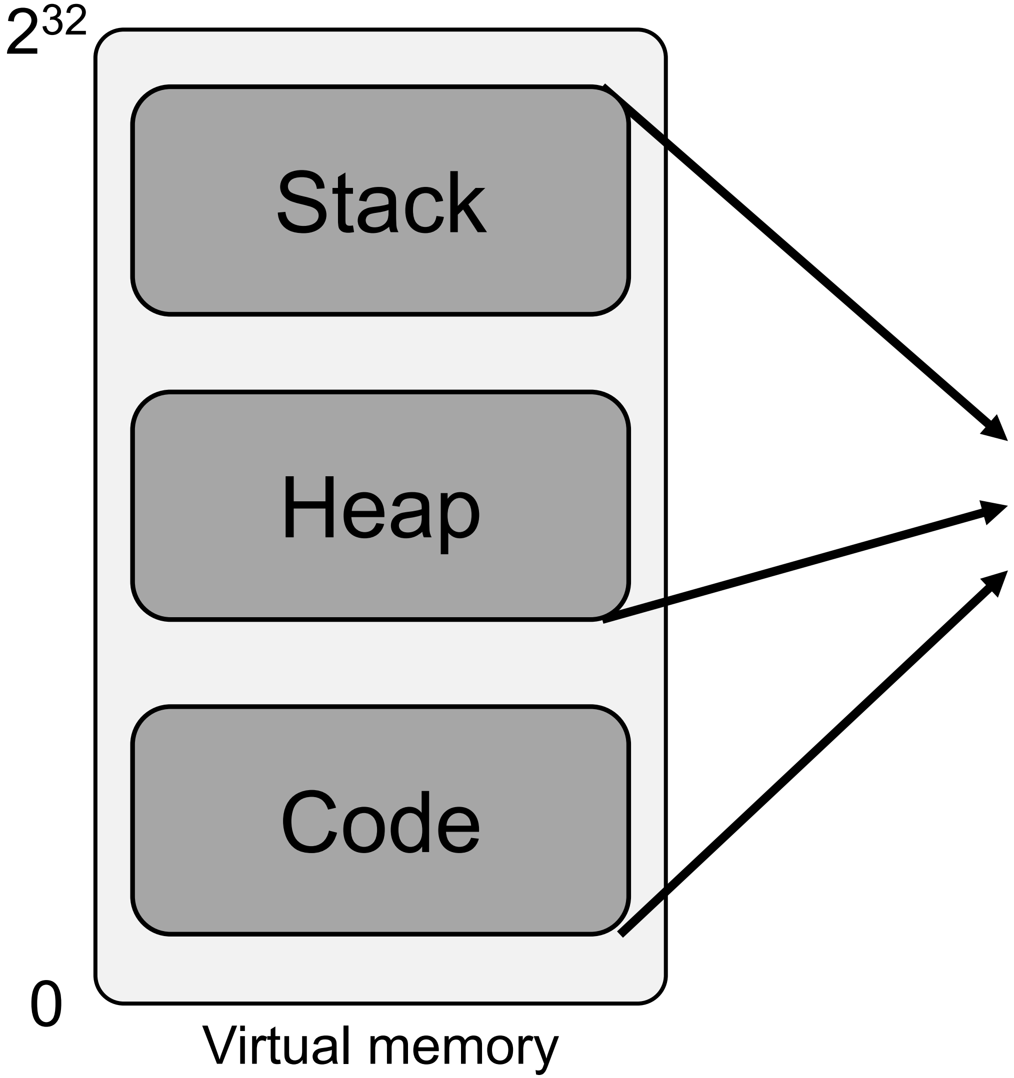
```
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

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

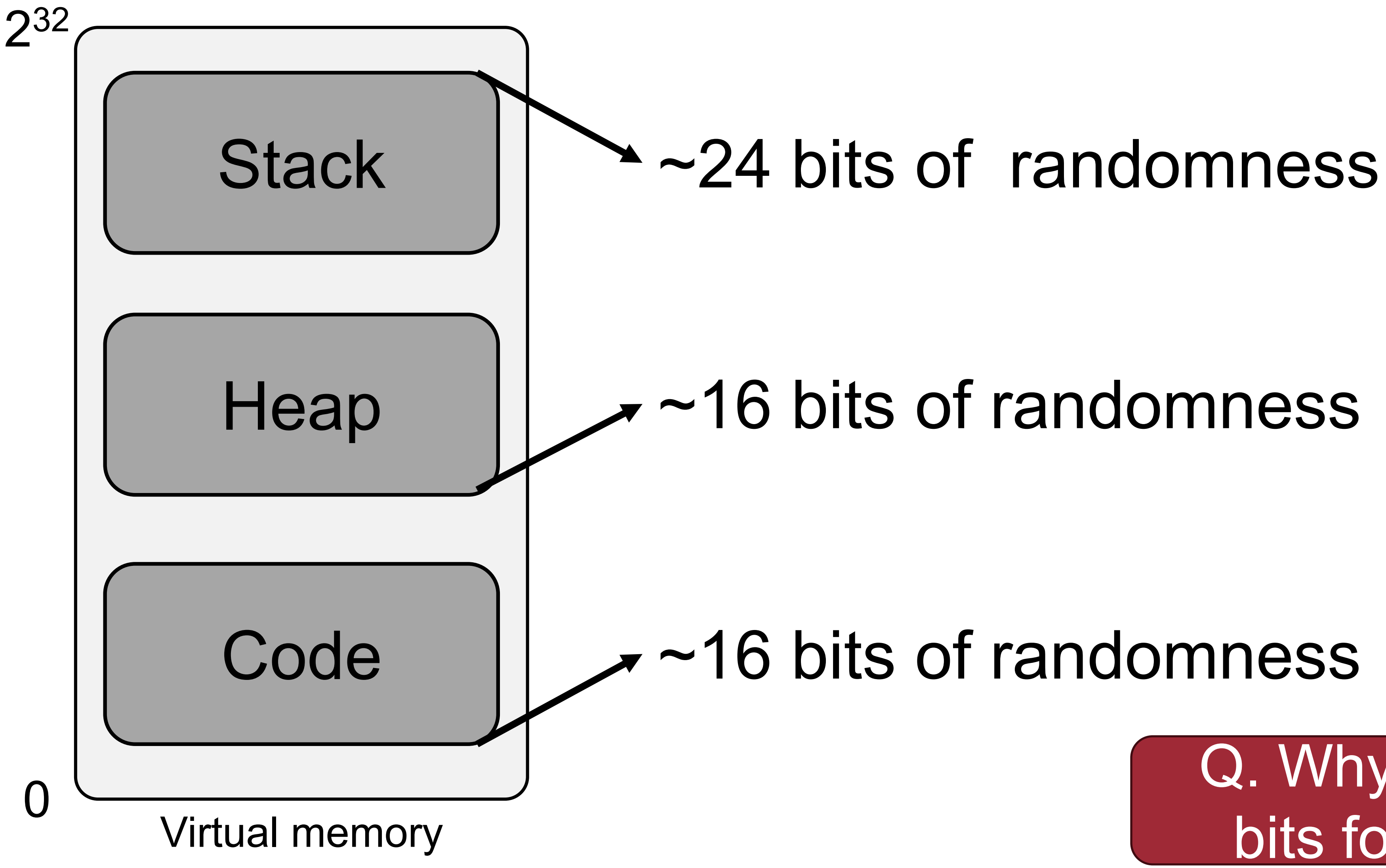
ASLR Randomizes Virtual Memory Areas



Randomized **based addresses**

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

ASLR Randomizes Virtual Memory Areas



Q. Why not fully utilize 32 bits for randomization?

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

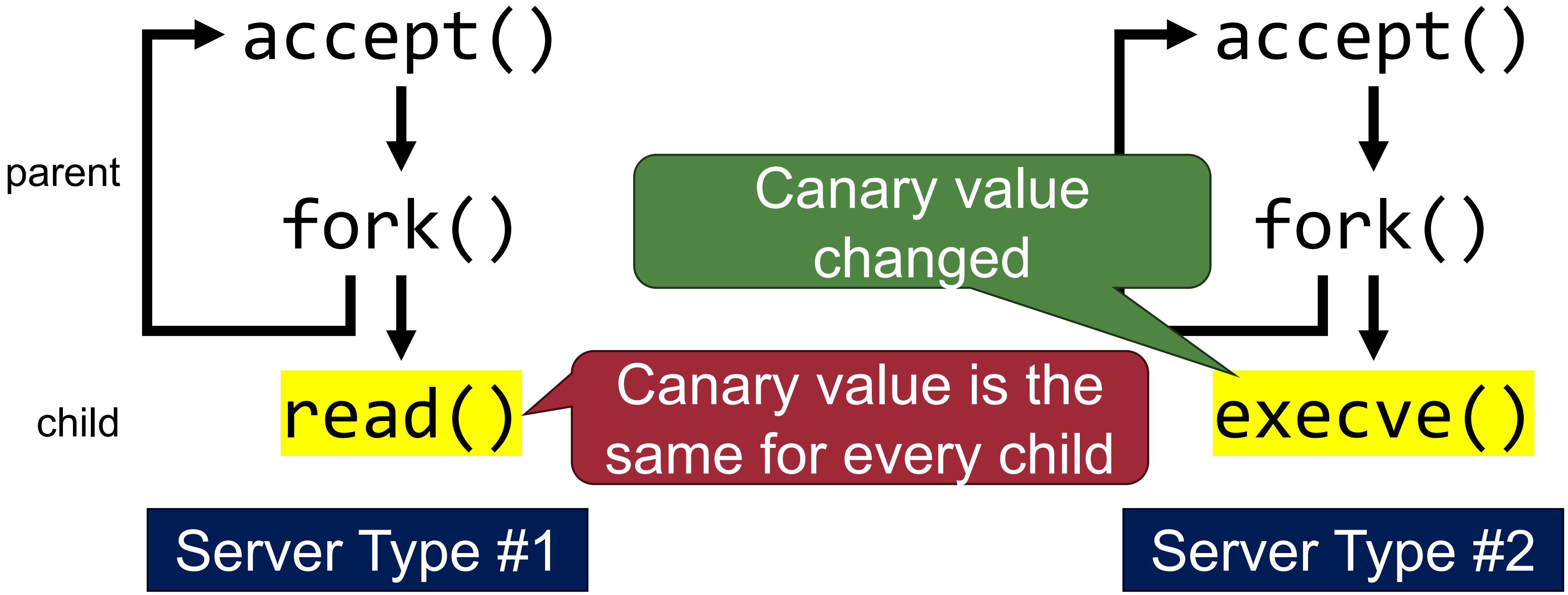
Attack #1: Entropy is Small on x86

- 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

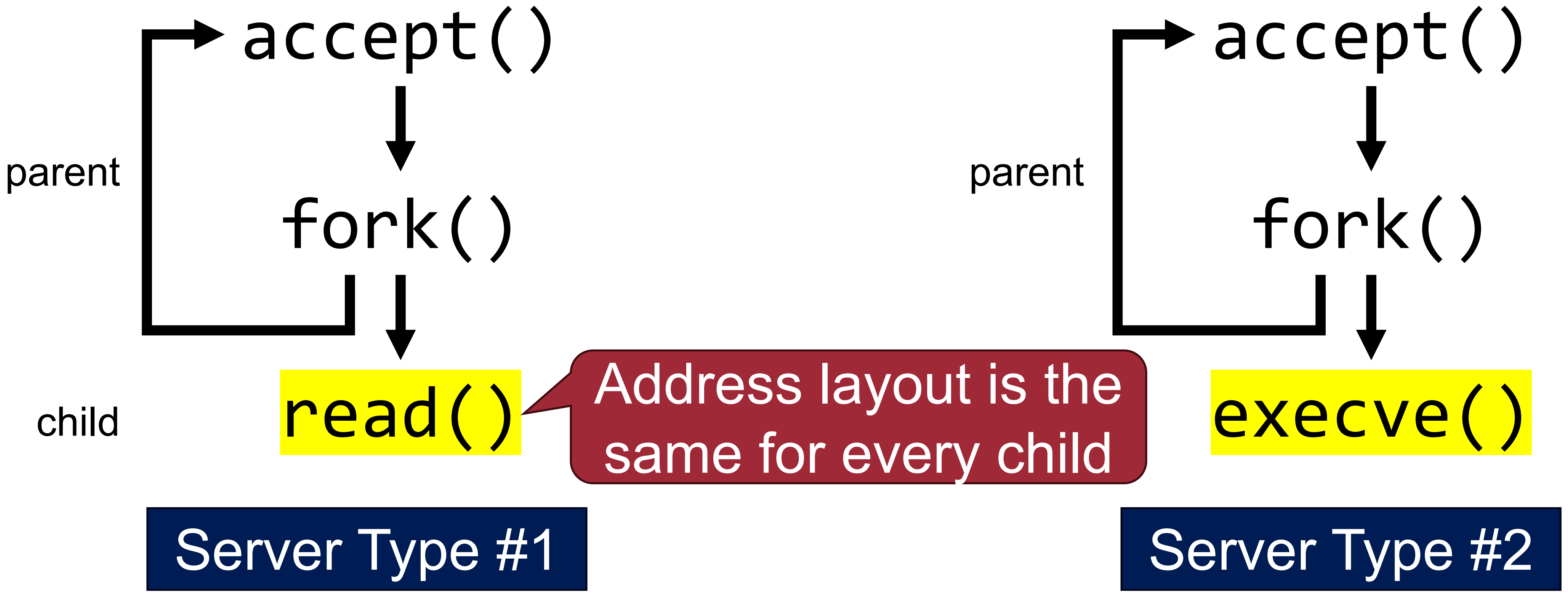
- Uses a random canary value for ***every process creation***



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

Brute-forcing Attack Example

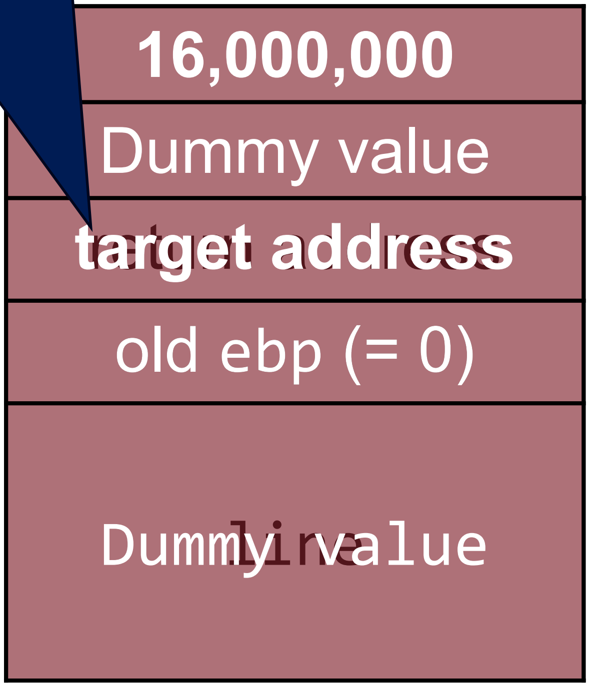
- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability

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
- 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`

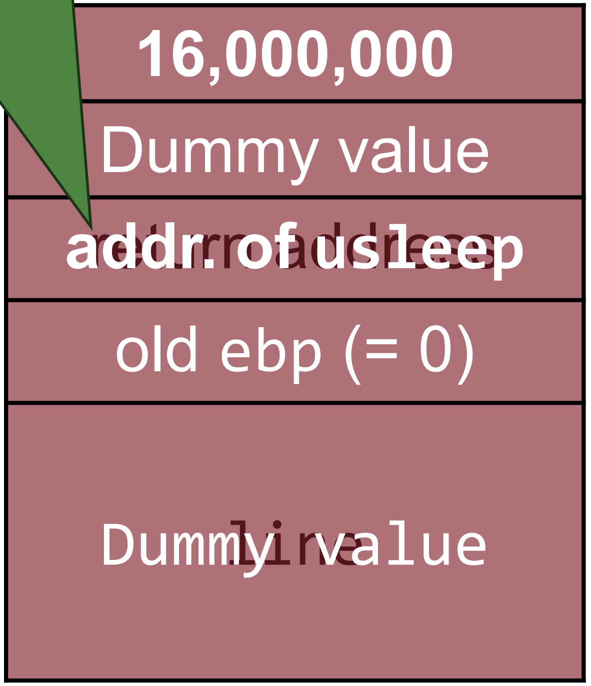


Brute-forcing Attack Example

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

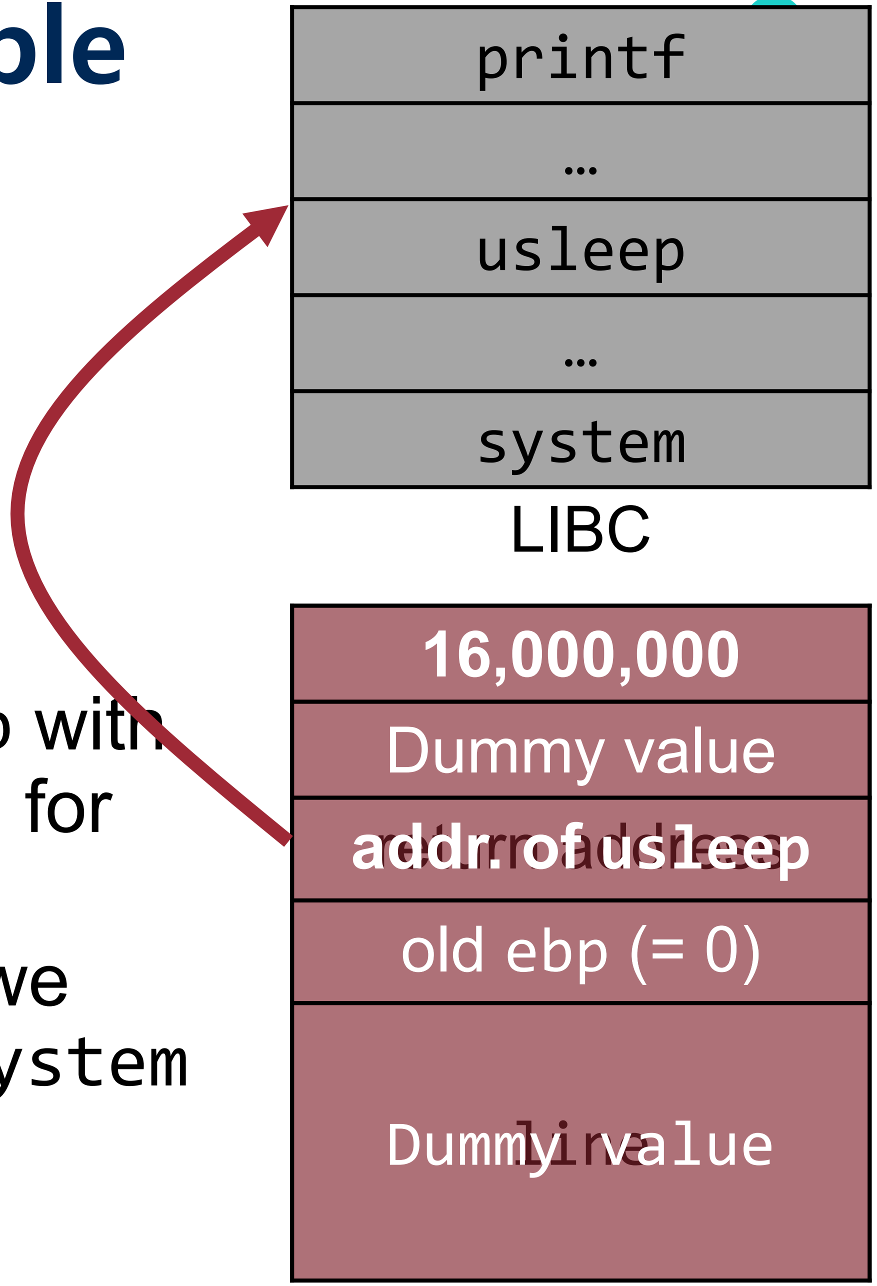
If correct, the server will wait 16 seconds

- 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)



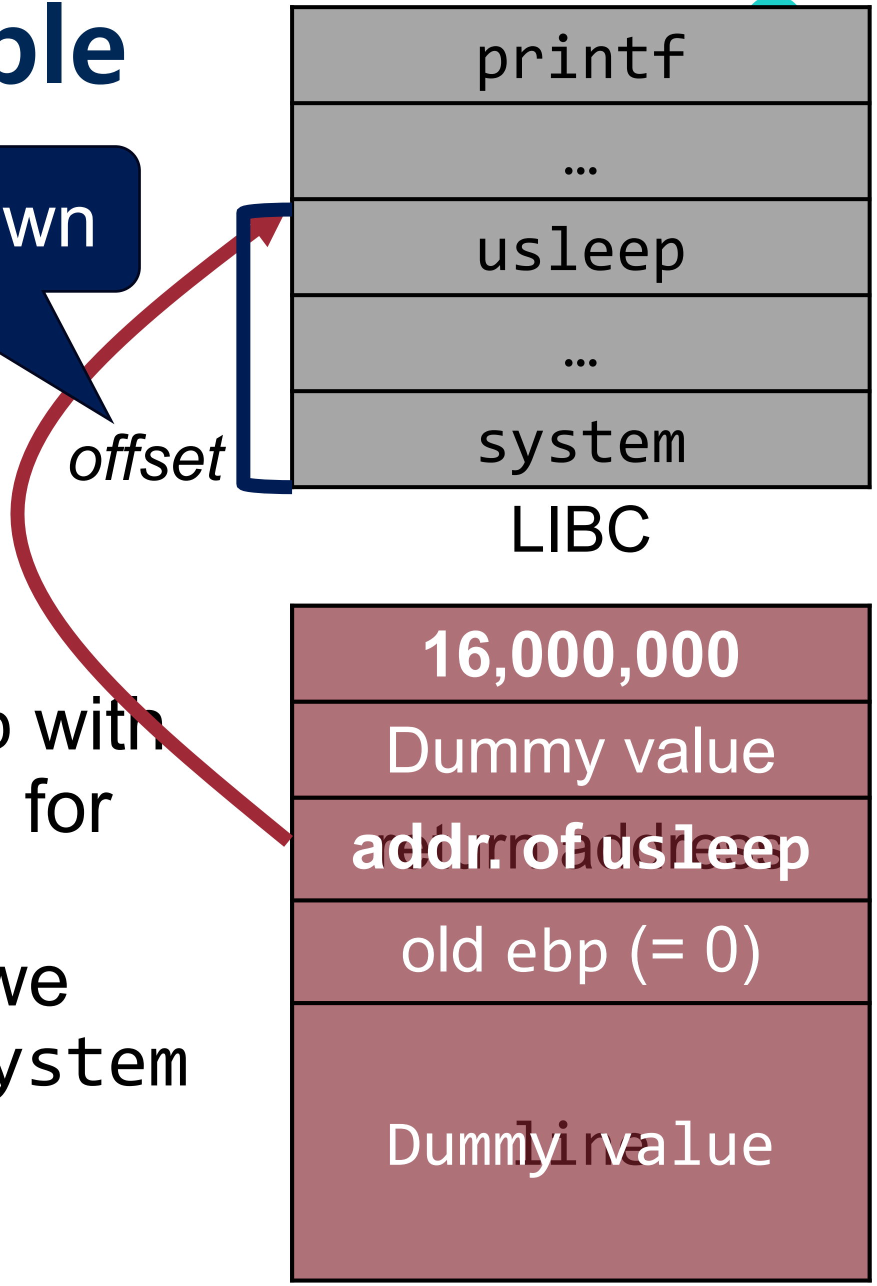
Brute-forcing Attack Example

- 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

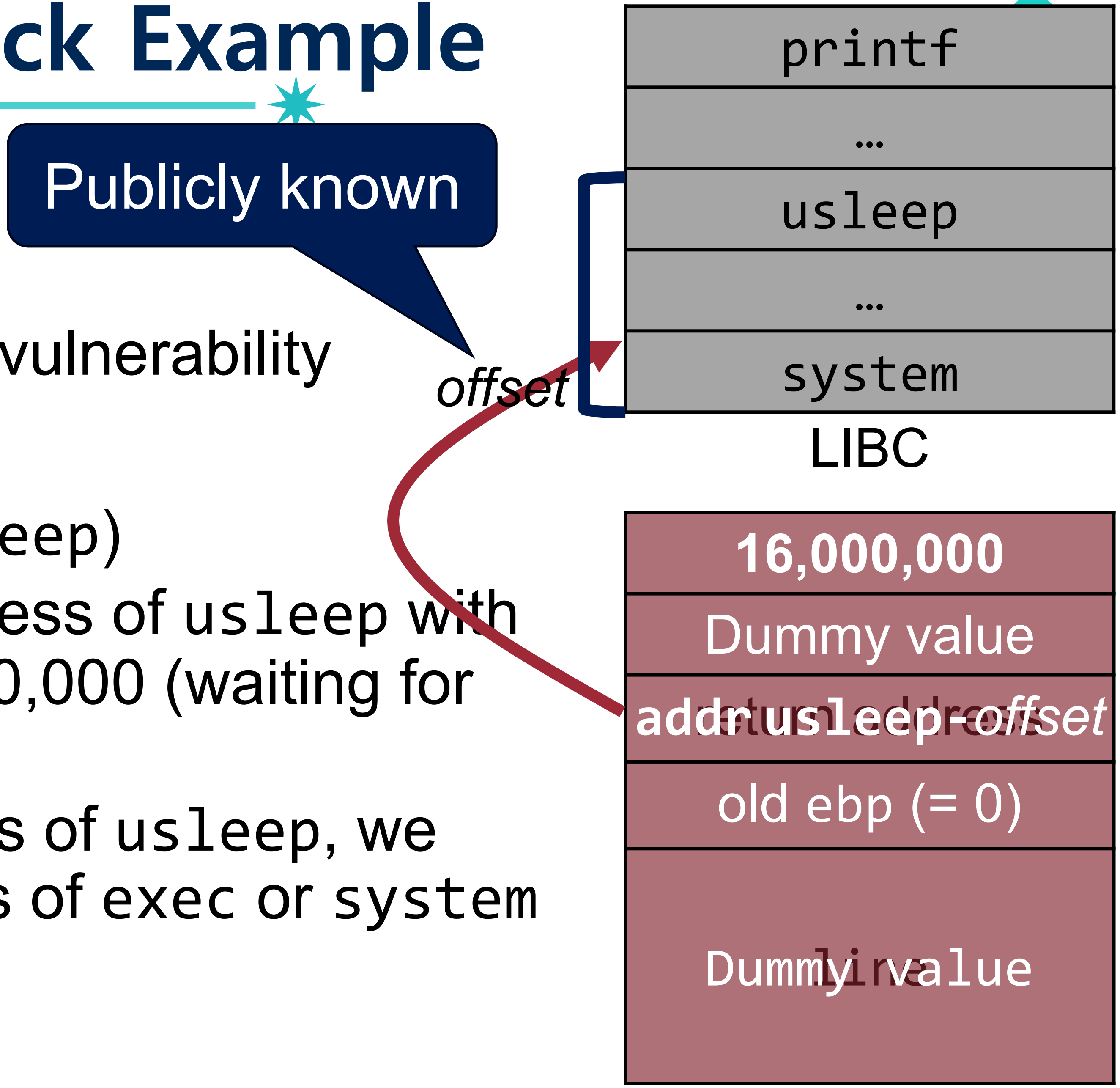
- Target: Apache web server
  - Forks children on requests
- Vulnerability: Buffer overflow vulnerability
- Method: Return-to-LIBC (`usleep`)
  - Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 seconds)
  - Once we know the address of `usleep`, we can determine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

Slower: relocation fixups for every execution

How about security?

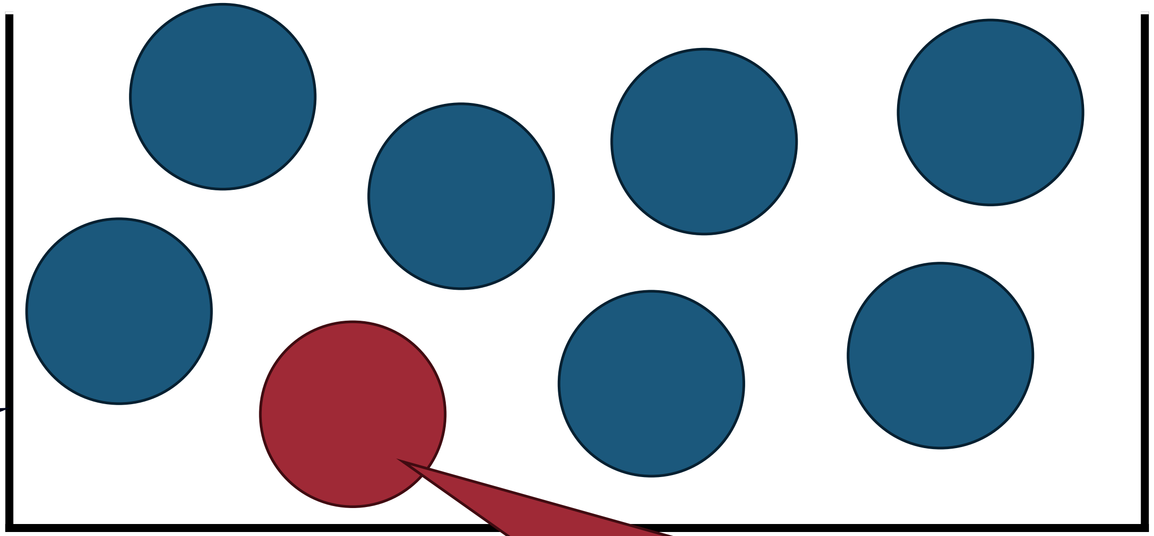
Security: Which One is Better?

- What is the expected number of trials 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.

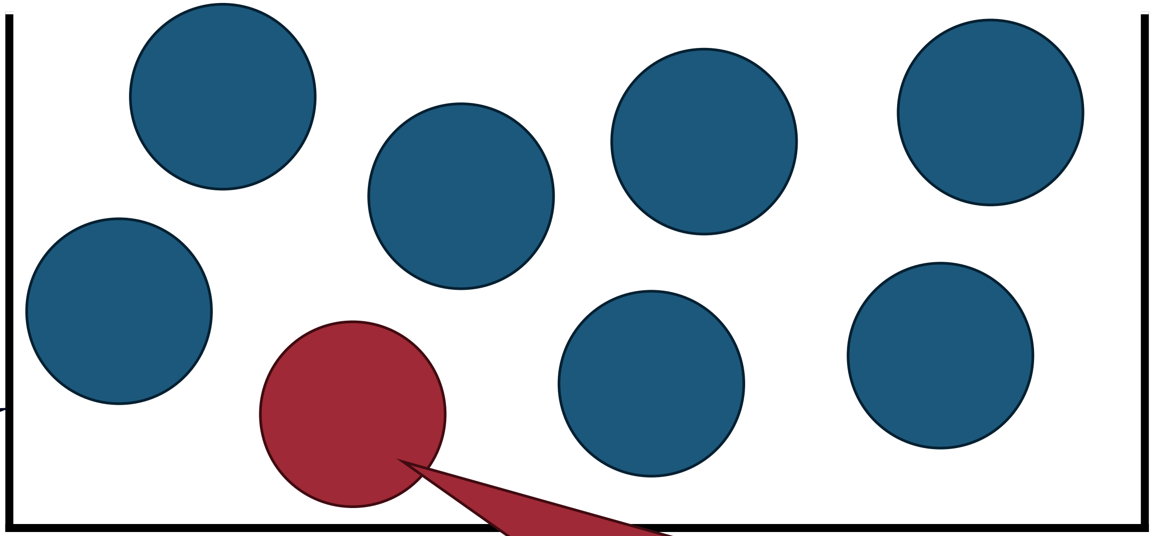
What is the probability of selecting the ***red ball***?

$2^N - 1$ 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



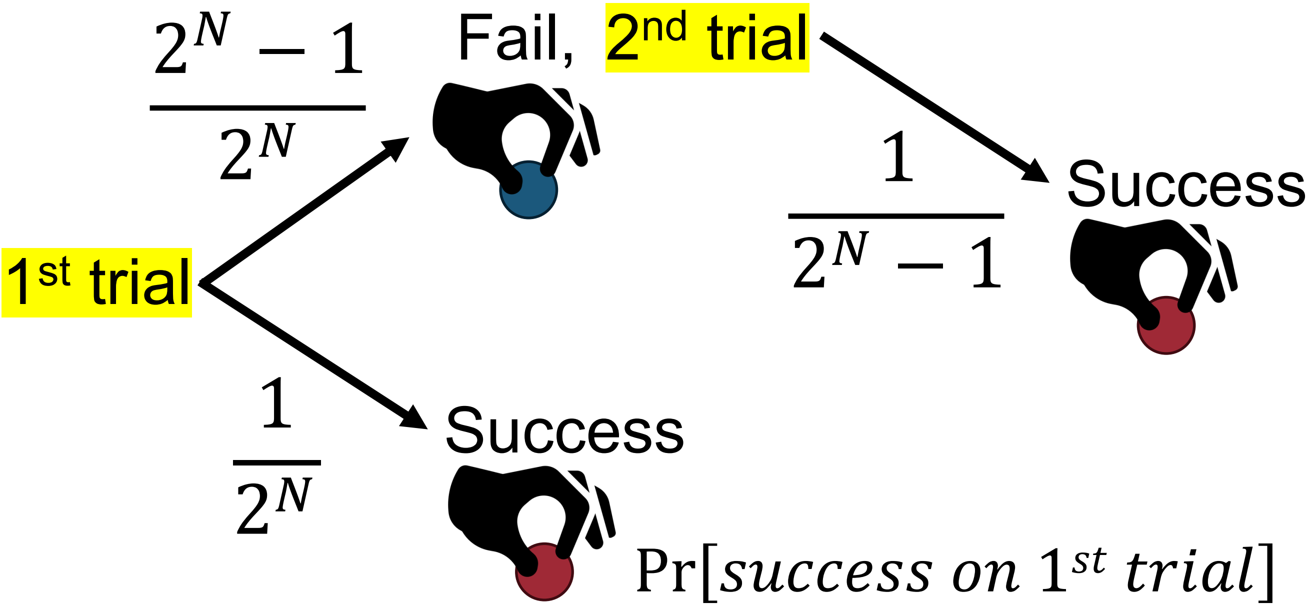
One red ball in a jar, which corresponds to the expected base address.

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

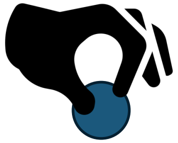
(Windows)

Without replacement!

1st trial

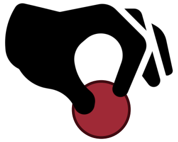
$\frac{2^N - 1}{2^N}$

Fail, 2nd trial



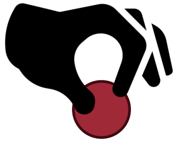
$\frac{1}{2^N - 1}$

Success



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

Success

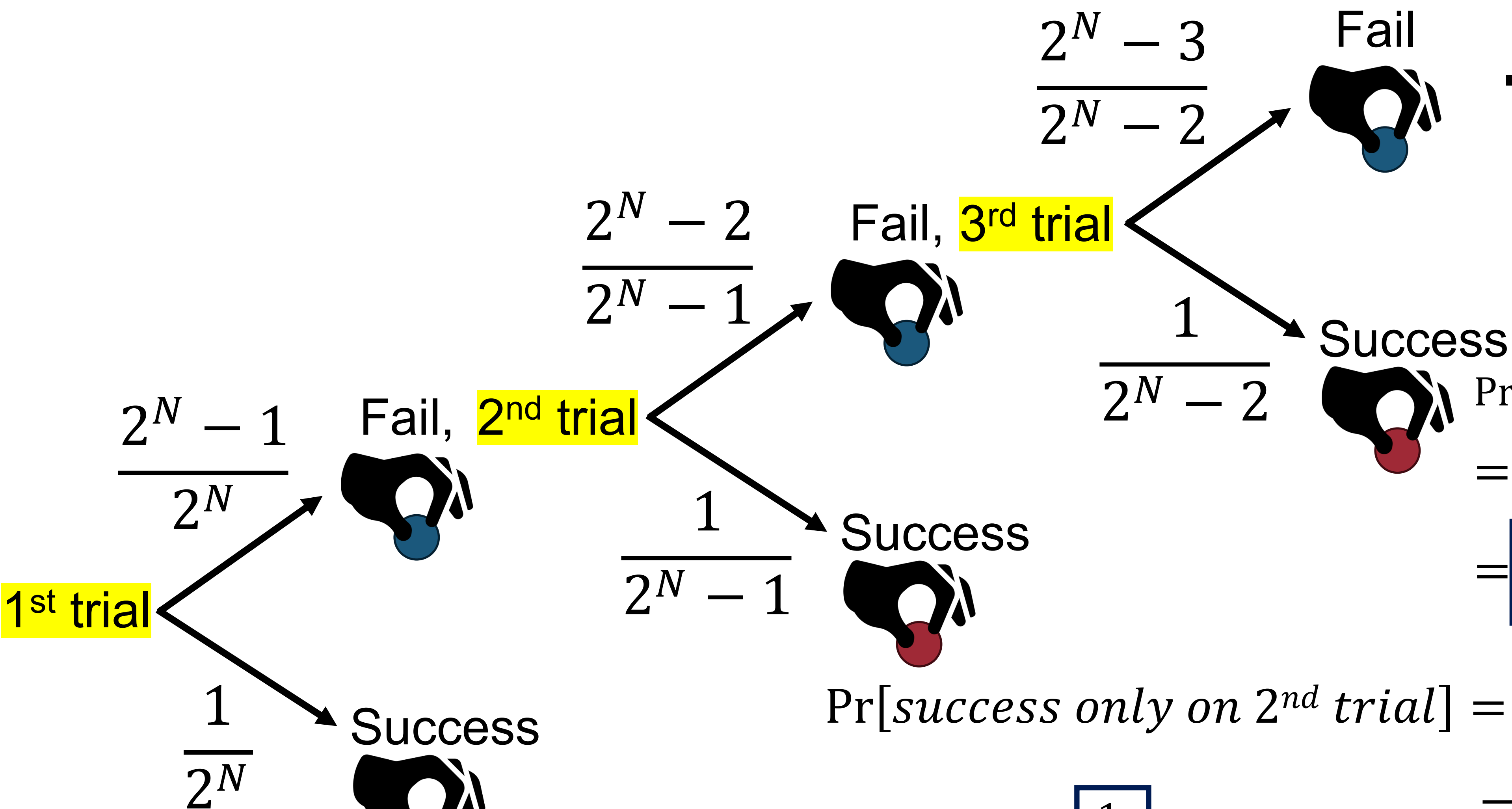


$$\Pr[success\ on\ 1^{st}\ trial] = \frac{1}{2^N}$$

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

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)

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

- **Expected # of trials before success**

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

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**

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

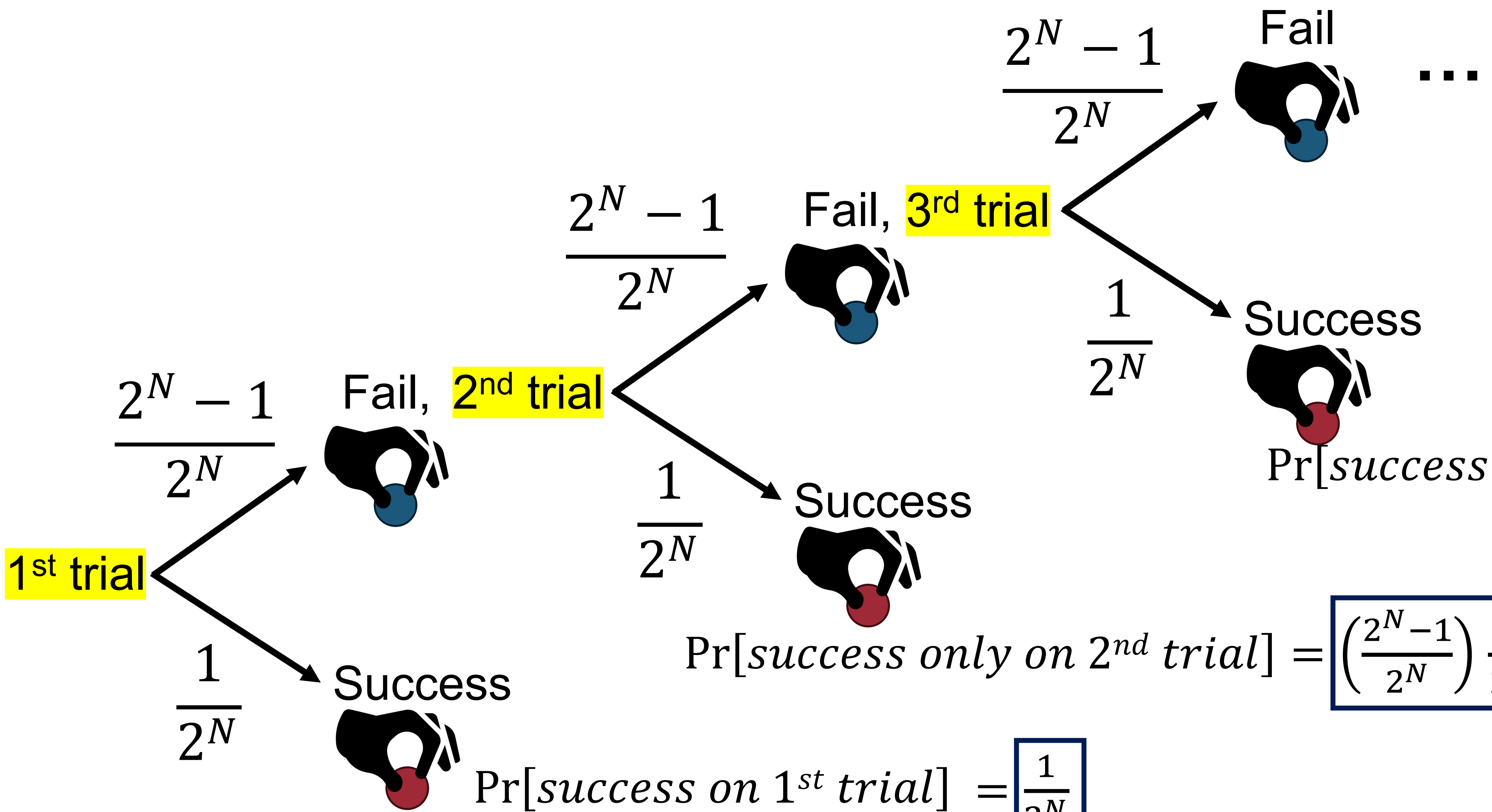
$$= \frac{1}{2^N} \sum_{k=1}^{2^N} k$$

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

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

Case #2: Selecting Balls w/ Replacement

(Linux)



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

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

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

Case #2: Selecting Balls w/ Replacement

(Linux)

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

(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

$$\frac{2^N + 1}{2} \approx 2^{N-1} \quad \text{vs.} \quad 2^N$$

trials on ***Windows***

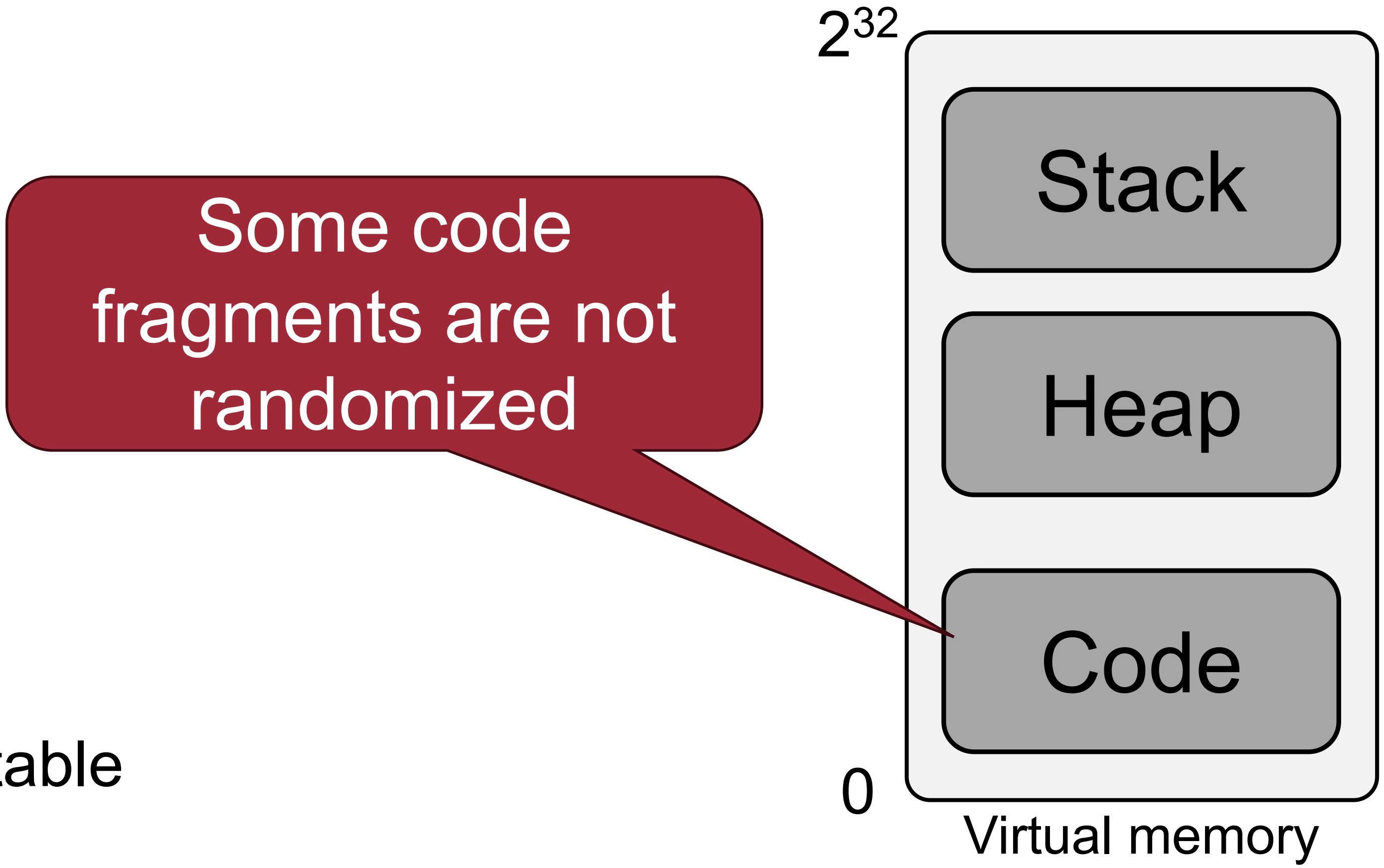
trials on ***Linux***

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

• P… de (P… runs re… e.g. … det… will …

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

- 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

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

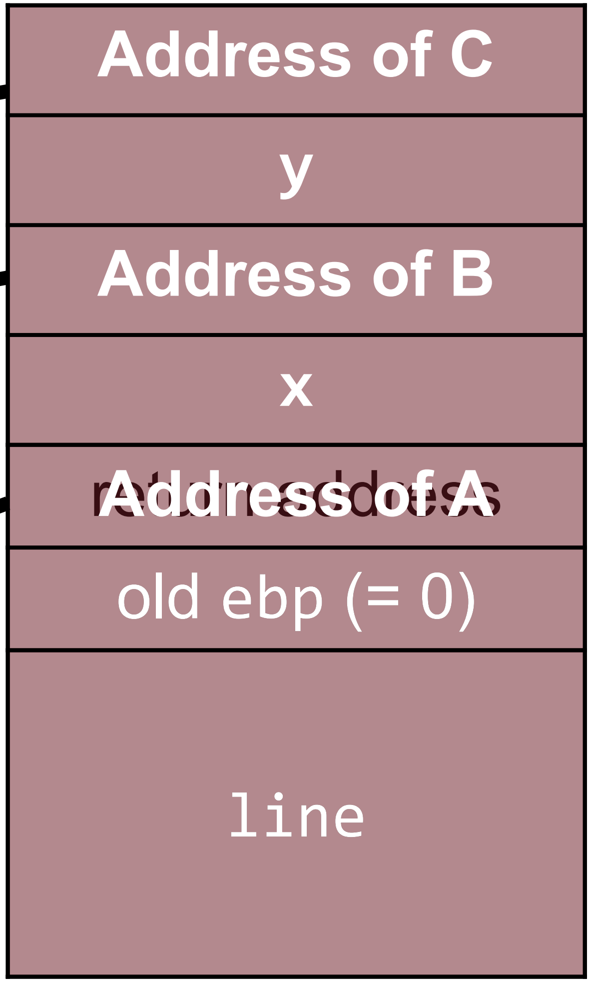
```
jmp [eax]
```

Gadget B

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

Gadget A

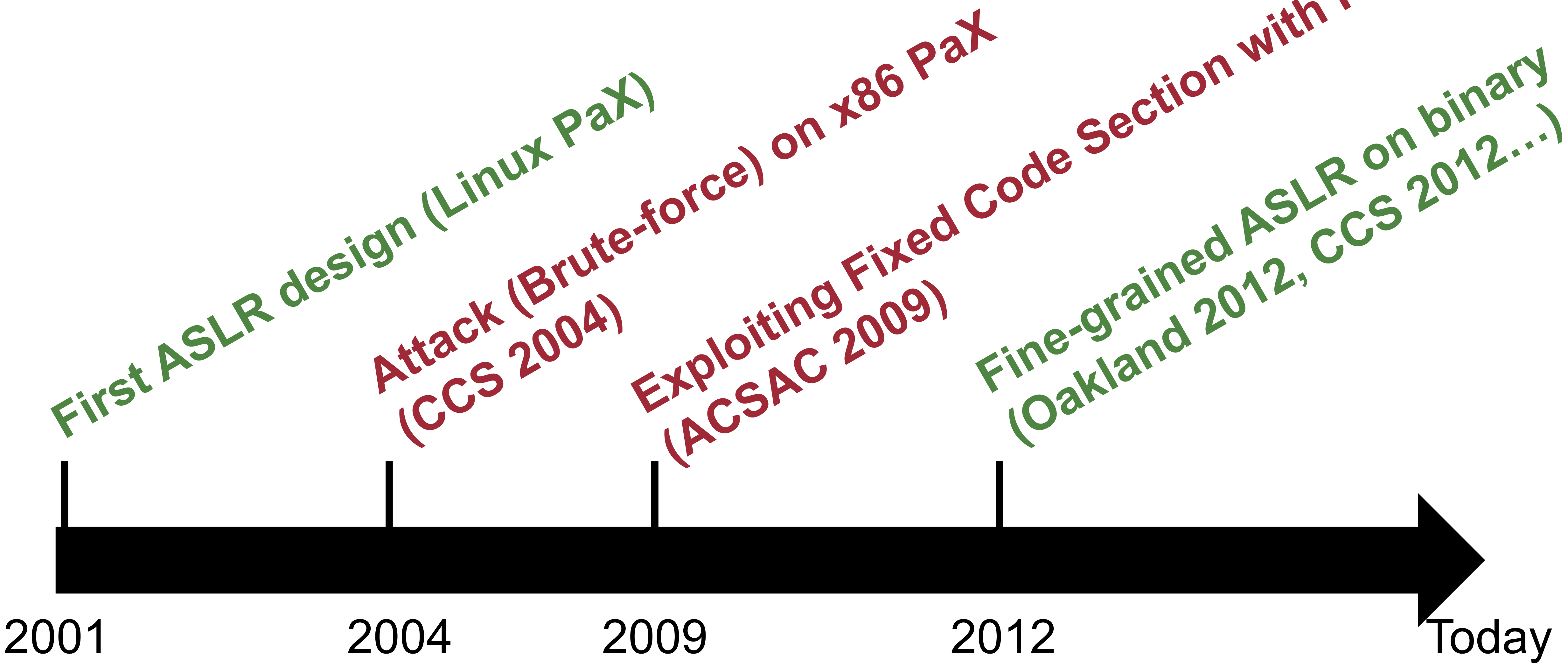
```
pop edi
ret
```



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

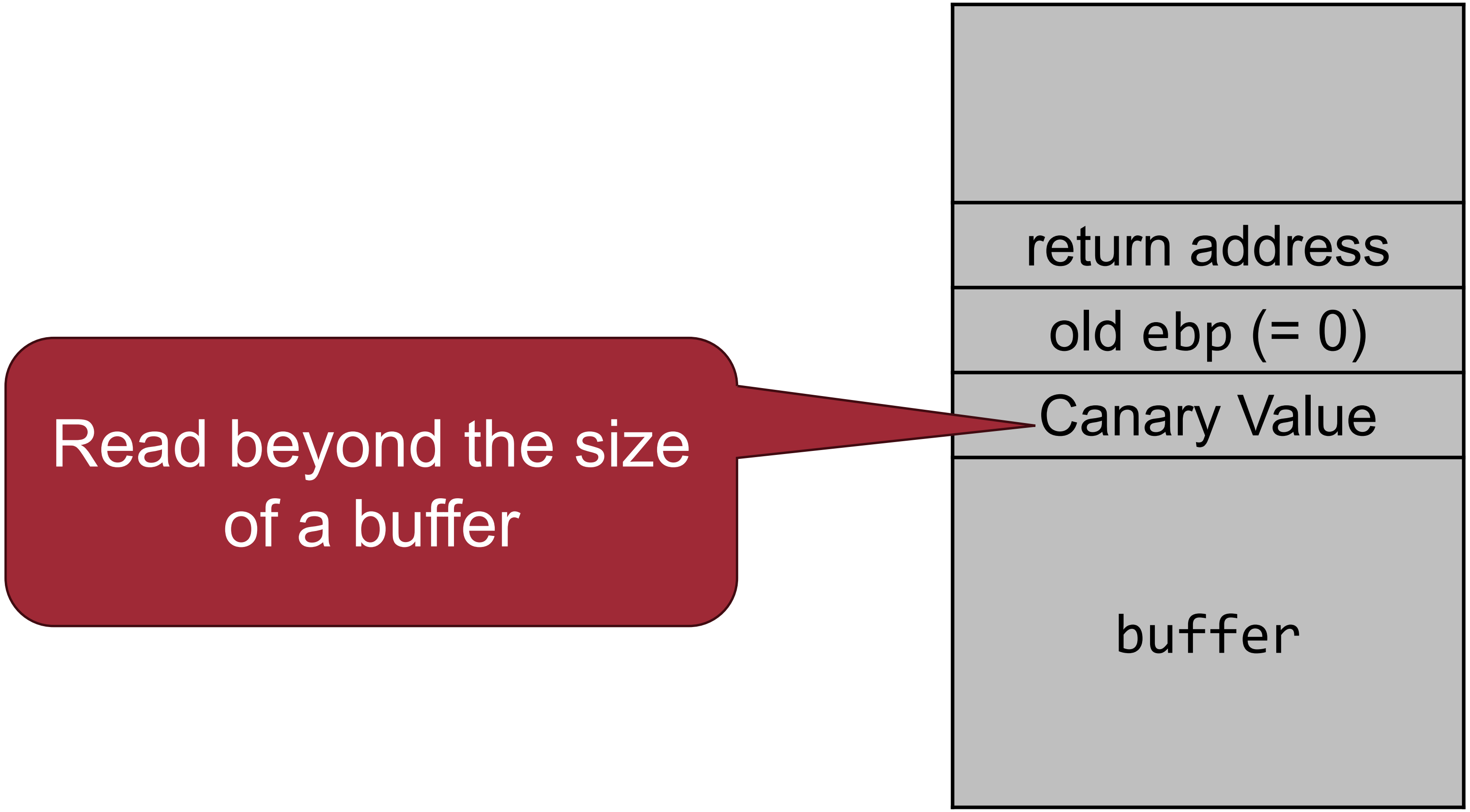
Memory Disclosure

Memory Disclosure $\neq$ Memory Corruption

Memory disclosure does not necessarily involve memory corruption

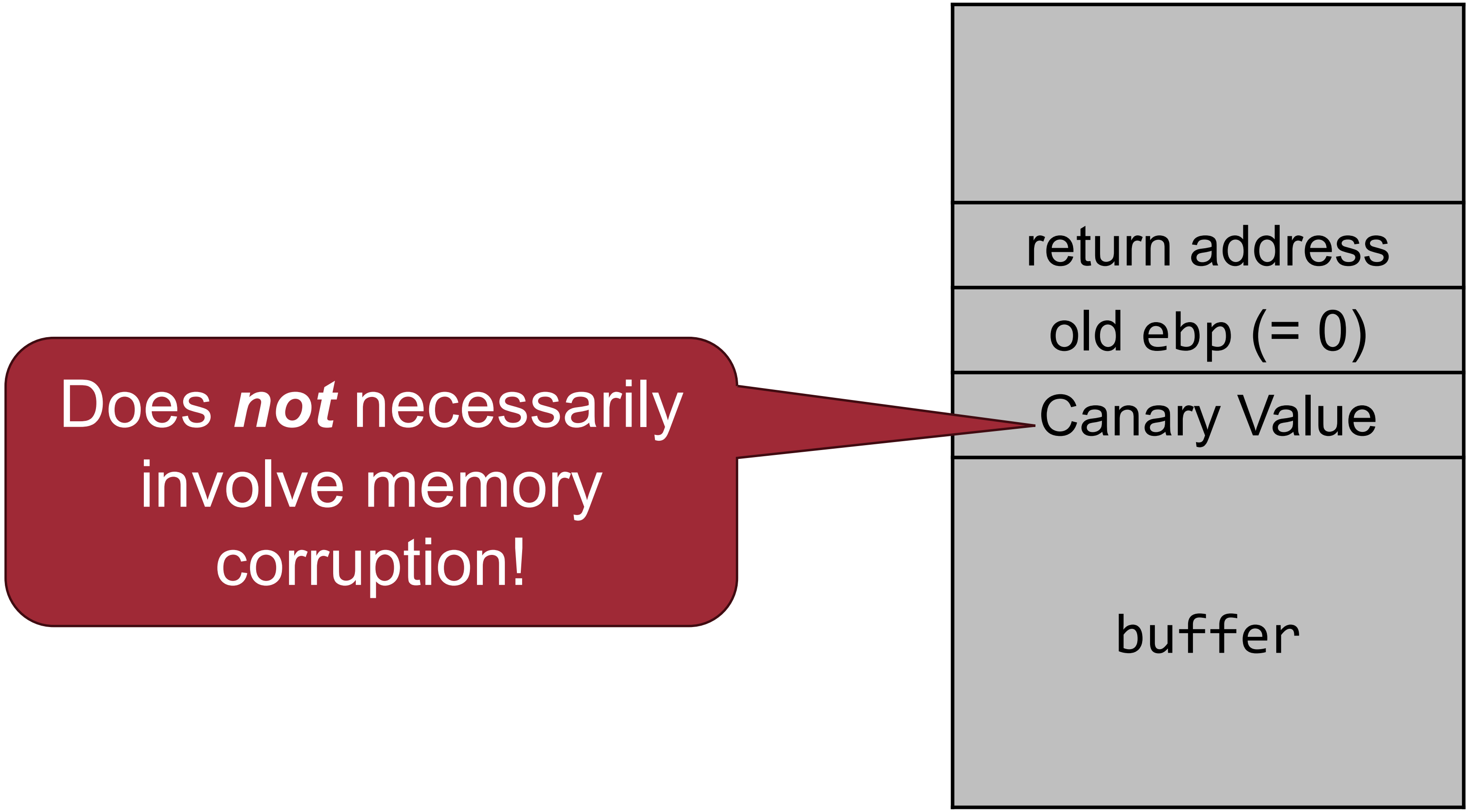
Buffer Over-Read

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



Buffer Over-Read

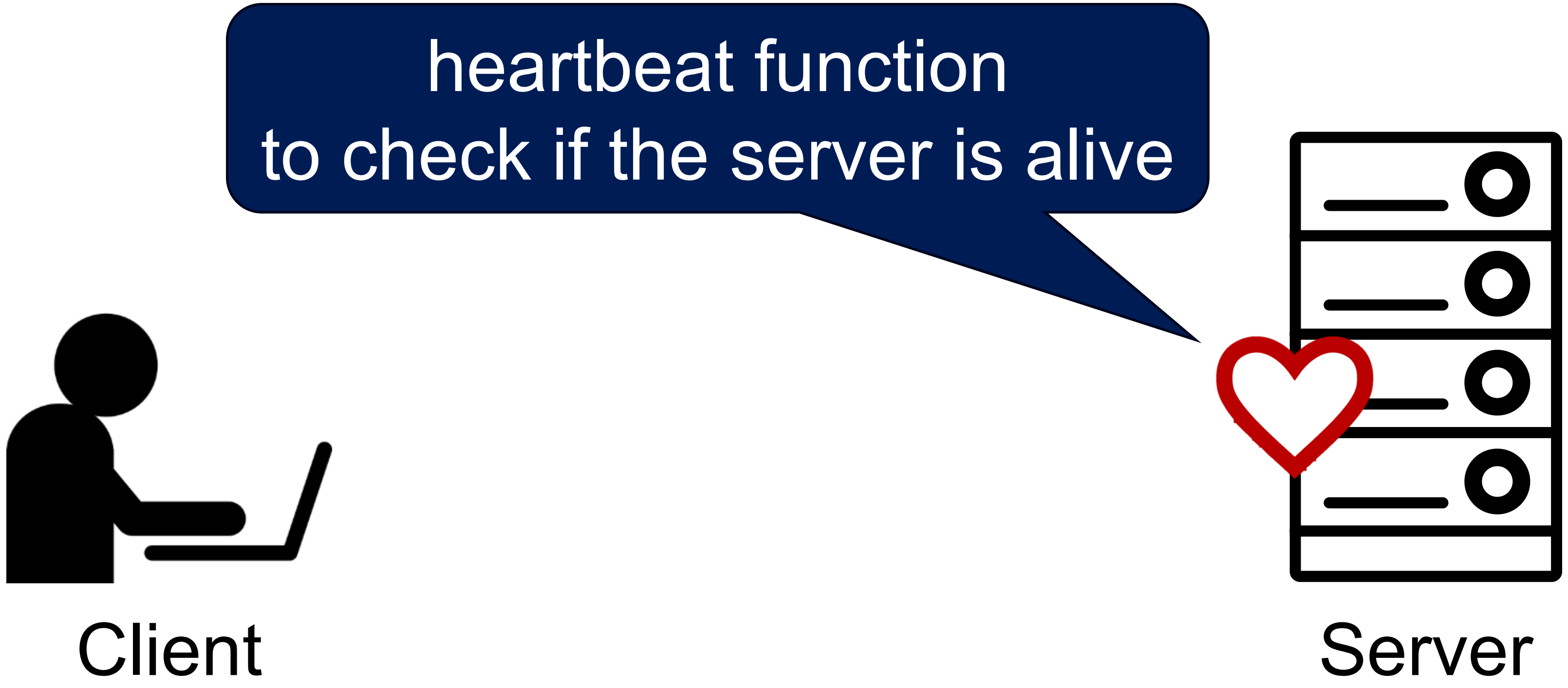
Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer



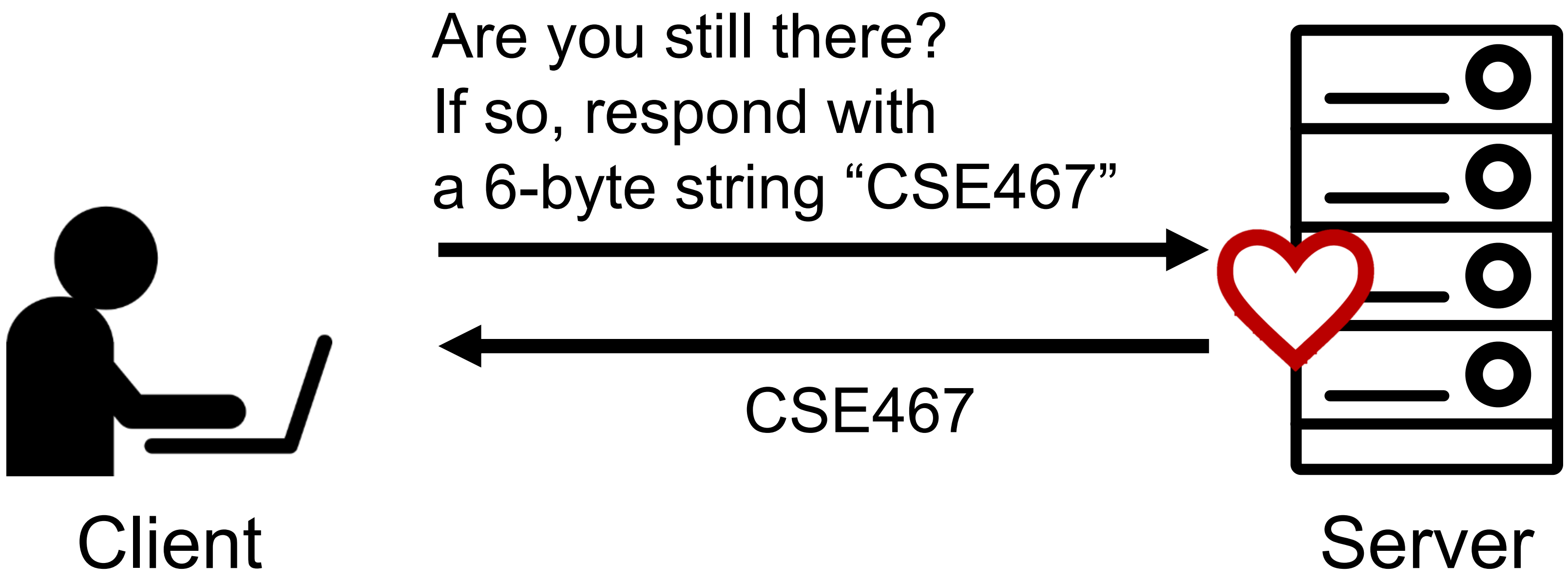
Example: Heartbleed Bug (in 2014)

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

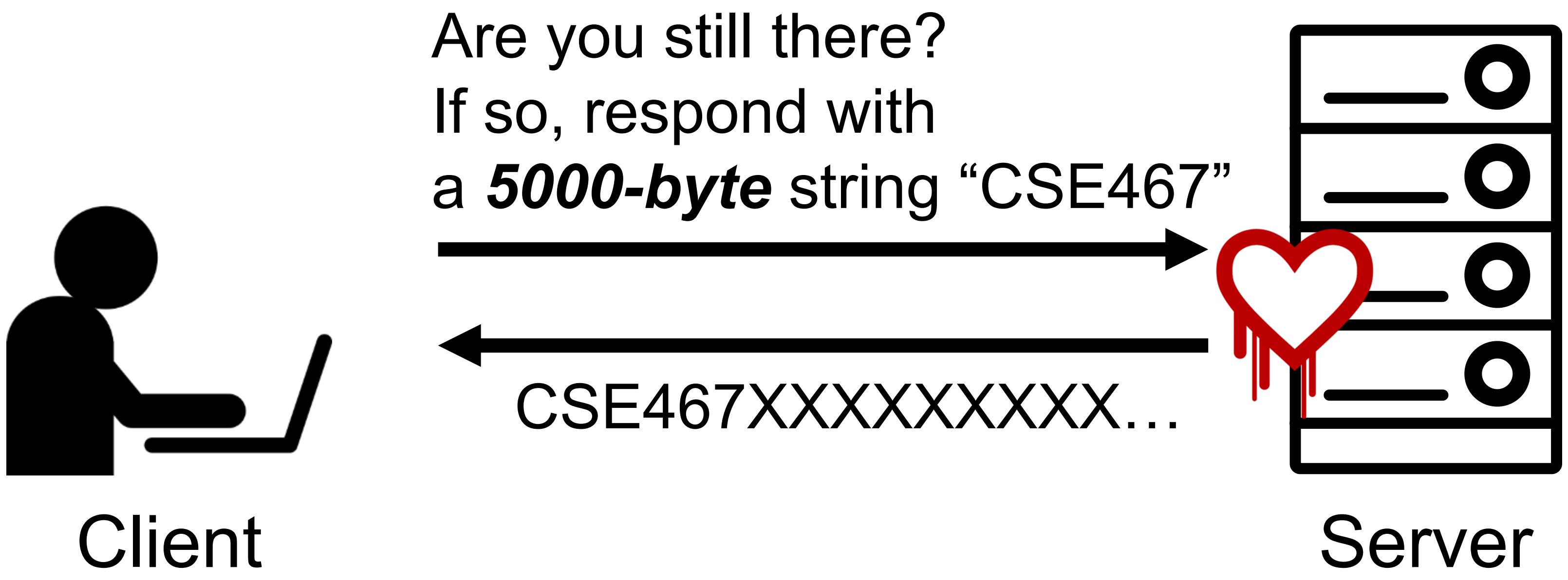
Heartbleed Bug: High-level Workflow



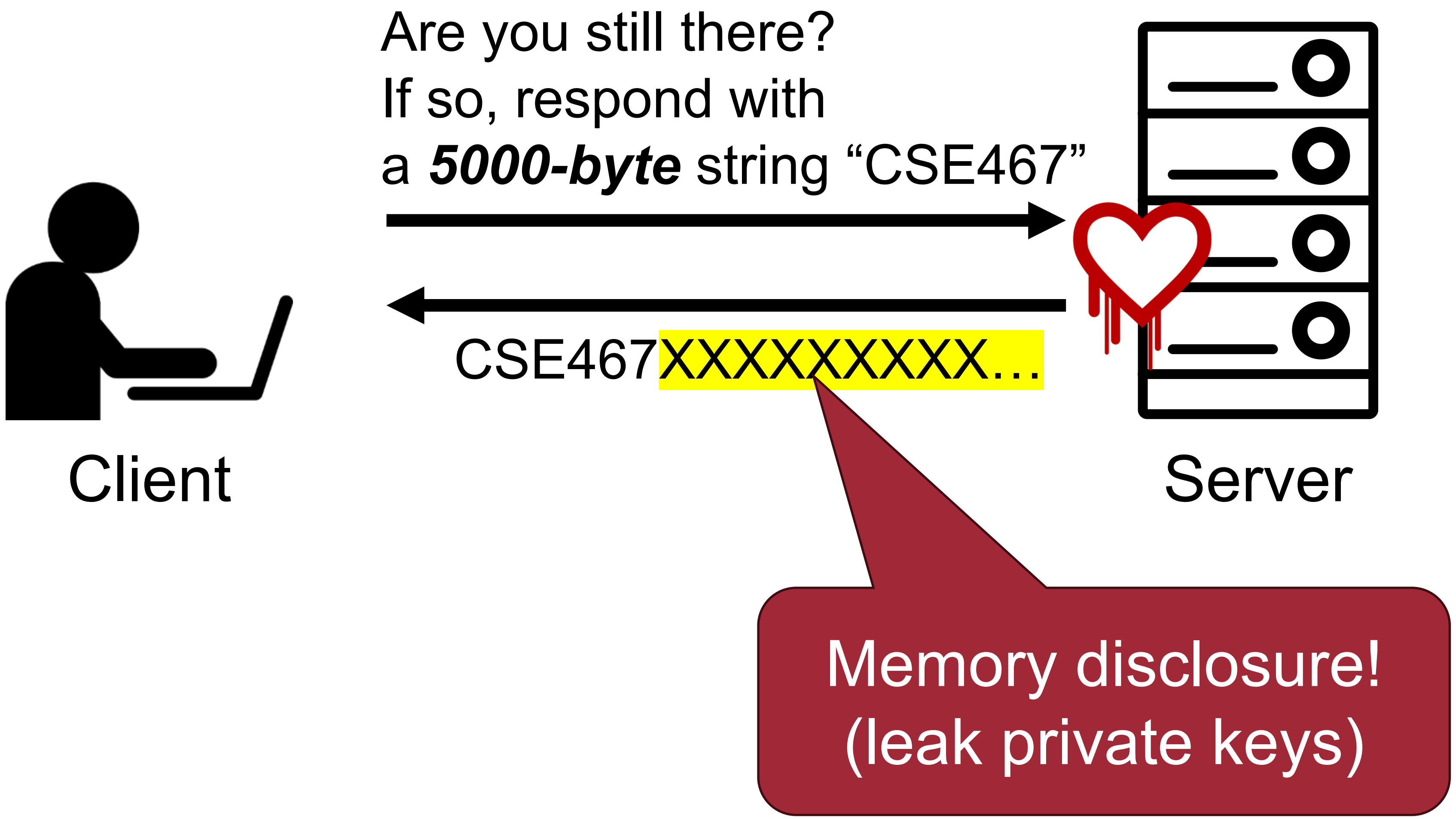
Heartbleed Bug: High-level Workflow



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 paddin
} 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., CSE467)

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

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

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., 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

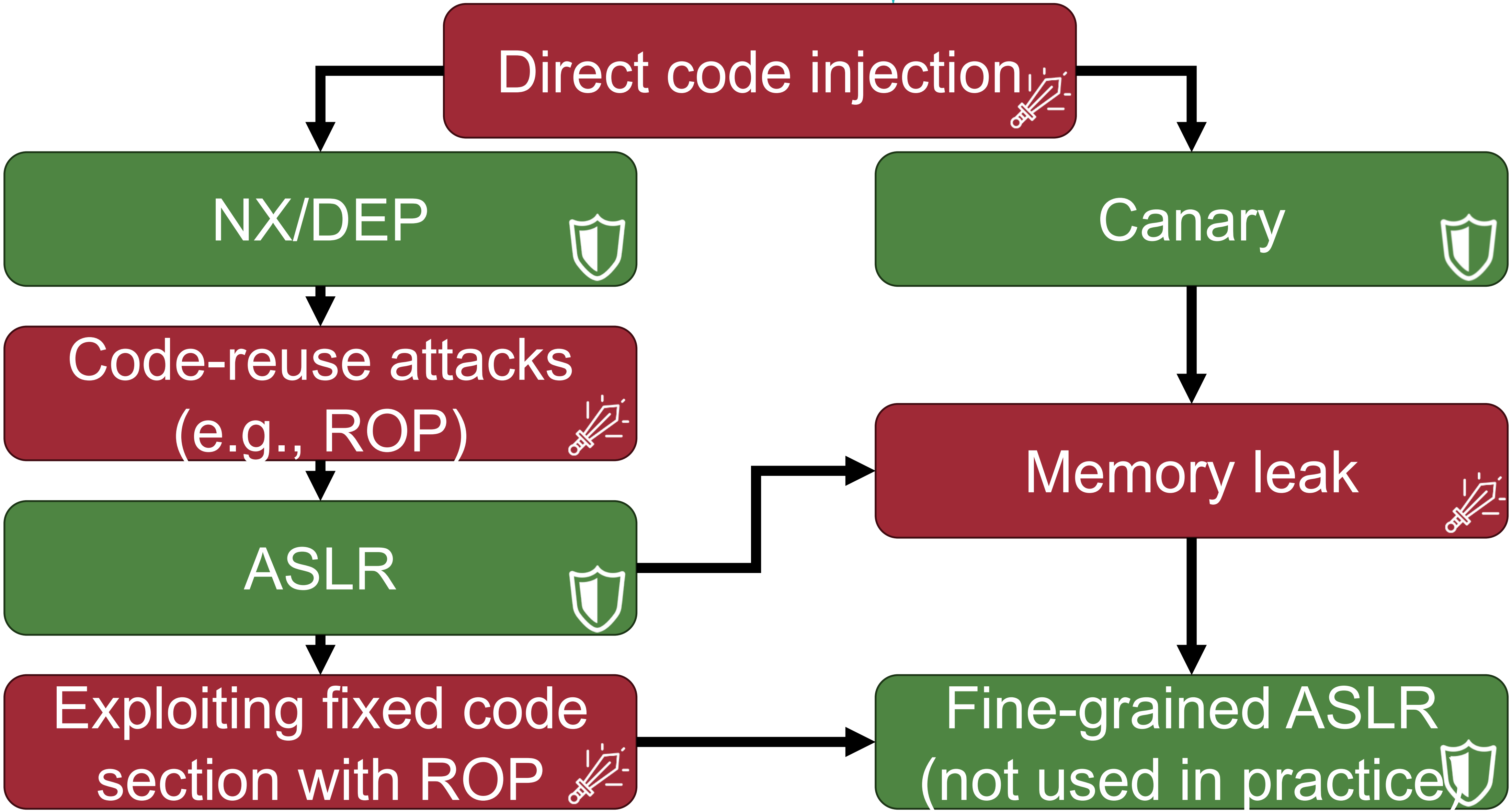
Memory Disclosure and Exploit

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

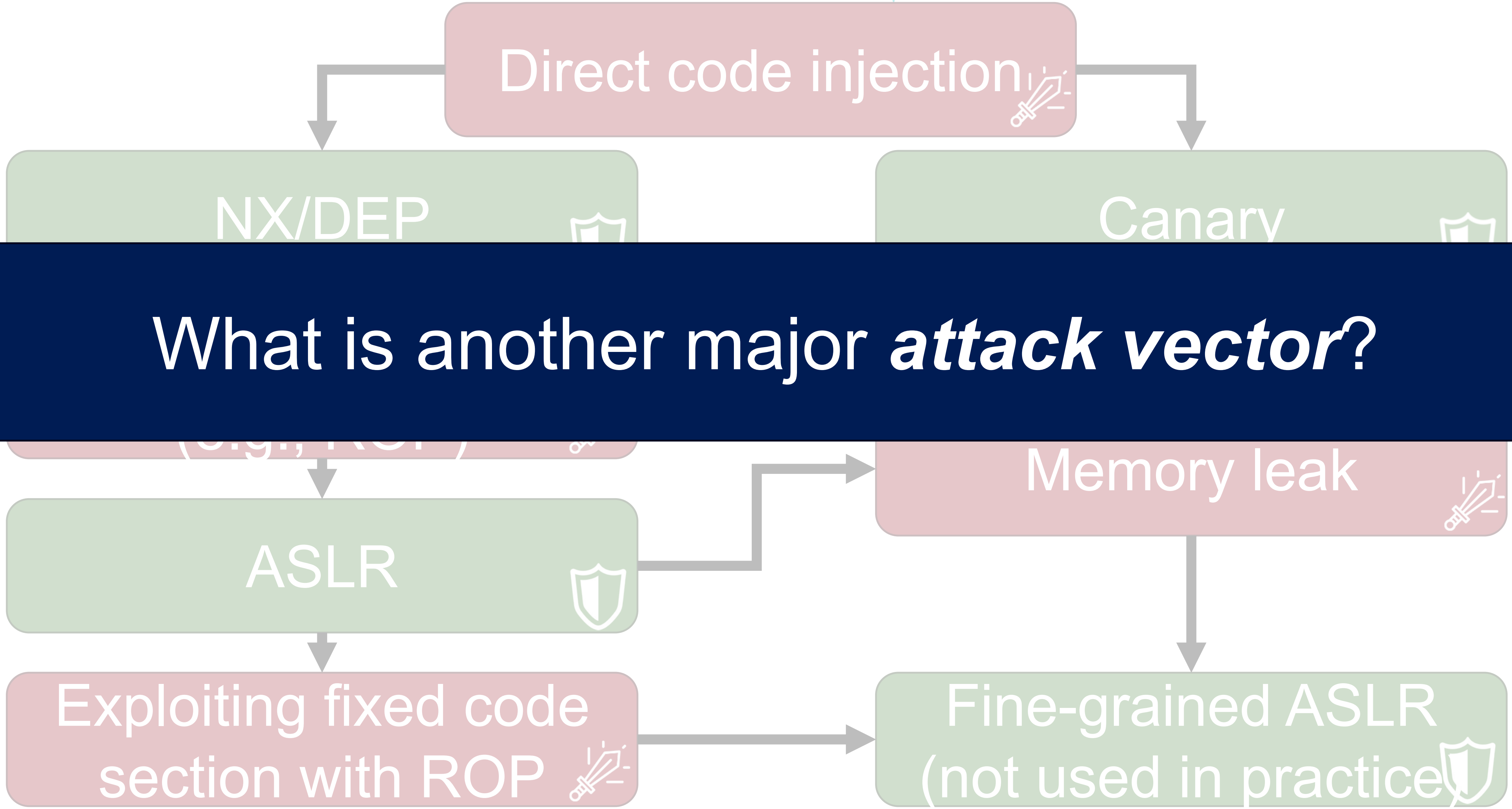
- In such a case, we can bypass existing defenses
  - ***Canary bypass***: canary value could be leaked
  - ***ASLR bypass***: code/stack pointers could be leaked

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

Attack / Defense So Far



Attack / Defense So Far



What is another major ***attack vector***?

Type Confusion

Type Confusion

Normal

Dog class

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Abnormal

Person class

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

Type Confusion

Dog class

Normal

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Type Confusion

Person class

Abnormal

```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

Type Confusion

Normal

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Dog class

Type Confusion

Abnormal

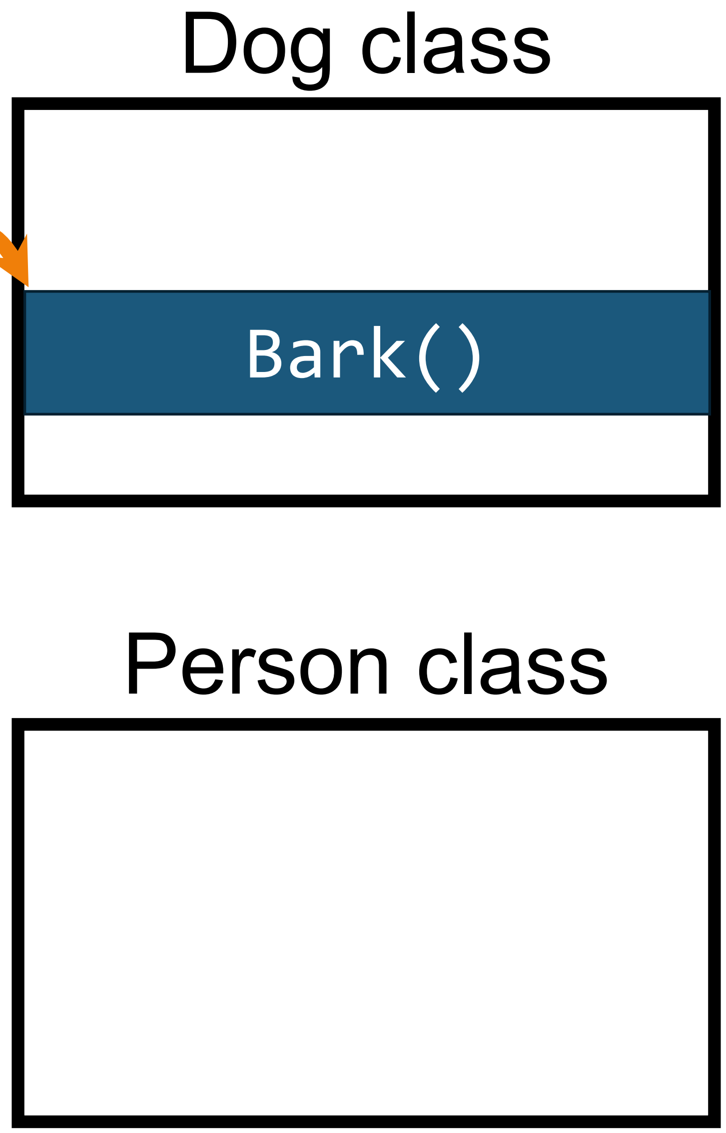
```
Dog *d = (Dog*) some_ptr;
d->bark(); //???
```

Person class

Invoke person's something

Type Confusion Attack (Implication)

```
Dog *d = (Dog*) some_ptr;
d->bark();
```




```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

Type Confusion Attack (Implication)

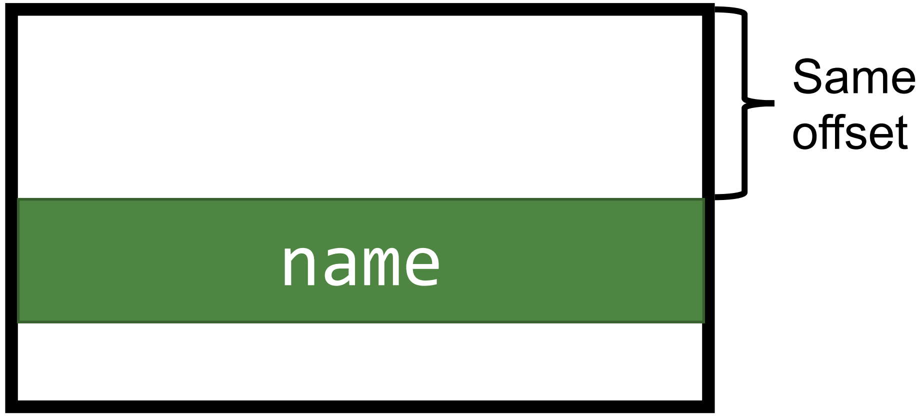
```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Control flow

Dog class


```
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

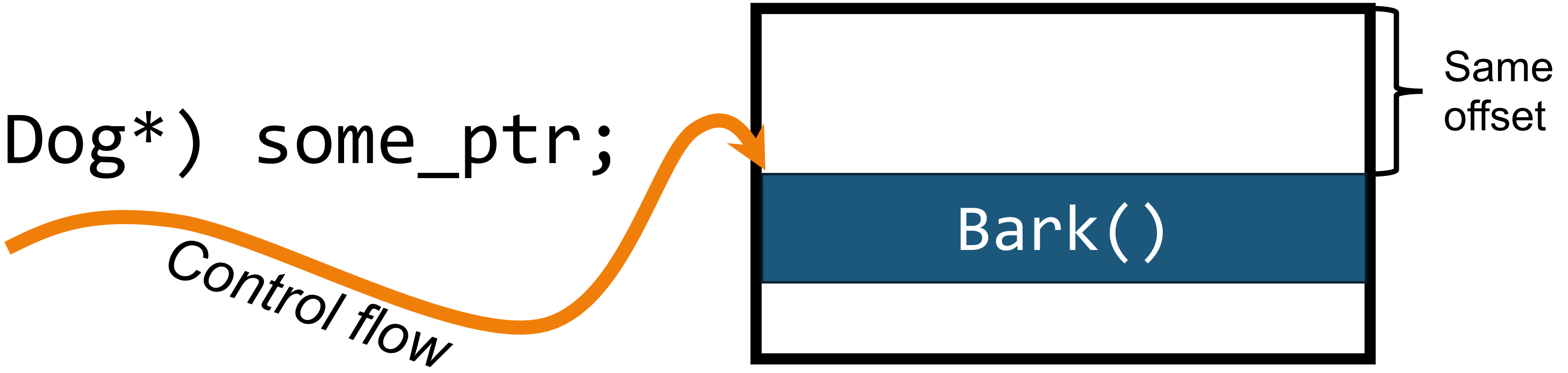
Person class



Type Confusion Attack (Implication)

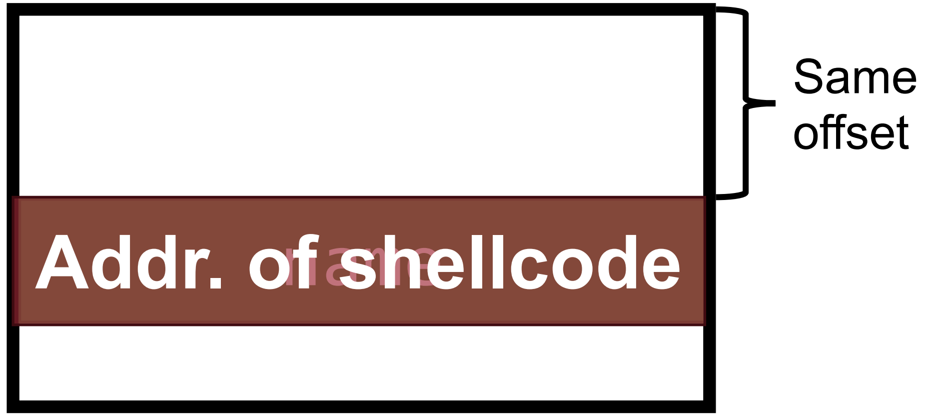
```
Dog *d = (Dog*) some_ptr;
d->bark();
```

Dog class




```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark();  //???
```

Person class



Type Confusion Attack (Implication)

```
Dog *d = (Dog*) some_ptr;
d->bark();
```

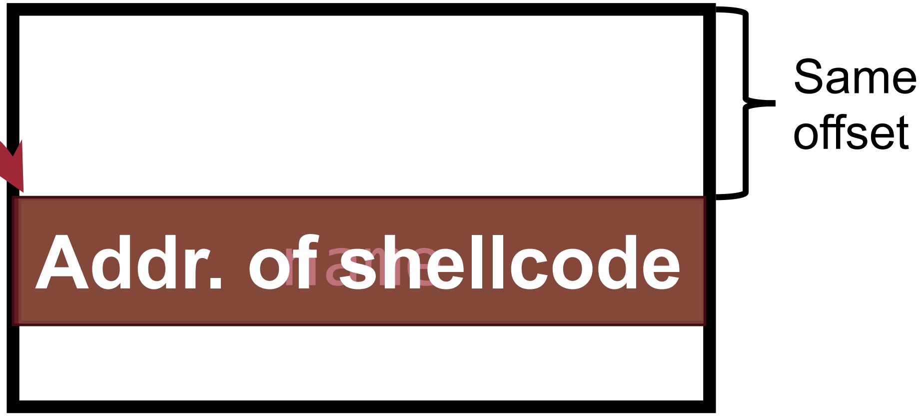
Control flow

Dog class


```
some_ptr->name="[shellcode]"
…
Dog *d = (Dog*) some_ptr;
d->bark(); //???
```

Control flow

Person class



Type Confusion Example: Downcasting

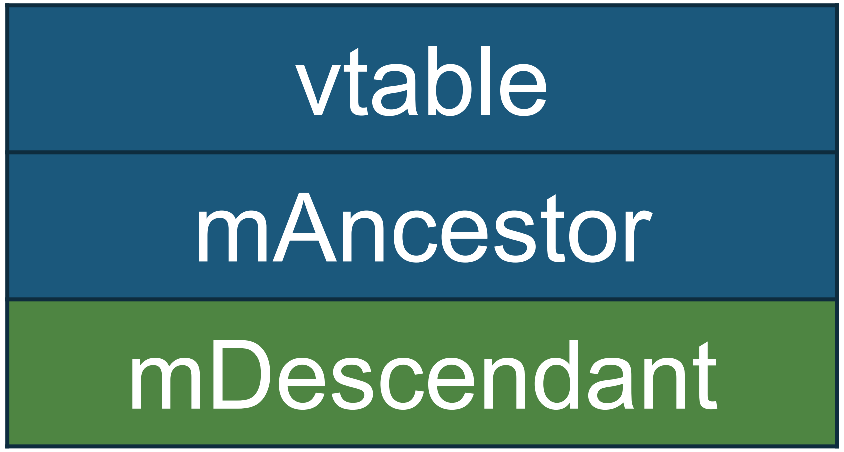
```
class Ancestor {
    public:
        int mAncestor;
    ...
};

class Descendant: public Ancestor {
    public:
        int mDescendant;
    ...
};
```

Inherit
Ancestor class

Type Confusion Example: Downcasting

```
class Ancestor {
    public:
        int mAncestor;
    ...
};
```


```
class Descendant: public Ancestor {
    public:
        int mDescendant;
    ...
};
```



Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

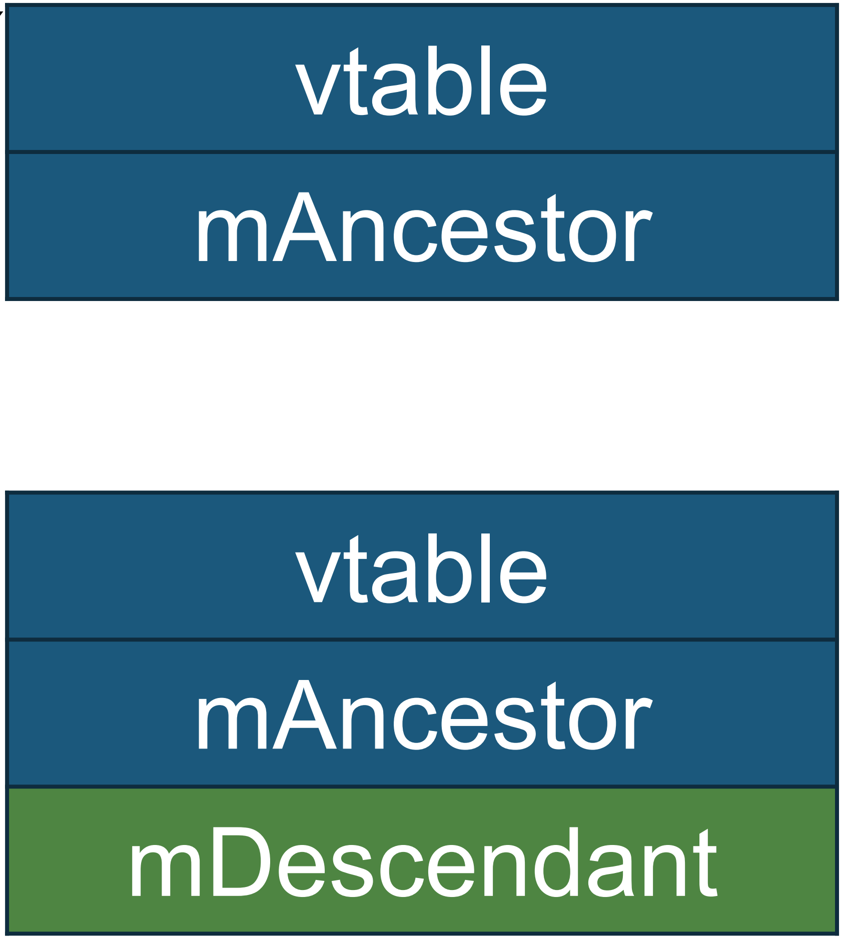
Type Confusion Example: Downcasting

```
class Ancestor {
    public:
        int mAncestor;
    ...
};

class Descendant: public Ancestor {
    public:
           scendant;

};
```

Downcasted pointer

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```



Type Confusion Example: Downcasting

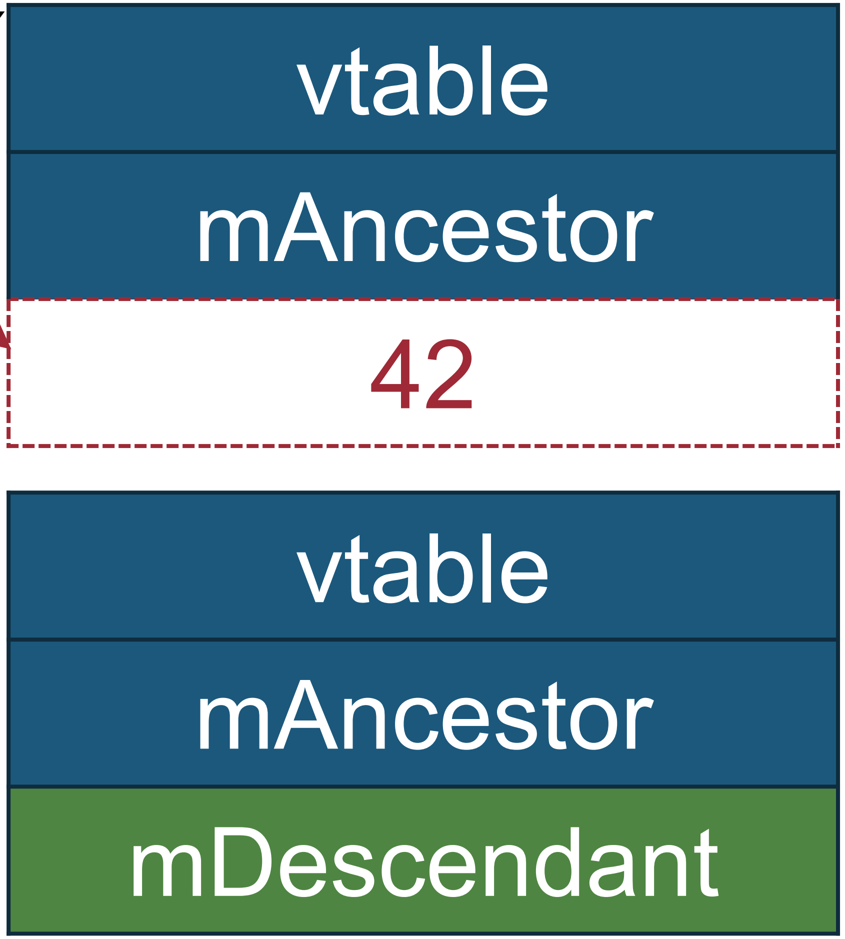

```
class Ancestor {
    public:

};

class Descendant: public Ancestor {
    public:
        int mDescendant;

};
```

Memory corruption:
It can now access a memory region that was not allocated!

Downcasted pointer

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

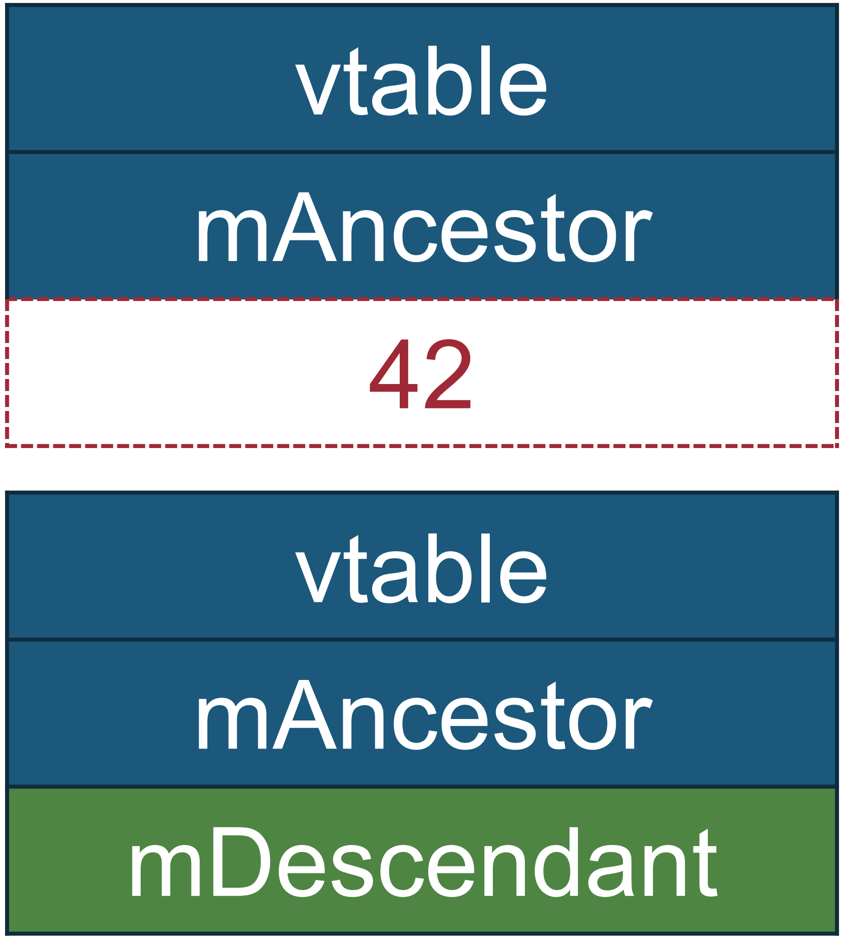


Downcasting is a Common Practice

Suppose these two lines are far away
(e.g., separated in two different libraries)

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

Implication of the Downcasting

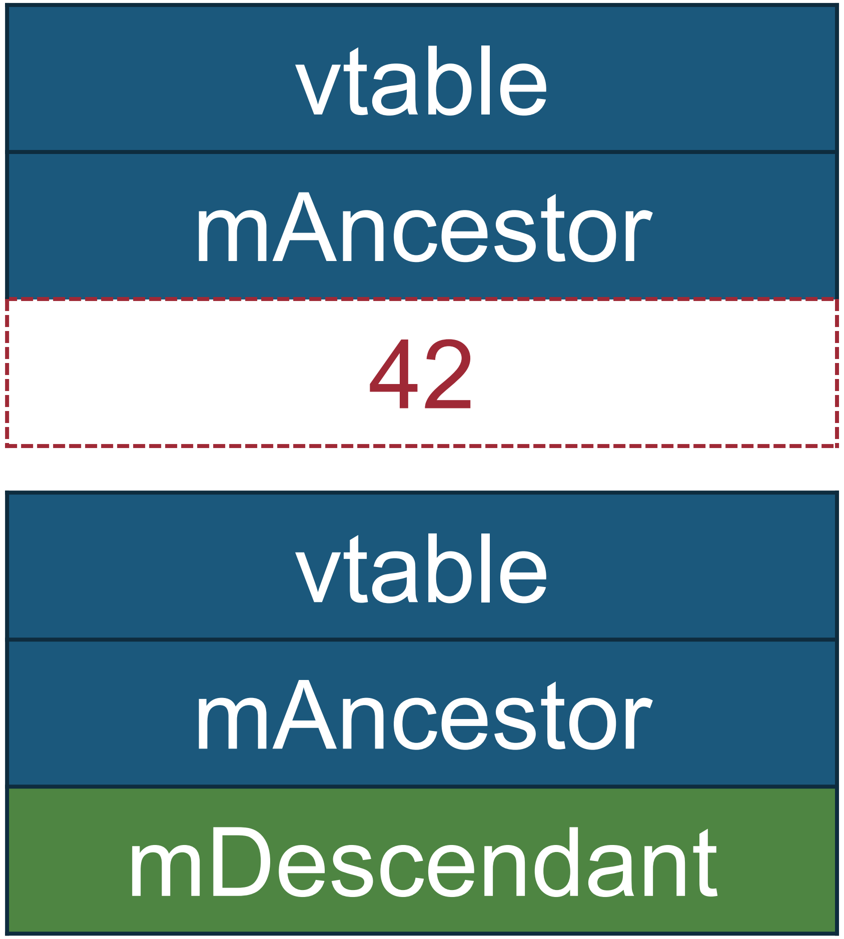


What if a user can write an arbitrary value to the confused pointer?

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

Implication of the Downcasting



Unlike other attack vectors, we can ***reliably*** corrupt a certain memory field, *i.e.*, we don't need to know the actual address of `mDescendant`

Vulnerable code

```
Ancestor* a = new Ancestor();
Descendant* d = static_cast<Descendant*>(a);
d->mDescendant = 42;
```

Patch: Use `dynamic_cast`

Limitations:

- Slow
- Compiler options such as `--fno-rtti` can disable it!

Use After Free (UAF)

- A popular source of type confusion (next lecture)

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 ($\neq$ Memory Corruption)
- Type confusion
- Security vs. Performance

Question?