

CSE467: Computer Security

9. Canary & DEP & ROP

Seongil Wi

Quiz #1



- Score will be released soon!

HW2 Announcement



The due date has been extended!

- Due: Oct. 24, 11:59PM => **Oct. 26, 11:59PM**
 - (There was a VPN issue during Chuseok, so you may have had trouble accessing it ☹)

HW2 Announcement



- Software security
 - Hacking practice: Capture the Flag (CTF)
- CTF server: <http://10.20.12.187:4000/>
 - This server can only be accessed from the UNIST internal network.
 - Please use a *VPN* to access from outside! Just log in to <https://vpn.unist.ac.kr> and turn on VPN.
- Each flag is in the following format: `flag{some_string}`
 - e.g., `flag{C0N9R@7u1aT1on!}`

Setting up a vagrant on M1/M2 MacBooks



- The solution we strongly recommend is to dockerize the environment
1. Install docker
 2. `$ mkdir YOUR_PATH; cd YOUR_PATH`
 3. Download our Dockerfile (<https://websec-lab.github.io/courses/2023f-cse467/hw/Dockerfile>) to YOUR_PATH
 4. `$ docker build --tag cse467 .`
 5. `$ docker run -it -v <ABSOLUTE_PATH_HOST_DIR>:/data --name cse467_container cse467 /bin/bash`

Setting up a vagrant on M1/M2 MacBooks



- **If you want to stop your container:** press `ctrl+d`
- **If you want to resume your container:**
 1. `$ docker start cse467_container`
 2. `$ docker attach cse467_container`

Recap: Format String Vulnerability

```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

- `buf = "Hello"` // No problem
- `buf = "%d.%d.%d\n"` // Leak memory

Recap: Format String Vulnerability



- Format string vulnerability allows us to ***read arbitrary memory*** contents on the stack

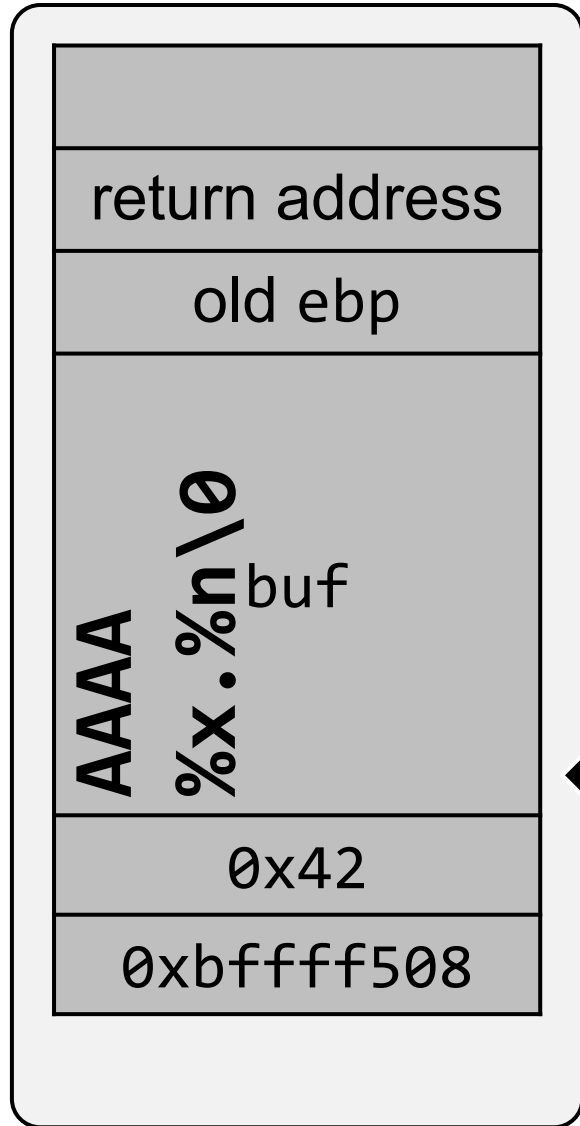
What about ***arbitrary memory write?***

Recap: Format String Vulnerability

Format	Meaning
%d	Decimal output
%x	Hexadecimal output
%u	Unsigned decimal output
%s	String output
%n	# of bytes written so far

Nothing printed for %n

Recap: Format String Vulnerability



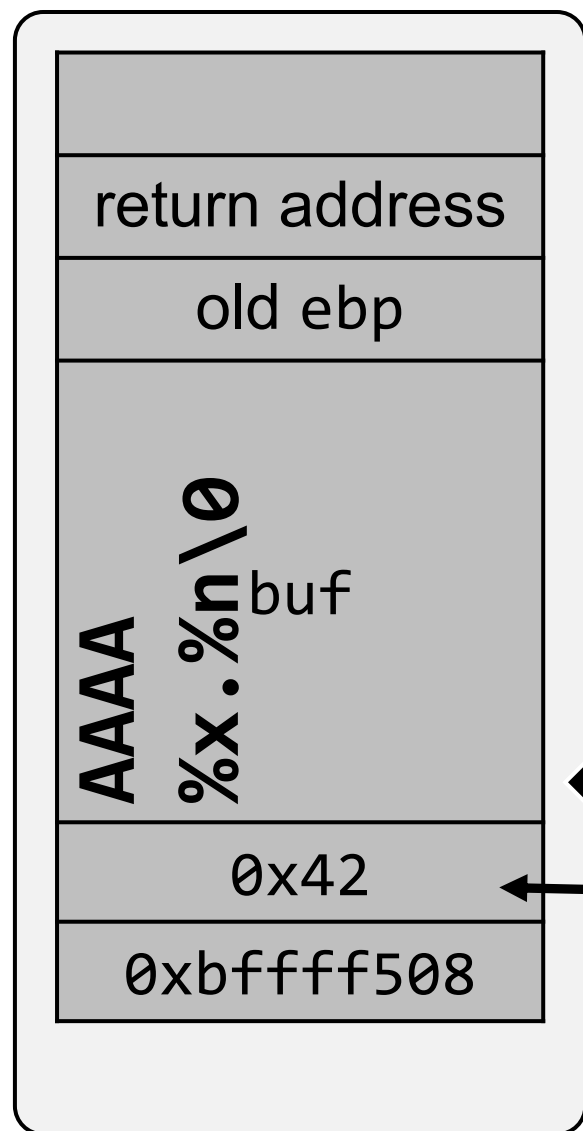
```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

buf = "AAAA%x.%n"

Printed value:
AAAA

Recap: Format String Vulnerability



```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

buf = "AAAA%x.%n"

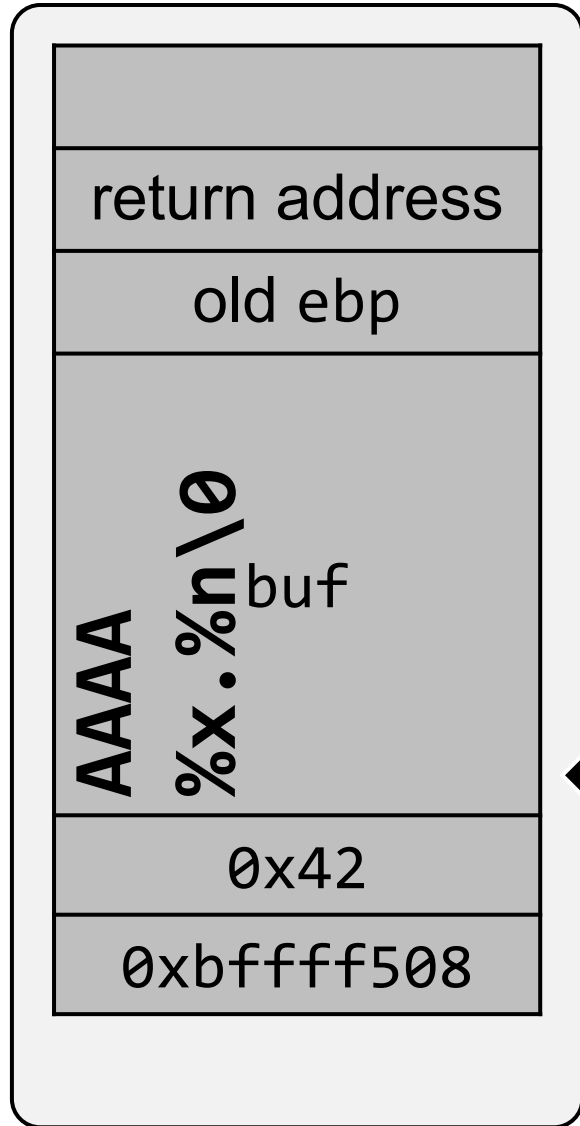
0xbffff508

Second
parameter!

Printed value:
AAAA42

Virtual memory

Recap: Format String Vulnerability



Virtual memory

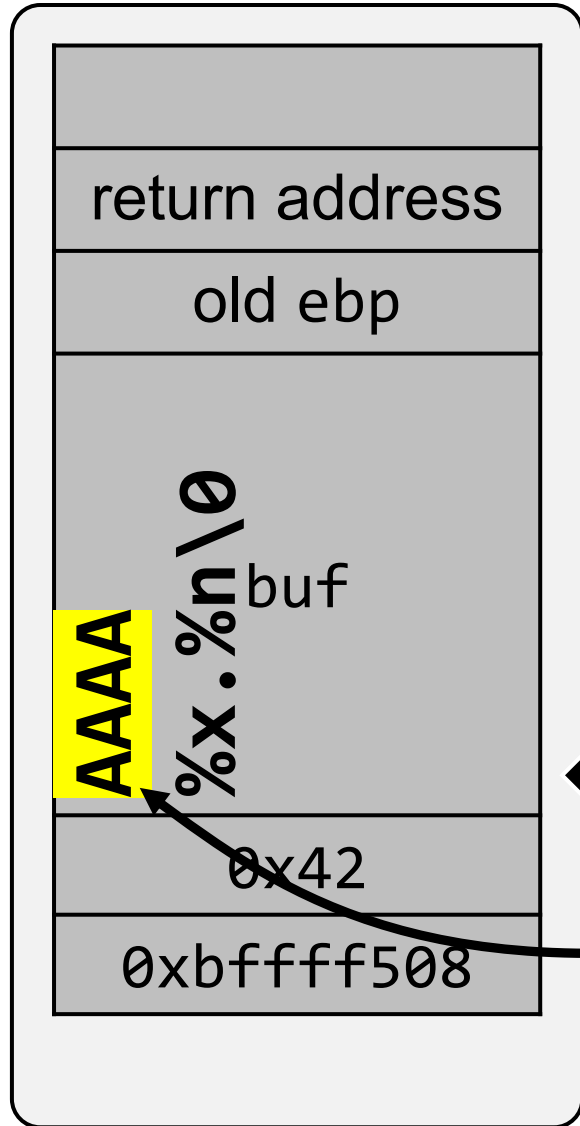
```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

buf = "AAAA%x.%n"

Printed value:
AAAA42.

Recap: Format String Vulnerability



```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

buf = "AAAA%x.%n"

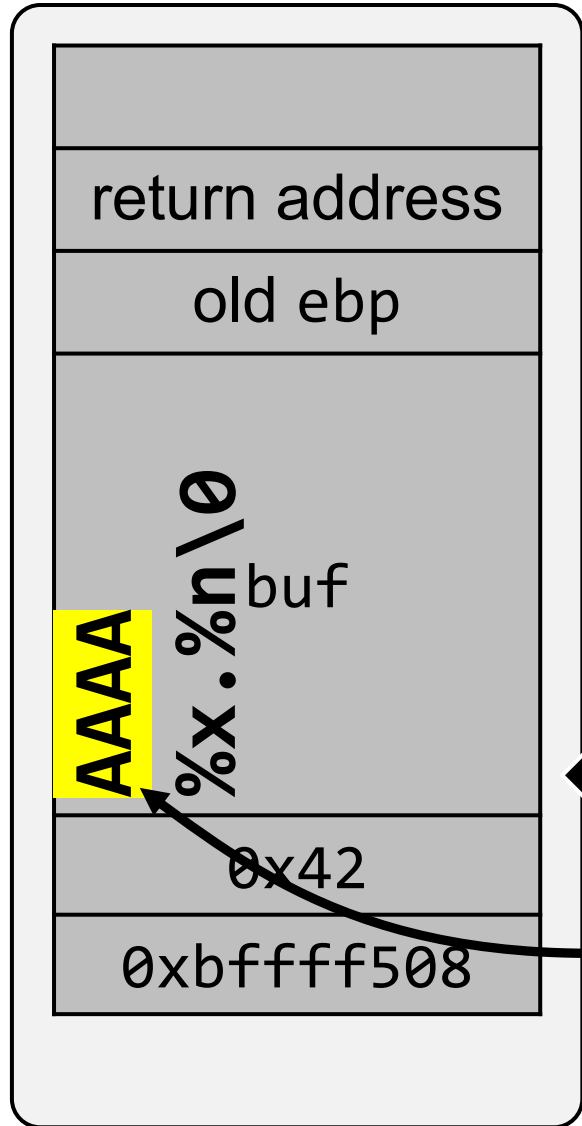
Third parameter!

Printed value:
AAAA42.

Write ? to the
address 0x41414141

Virtual memory

Recap: Format String Vulnerability



```
// ...
```

```
recv(sock, buf, sizeof(buf), 0);  
printf(buf);
```

buf = "AAAA%x.%n"

Third parameter!

Printed value:

AAAA42.
7

Write 7 to the
address 0x41414141

Recap: Optimization with Dollar Sign (\$)

15

- Enables direct access to the n -th parameter
- Syntax: `%<n>$<format specifier>`

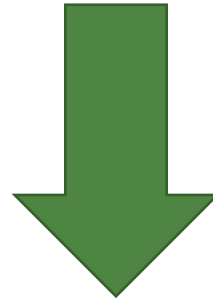
- Example

```
printf(“%d, %d, %d, %2$d\n”, 1, 2, 3);  
// prints 1, 2, 3, 2
```

Recap: Optimization with Dollar Sign (\$)

16

```
$ echo "AAAABBBBBBAAAADBBBB%8722d%hn%58850d%hn" | ./fmt
```



```
$ echo "BBBBDBBB%8730d%1$hn%58850d%2$hn" | ./fmt
```


Recap: Integer Overflow

- Happens because the size of registers is fixed

Logically,

$$0xffffffff + 1 = 0x100000000$$

But, in reality, on x86,

$$0xffffffff + 1 = 0$$

Recap: Why Integer Overflows Matter?

18

- Usually, an integer overflow itself does not lead to control flow hijack exploits
- However, integer overflows can cause an ***unexpected buffer overflows***

Recap: Example



```
int catvars(char *buf1, char *buf2, unsigned len1, unsigned len2)
{
    char mybuf[256];
    if((len1 + len2) > 256) {
        return -1;
    }
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

Recap: Example

What if `len1=0x104` and
`len2=0xfffffffffc`?

```
int catvars(char *buf1, char *buf2, unsigned len1, unsigned len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
        return -1;
    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2);

    do_some_stuff(mybuf);

    return 0;
}
```

Len1=0x104 (=260)
→ Overflow already!

Prevention vs. Mitigation



- Preventing buffer overflows
 - Buffer overflows will never happen
- Mitigating buffer overflows
 - Buffer overflows will happen, but will be ***hard to exploit them***

How to Prevent Buffer Overflows?

22

Do NOT use C/C++!

C is the root of evil!

Easy to Prevent Buffer Overflows!



Have you ever seen buffer overflows in other safe languages such as F#, OCaml, Haskell, Python, etc.?

```
>>> x = array('l', [1,2,3])
```

```
>>> x[4]
```

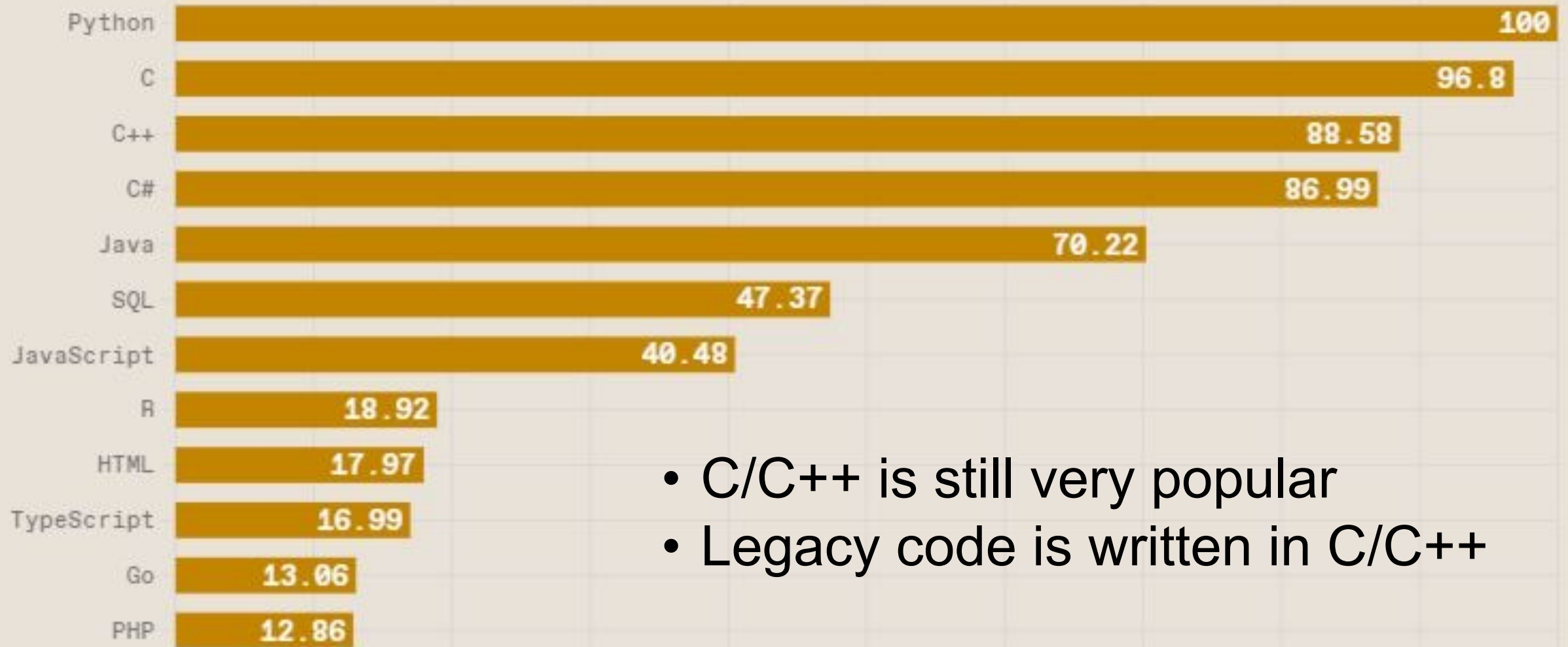
```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: array index out of range
```

Unfortunately though ...

Top Programming Languages 2022



- C/C++ is still very popular
- Legacy code is written in C/C++

Okay ...

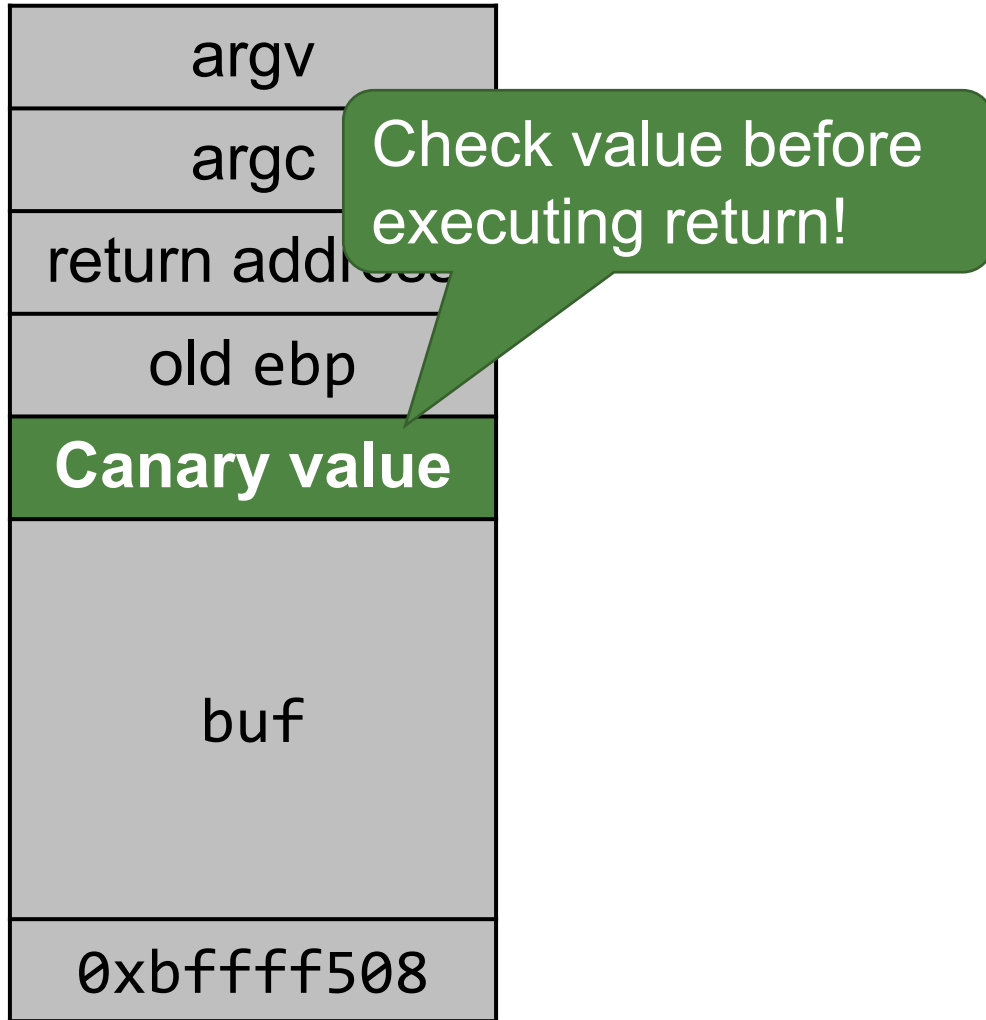


25

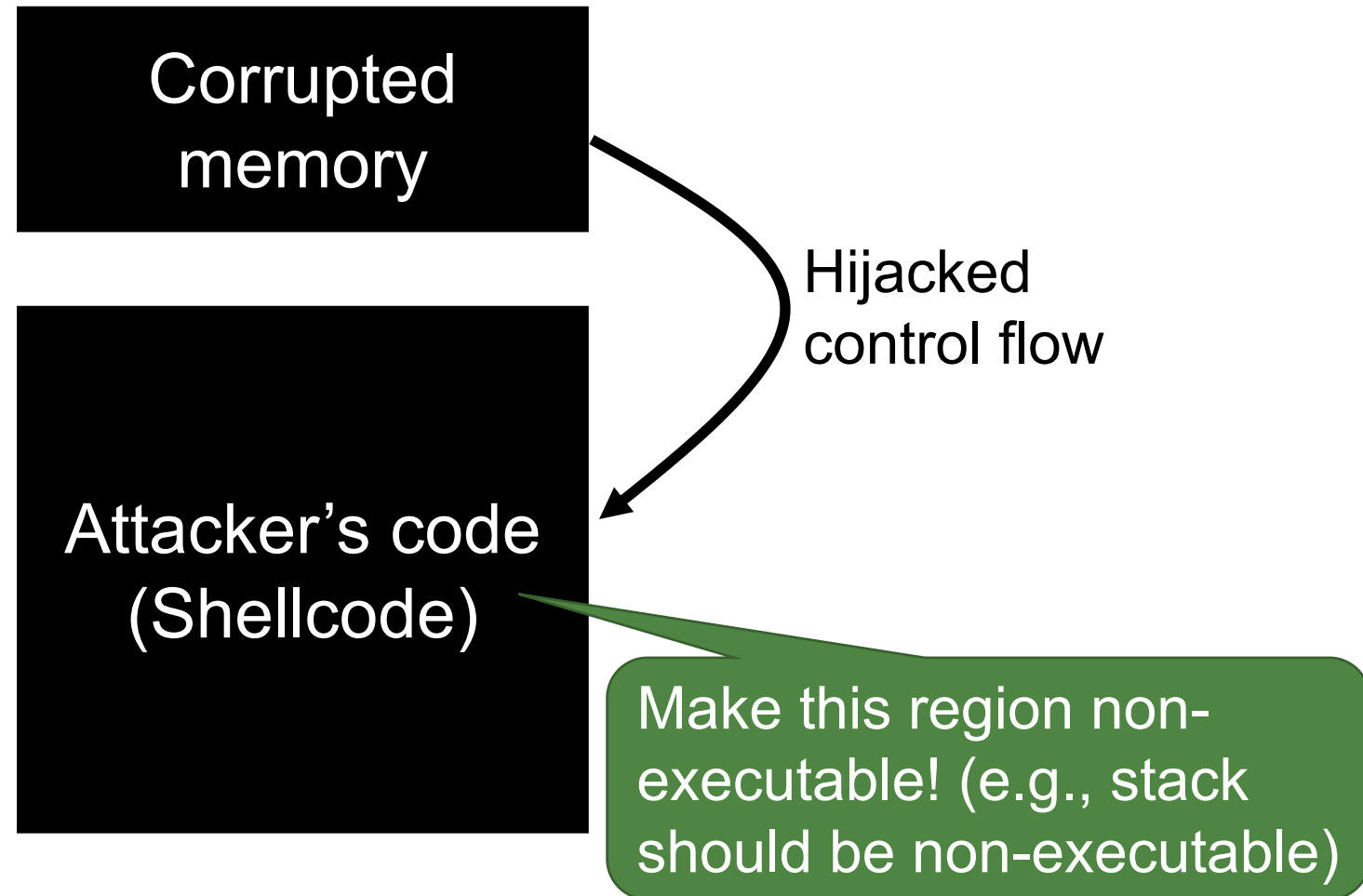
Let's mitigate it then ☹️

Preview: Mitigating Memory Corruption Bugs ²⁶

Mitigation #1: Canary



Mitigation #2: NX (No eXcute)



Buffer Overflow Mitigation #1: Canary

is a bird



Canary in a Cole Mine



- The bird would act as an early warning for harmful gas



Mitigating Buffer Overflows with Canary 29

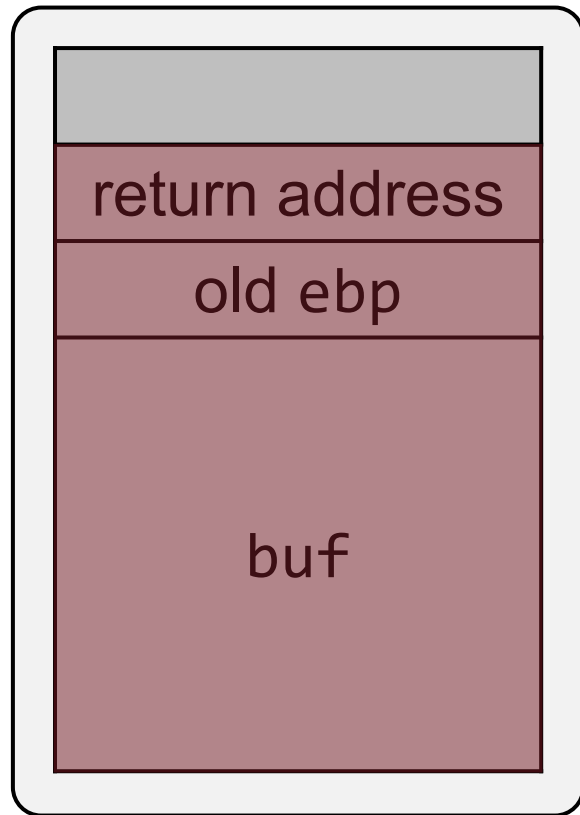
- Early warnings of buffer overflows
- First introduced in 1998

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, ***USENIX Security 1998***

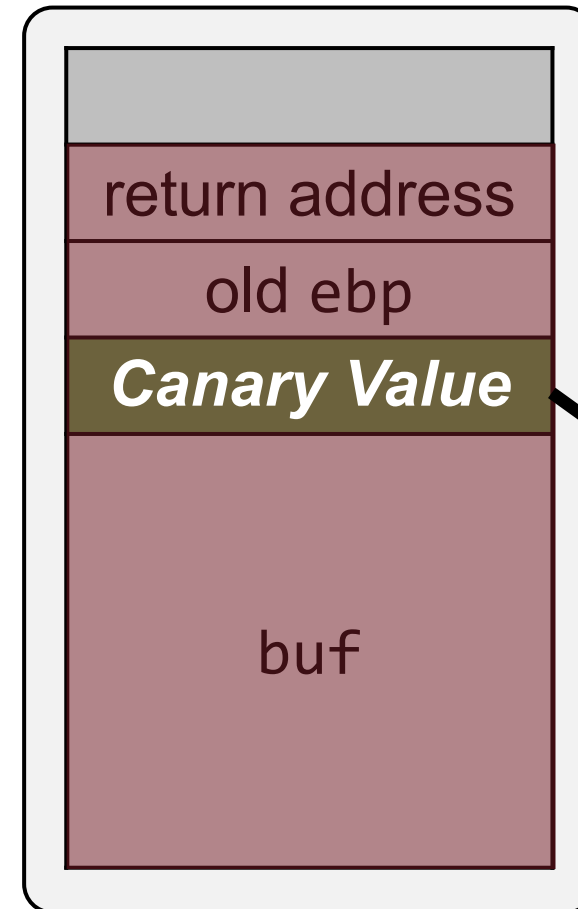
- Not necessarily used for stack, but can also be used for heap

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

- Key idea: insert a checking value before the return address



Without stack canary



With stack canary

***Check
before
executing
return!***



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

- Key idea: insert a checking value before the return address

Before executing return, check...

(Inserted canary value)

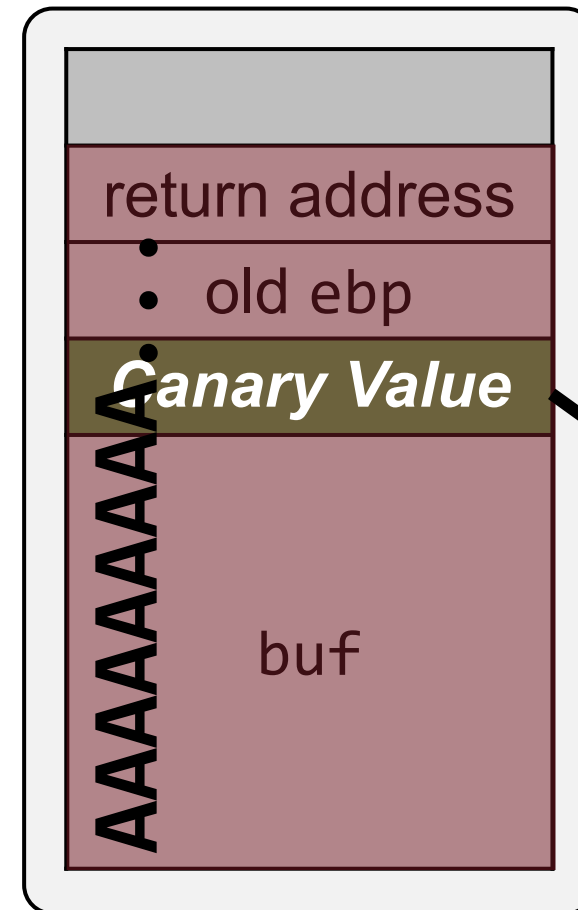
Canary Value

≠

(Current canary value)

0x41414141

Overflow is occurred!
Stop the program



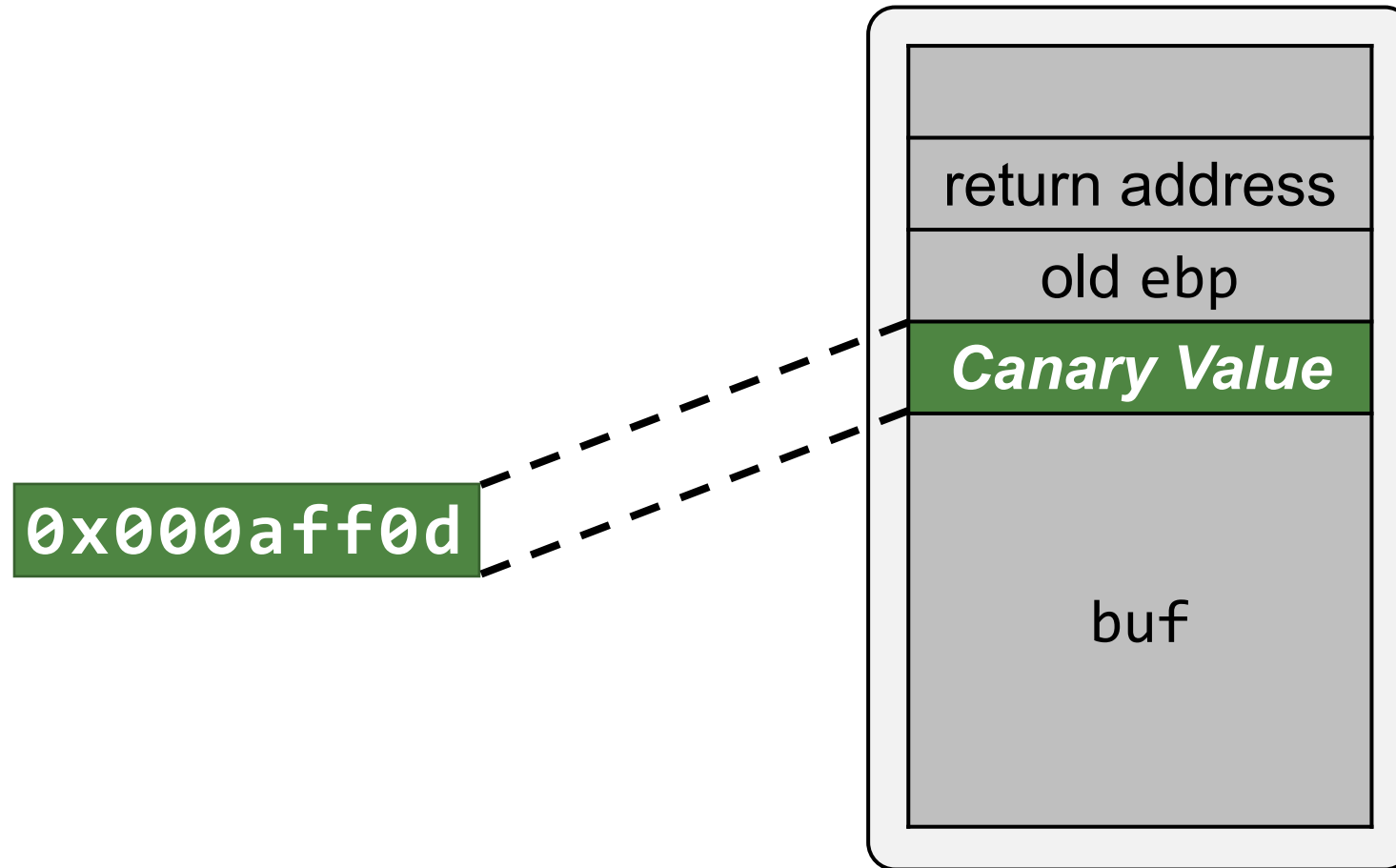
**Check
before
executing
return!**

With stack canary

StackGuard (1998)



- Uses a constant canary value 0x000aff0d



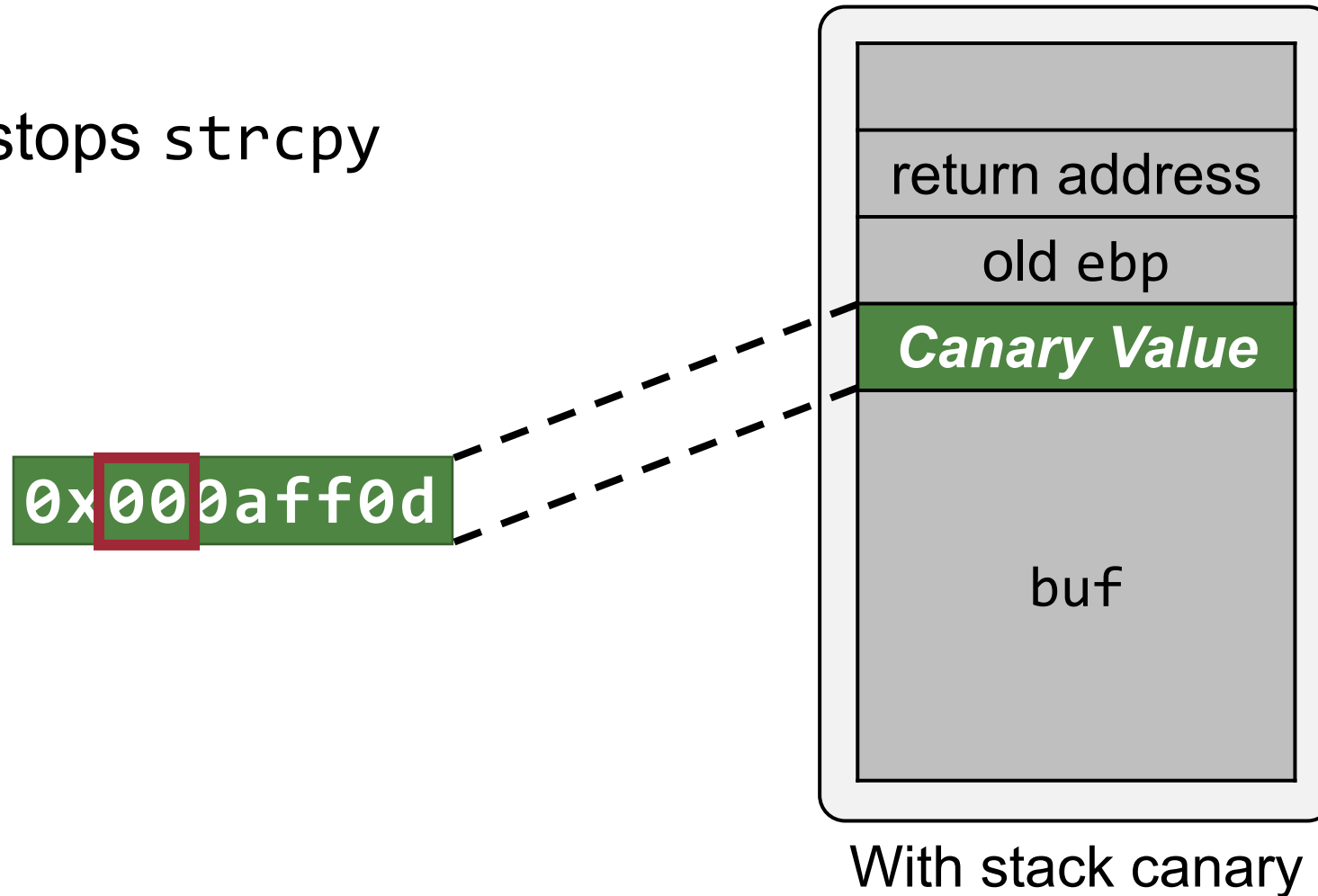
With stack canary

StackGuard (1998)



- Uses a constant canary value 0x000aff0d

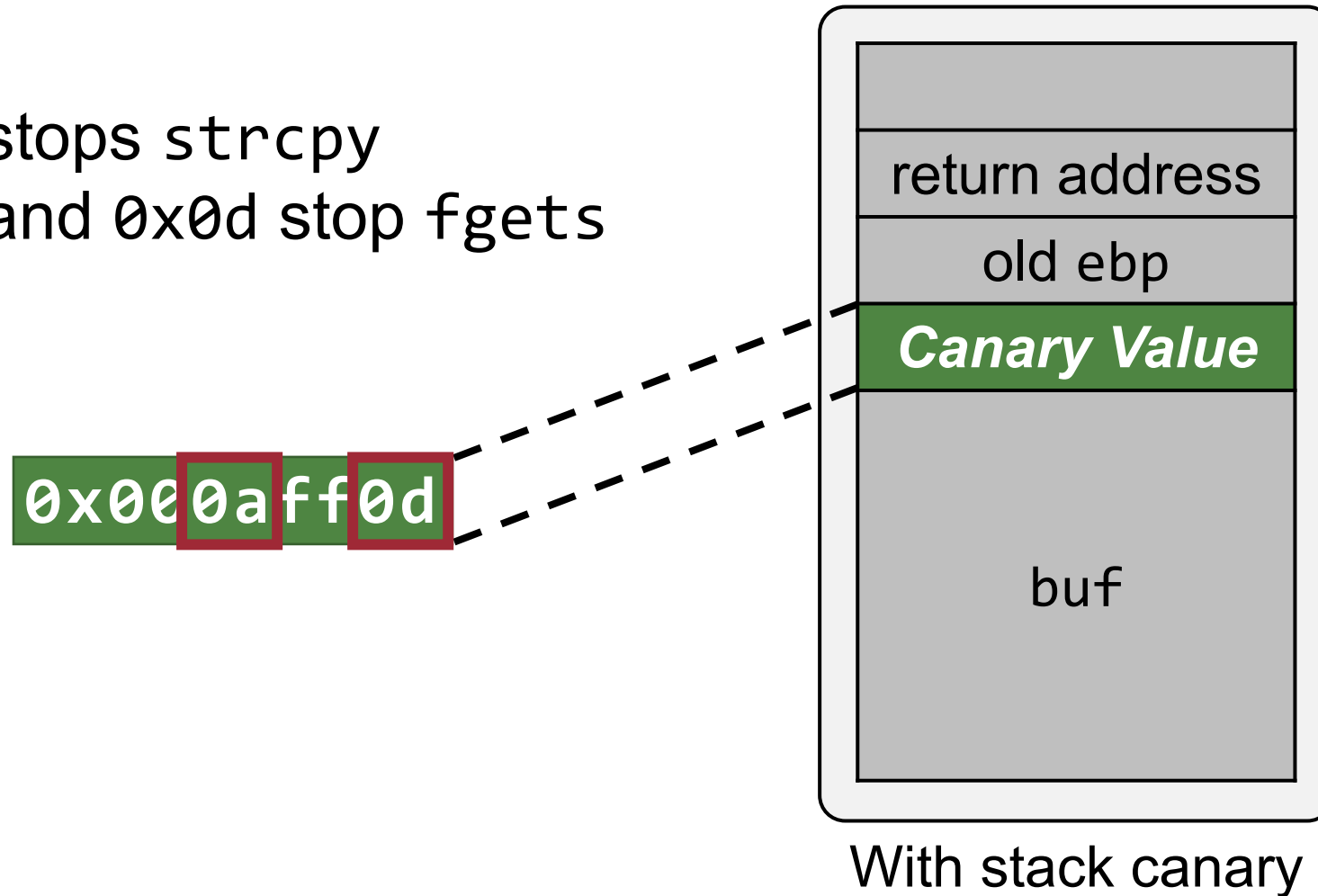
✓ 0x00 stops strcpy



StackGuard (1998) *

- Uses a constant canary value 0x000aff0d

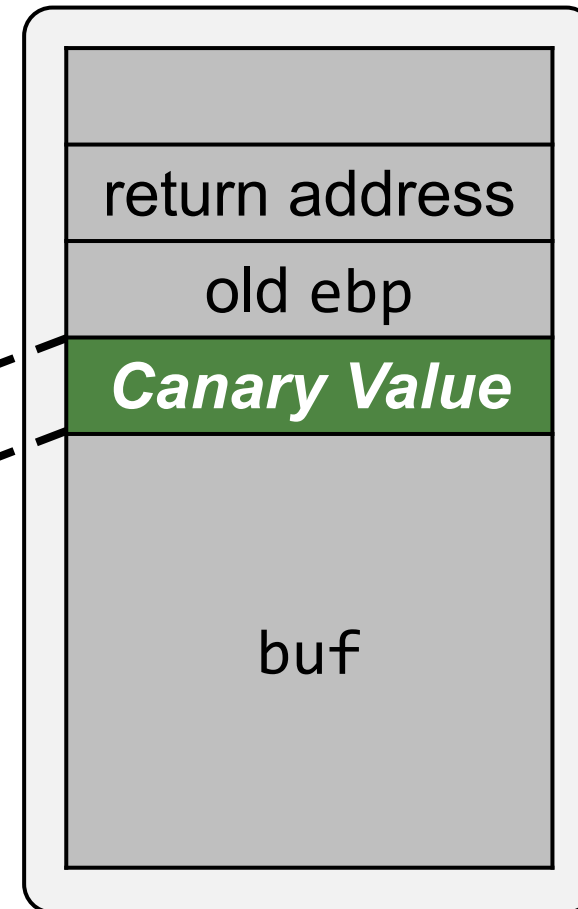
- ✓ 0x00 stops strcpy
- ✓ 0x0a and 0x0d stop fgets



StackGuard (1998) *

- Uses a constant canary value 0x000aff0d
 - ✓ 0x00 stops strcpy
 - ✓ 0x0a and 0x0d stop fgets
 - ✓ 0xff stops EOF checks

0x000aff0d



With stack canary

Problem of Using a Constant Canary Value

36



memcpy?

Problem of Using a Constant Canary Value

37



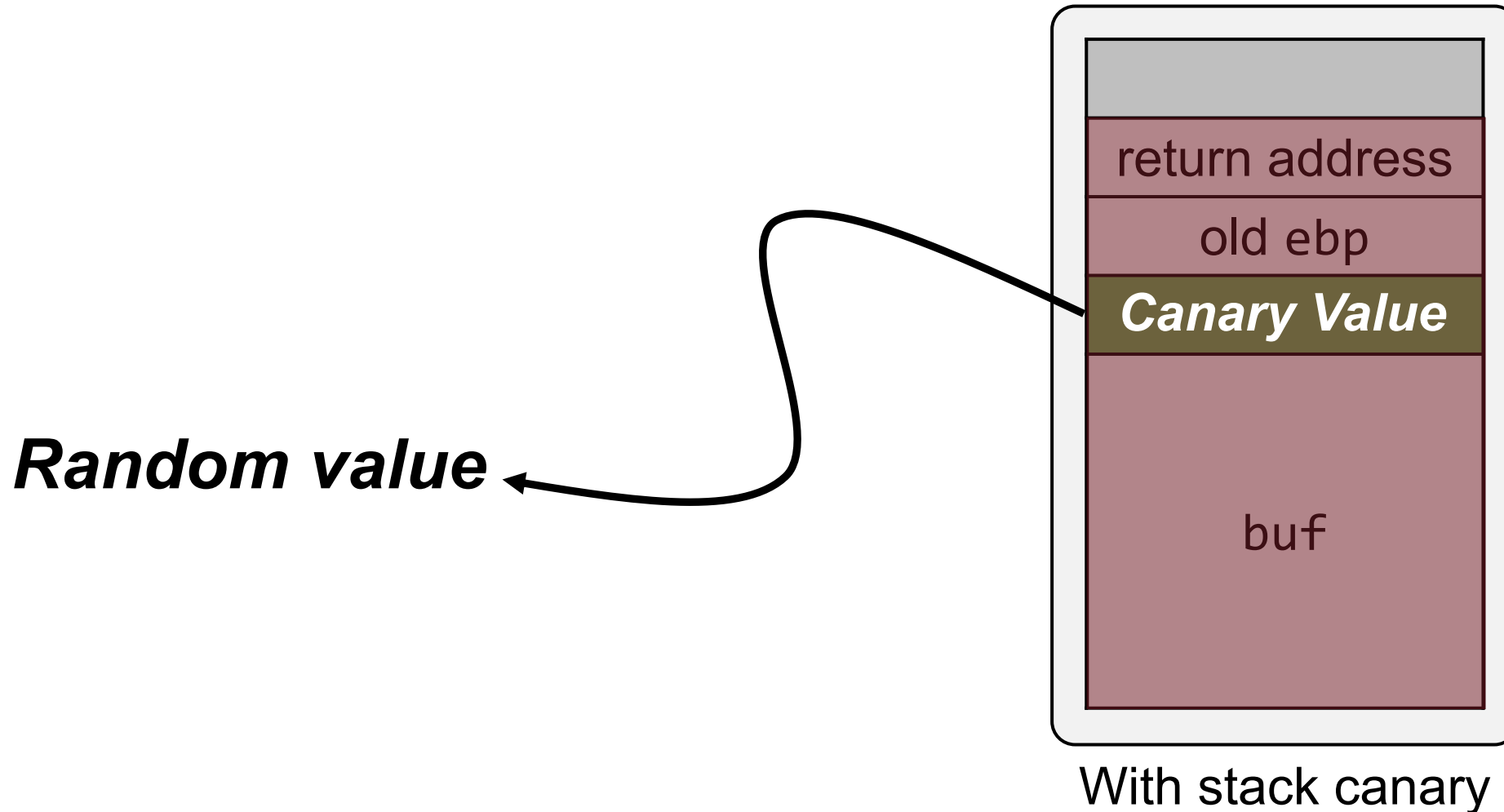
```
memcpy(void dest, void src, size_t n)
```

The `memcpy()` function copies **n bytes** from memory area `src` to memory area `dest`

Random Canaries

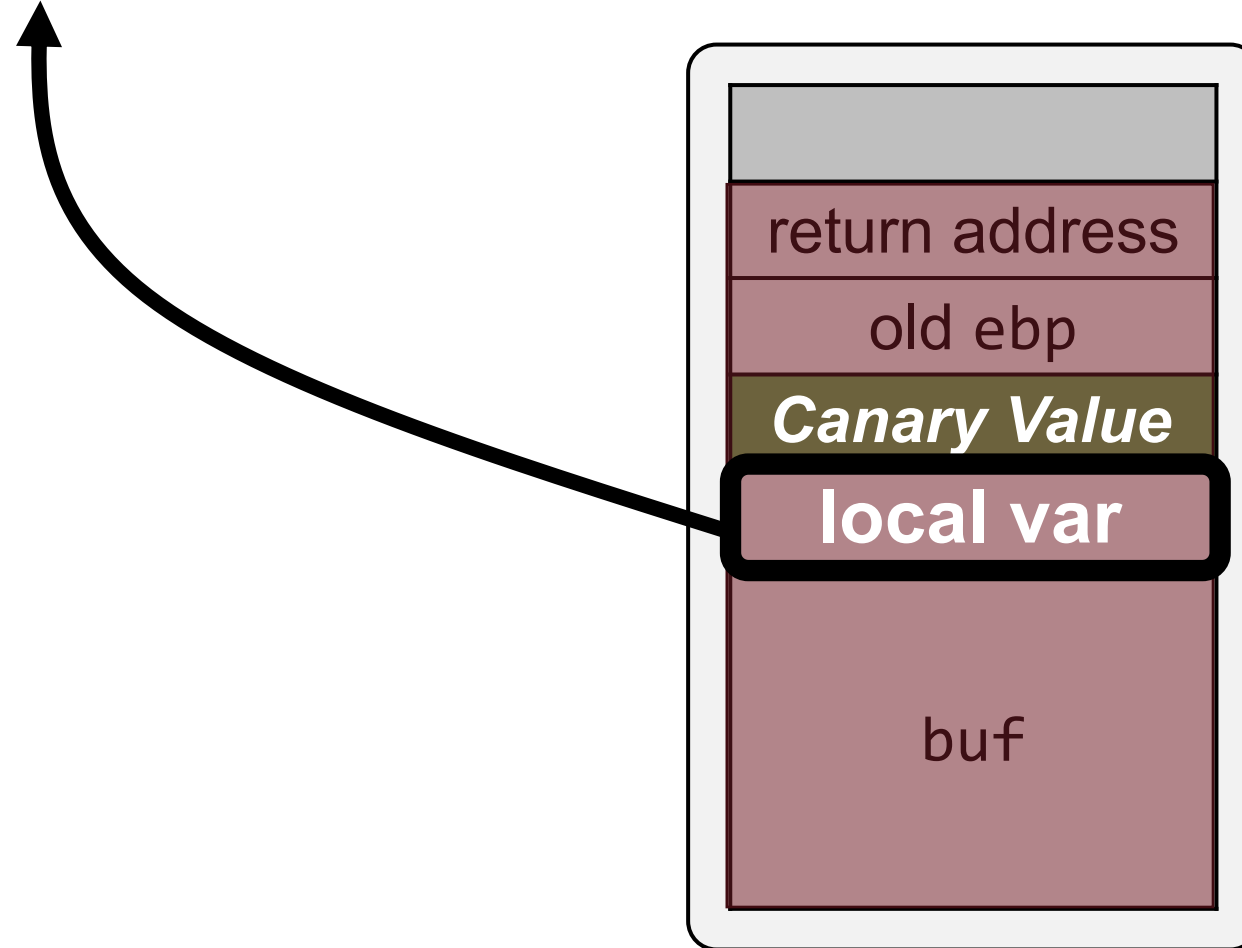


- Pick a random value at process initialization, put it on the stack



Problem Still Exists

- Local variables are not protected!



With stack canary

Solution: Reordering Local Variables



- Always put local buffers after local pointers
- This idea is implemented by GCC 4.1 in 2005

GCC Stack Canary Implementation

```

80483fb: push    ebp
80483fc: mov     ebp, esp
80483fe: sub     esp, 0x100
8048404: push    DWORD PTR [ ebp+0x8 ]
8048407: lea     eax, [ ebp-0x100 ]
804840d: push    eax
804840e: call    80482d0 <strcpy@plt>
8048413: add     esp, 0x8
8048416: leave
8048417: ret

```



```

804844b: push    ebp
804844c: mov     ebp, esp
804844e: sub     esp, 0x108
8048454: mov     eax, DWORD PTR [ ebp+0x8 ]
8048457: mov     DWORD PTR [ ebp-0x108 ], eax
804845d: mov     eax, gs:0x14
8048463: mov     DWORD PTR [ ebp-0x4 ], eax
8048466: xor     eax, eax
8048468: push    DWORD PTR [ ebp-0x108 ]
804846e: lea     eax, [ ebp-0x104 ]
8048474: push    eax
8048475: call    8048320
804847a: add     esp, 0x8
804847d: mov     eax, DWORD PTR [ ebp-0x4 ]
8048480: xor     eax, DWORD PTR gs:0x14
8048487: je      804848e
8048489: call    8048310 <__stack_chk_fail@plt>
804848e: leave
804848f: ret

```

Without stack canary

gcc -fno-stack-protector

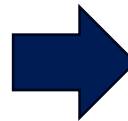
With **stack canary**

gcc -fstack-protector

GCC Stack Canary Implementation

42

```
80483fb: push    ebp
80483fc: mov     ebp, esp
80483fe: sub     esp, 0x100
8048404: push    DWORD PTR [ ebp+0x8 ]
8048407: lea     eax, [ ebp-0x100 ]
804840d: push    eax
804840e: call    80482d0 <strcpy@plt>
8048413: add     esp, 0x8
8048416: leave
8048417: ret
```



```
804844b: push    ebp
804844c: mov     ebp, esp
804844e: sub     esp, 0x108
8048454: mov     eax, DWORD PTR [ ebp+0x8 ]
8048457: mov     DWORD PTR [ ebp-0x108 ], eax
```

```
804845d: mov     eax, gs:0x14
8048463: mov     DWORD PTR [ ebp-0x4 ], eax
8048466: xor     eax, eax
```

```
8048468: push    DWORD PTR [ ebp-0x108 ]
804846e: lea     eax, [ ebp-0x104 ]
8048474: push    eax
8048475: call    8048320
804847a: add     esp, 0x8
```

```
804847d: mov     eax, DWORD PTR [ ebp-0x4 ]
8048480: xor     eax, DWORD PTR gs:0x14
8048487: je      804848e
8048489: call    8048310 <_stack_chk_fail@plt>
```

```
804848e: leave
```

```
804848f: ret
```

Without stack canary
gcc -fno-stack-protector

With **stack canary**
gcc -fstack-protector

GCC Stack Canary Implementation

43

Random canary value
at `gs:0x14`

```
8048454: mov     eax, esp
8048455: mov     eax, [ebp+0x108]
8048457: mov     DWORD PTR [ebp-0x108], eax
804845d: mov     eax, gs:0x14
8048463: mov     DWORD PTR [ebp-0x4], eax
8048466: xor     eax, eax
8048468: push    DWORD PTR [ebp-0x108]
804846e: lea     eax, [ebp-0x104]
8048474: push    eax
8048475: call    8048320
804847a: add     esp, 0x8
804847d: mov     eax, DWORD PTR [ebp-0x4]
8048480: xor     eax, DWORD PTR gs:0x14
8048487: je      804848e
8048489: call    8048310 <_stack_chk_fail@plt>
804848e: leave
804848f: ret
```

With **stack canary**
`gcc -fstack-protector`

Who Initializes `[gs:0x14]`?

Runtime Dynamic Linker (RTLD) does it every time it launches a **process**

// Below is roughly what RTLD does at process creation time

```
uintptr_t ret;
int fd = open("/dev/urandom", O_RDONLY);
if (fd >= 0) {
    ssize_t len = read(fd, &ret, sizeof(ret));
    if (len == (ssize_t) sizeof(ret)) {
        // inlined assembly for moving ret to [gs:0x14]
    }
}
```

GCC Stack Canary Implementation

Random canary value
at `gs:0x14`

Move canary value
onto the stack

Why?

```
8048457: mov     eax, DWORD PTR [ ebp+0x8 ]
8048457: mov     DWORD PTR [ ebp-0x108 ], eax
804845d: mov     eax, gs:0x14
8048463: mov     DWORD PTR [ ebp-0x4 ], eax
8048466: xor     eax, eax
8048468: push    DWORD PTR [ ebp-0x108 ]
804846e: lea     eax, [ ebp-0x104 ]
8048474: push    eax
8048475: call    8048320
804847a: add     esp, 0x8
804847d: mov     eax, DWORD PTR [ ebp-0x4 ]
8048480: xor     eax, DWORD PTR gs:0x14
8048487: je      804848e
8048489: call    8048310 <_stack_chk_fail@plt>
804848e: leave
804848f: ret
```

With **stack canary**
`gcc -fstack-protector`

GCC Stack Canary Implementation

46

```
804844b: push ebp
804844c: mov  ebp, esp
804844e: sub  esp, 0x108
8048454: mov  eax, DWORD PTR [ebp+0x8]
8048457: mov  DWORD PTR [ebp-0x108], eax
804845d: mov  eax, gs:0x14
8048463: mov  DWORD PTR [ebp-0x4], eax
8048466: xor  eax, eax
8048468: push DWORD PTR [ebp-0x108]
804846a: push DWORD PTR [ebp-0x104]
```

Get current canary value from stack

Compare to the original canary value

Jump to the leave instruction if equal

```
804847a: add  esp, 0x8
804847d: mov  eax, DWORD PTR [ebp-0x4]
8048480: xor  eax, DWORD PTR gs:0x14
8048487: je   804848e
8048489: call 8048310 <stack_chk_fail@plt>
804848e: leave
804848f: ret
```

With **stack canary**
gcc -fstack-protector

GCC Canary Implementation

- Uses a random canary value for every process creation
- Puts buffers after any local pointers on the stack

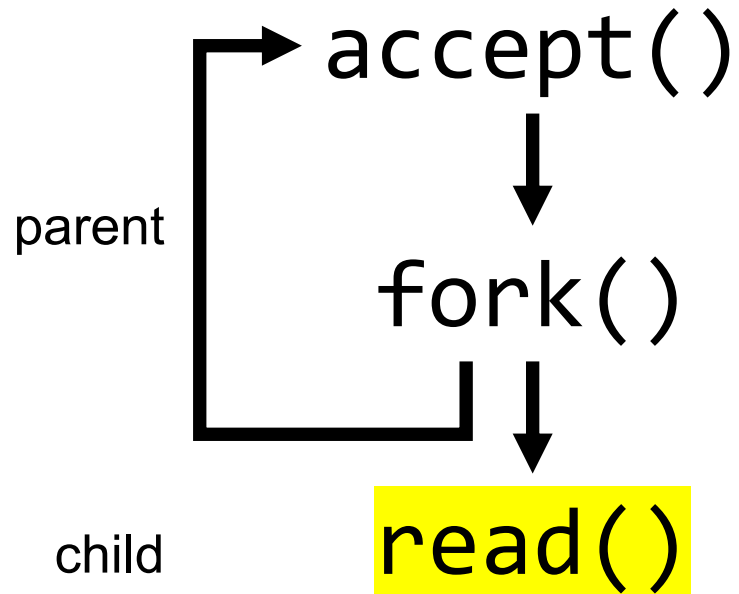
Attacking Canary Protection



Reused Canary Value

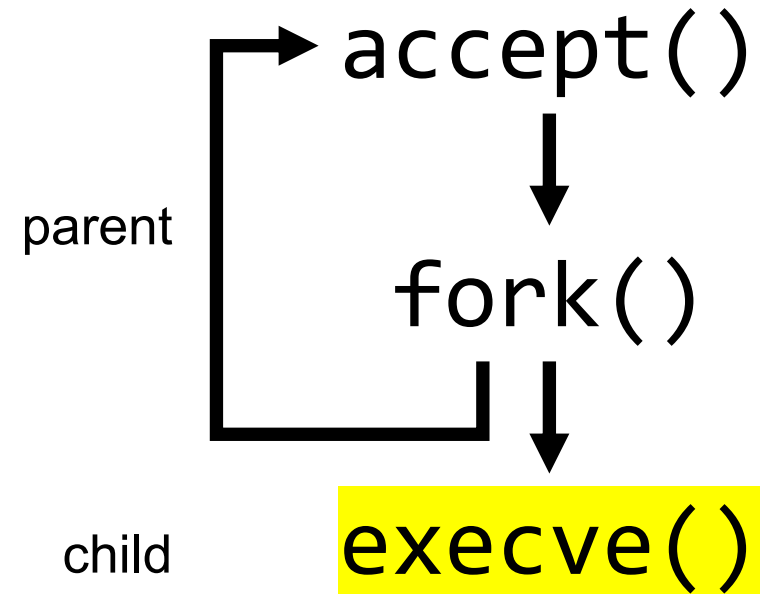


- Uses a random canary value for **every process creation**



Server Type #1

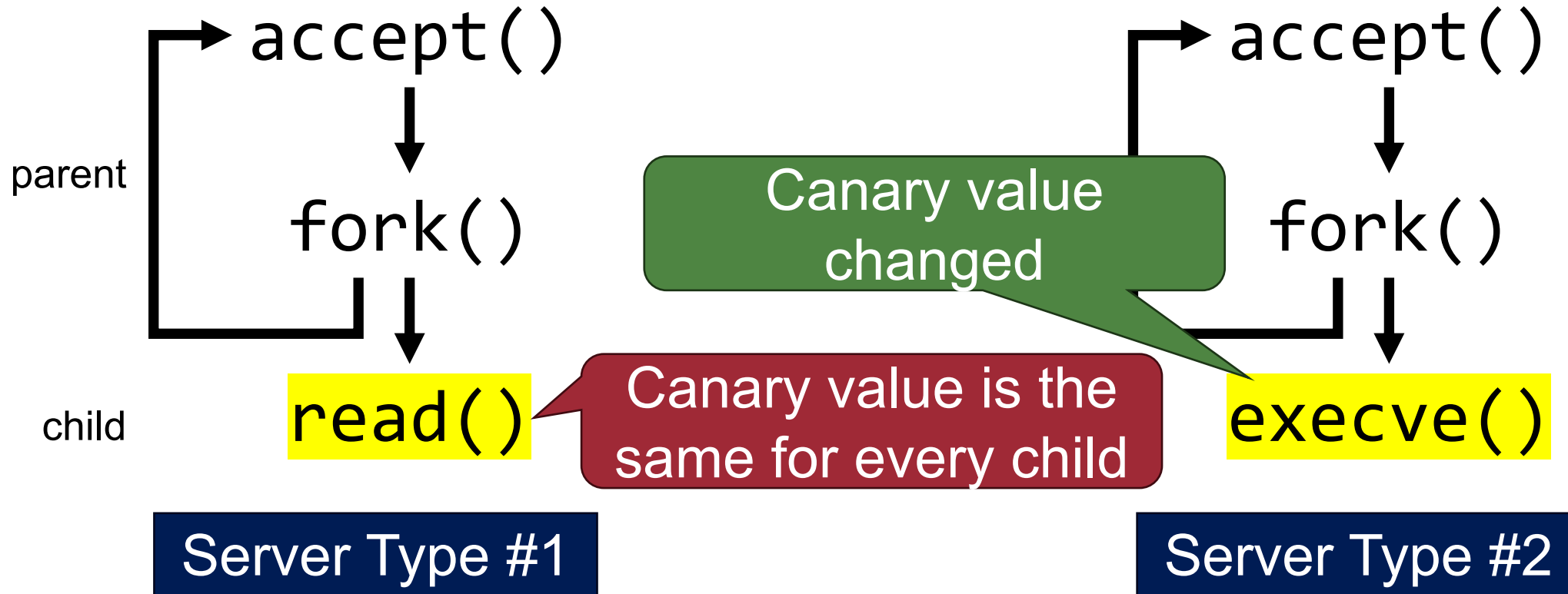
e.g., OpenSSH does this



Server Type #2

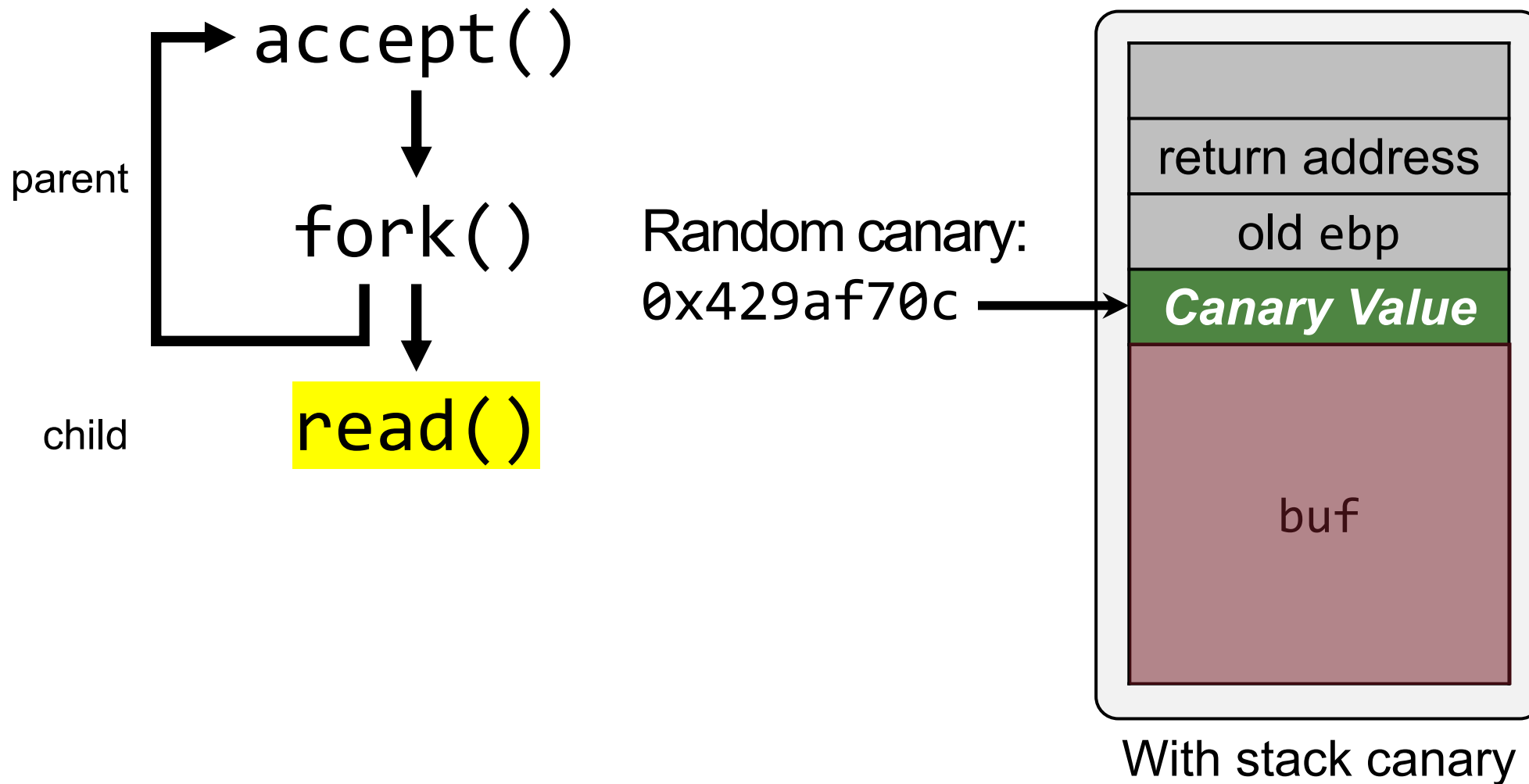
Reused Canary Value

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




e.g., OpenSSH does this

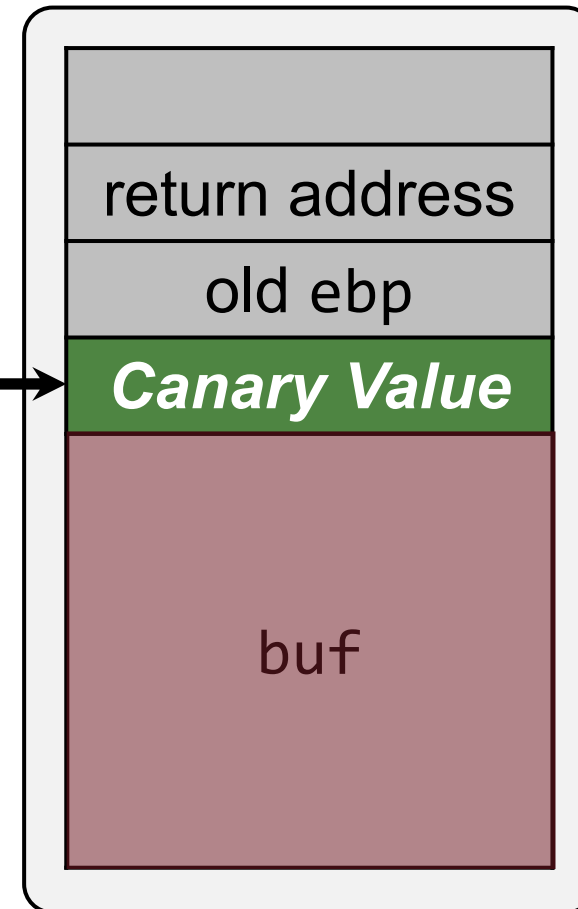
Attack #1: Byte-by-Byte Brute Forcing



Attack #1: 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

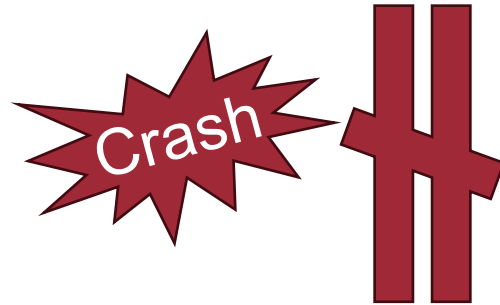


With stack canary

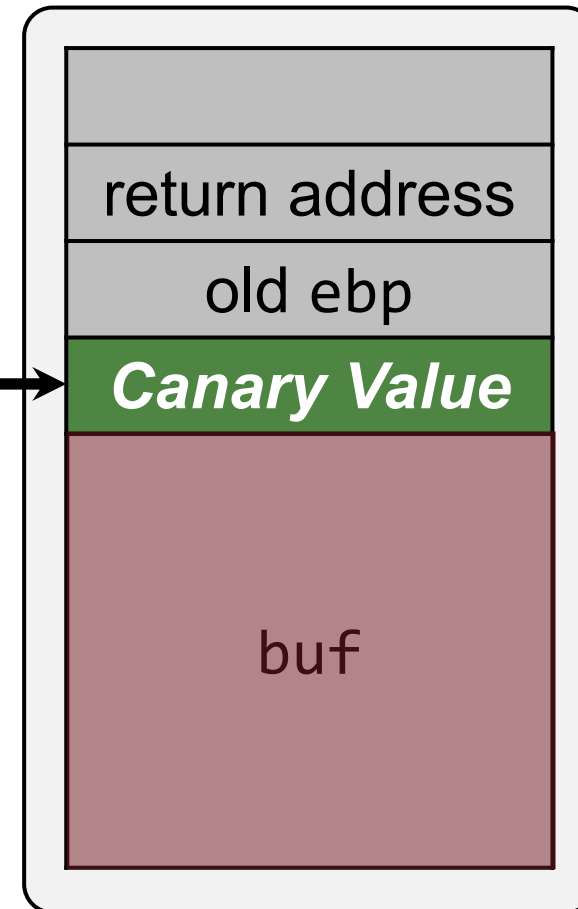
Attack #1: Byte-by-Byte Brute Forcing

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

Random canary:
`0x429af70c`




1st try: insert `\x00`

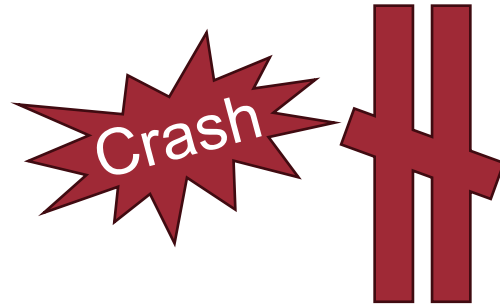


With stack canary

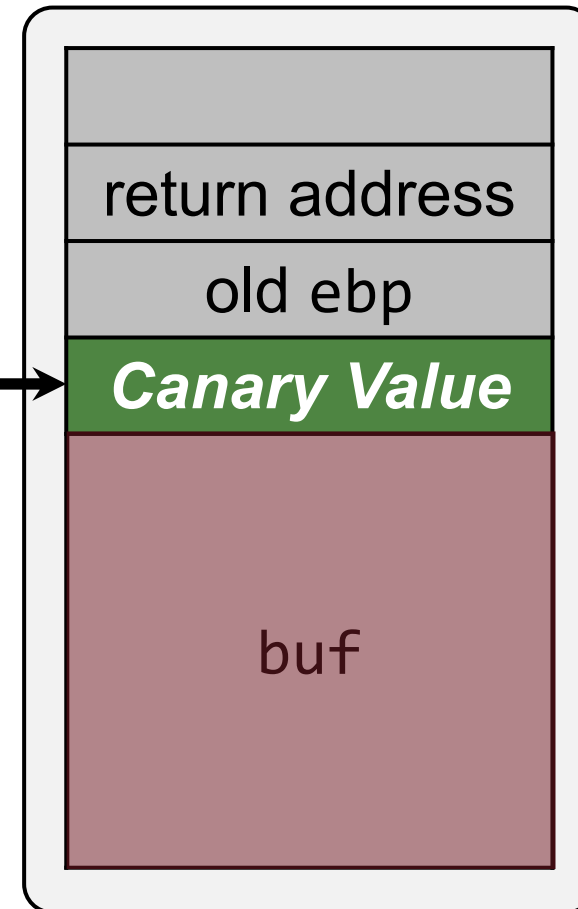
Attack #1: Byte-by-Byte Brute Forcing

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

Random canary:
0x429af70c




2nd try: insert \x01



With stack canary

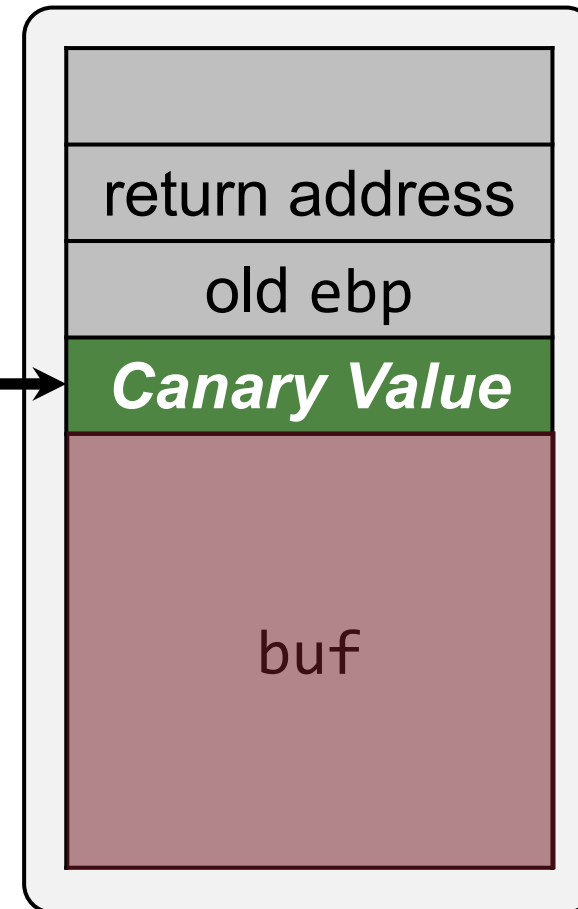
Attack #1: Byte-by-Byte Brute Forcing

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

Random canary:
0x429af70c




67th try: insert \x42



With stack canary

Attack #1: Byte-by-Byte Brute Forcing

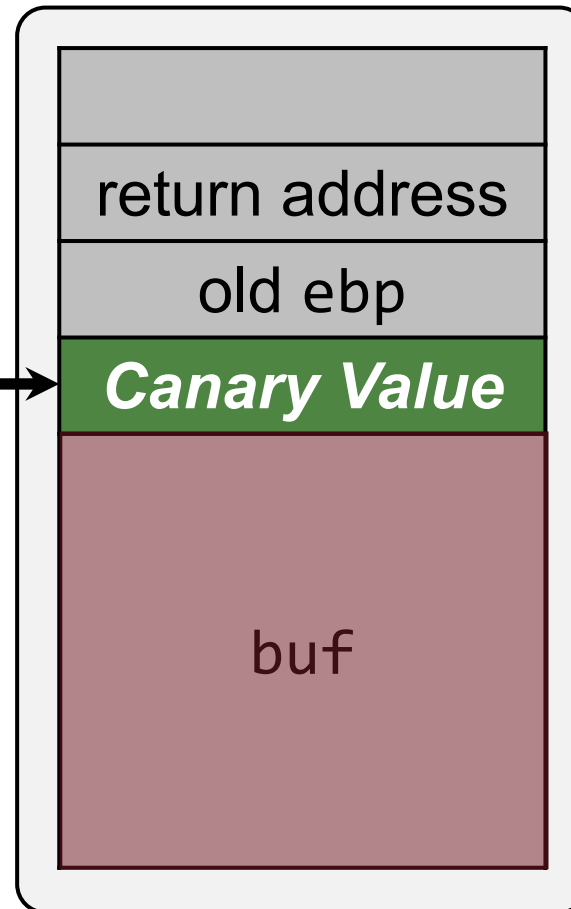
 Try to overwrite only 1 byte with a character from `\x00` to `\xff` until the program does not crash

Do the same for all 4 bytes!
⇒ Worst case?

67th try: insert `\x42`



Random canary:
`0x429af70c`



With stack canary

Protecting Canary Brute-Forcing Attack

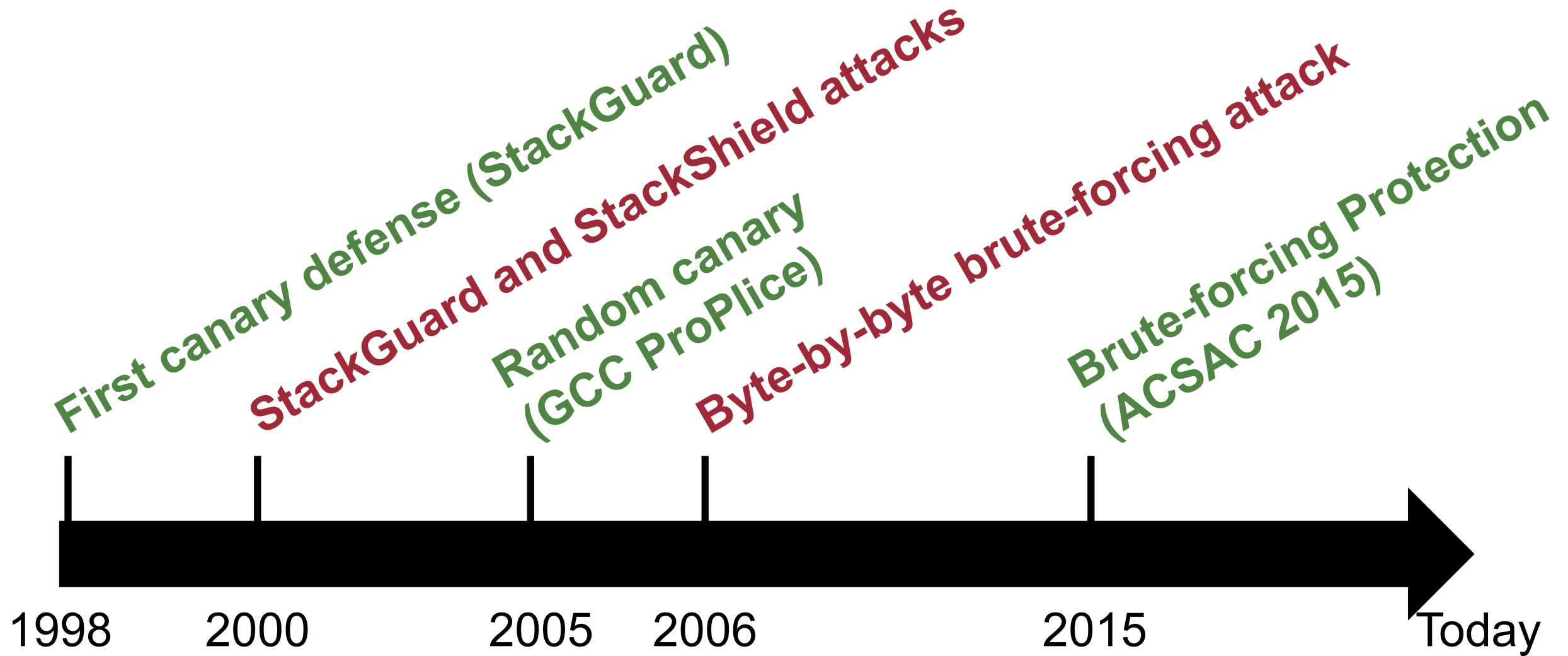
58

(Optional Reading)

DynaGuard: Armoring Canary-based Protections against Brute-force Attacks, **ACSAC 2015**

Canary Attack and Defense Timeline

59



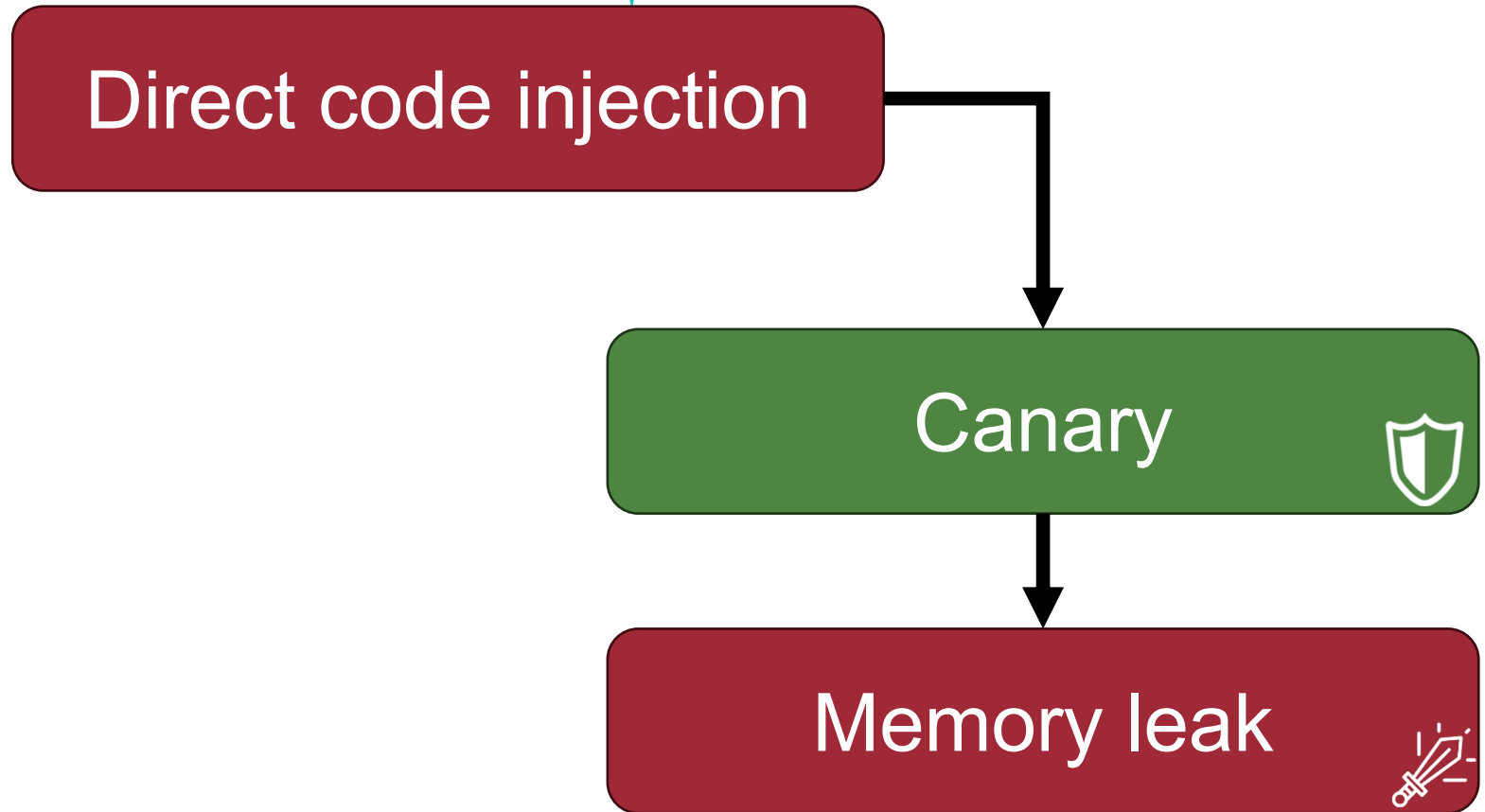
Attack #2: Leaking Canary Value



- 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*

Control Hijack Attack / Defense So Far

61



Buffer Overflow Mitigation #2: NX

NX (No eXecute)



a.k.a Data Execution Prevention* (***DEP***)

Stack stores data, but not code. Therefore, OS makes the stack memory area ***non-executable***

* DEP ***prevents*** data execution, but it does not prevent buffer overflows

NX (No eXecute)



AMD Athlon™ Processor Competitive Comparison

<i>FEATURES</i>	<i>AMD ATHLON™ CPU</i>	<i>PENTIUM® 4</i>
Architecture Introduction	2006	2000
Infrastructure	Socket AM2	Socket LGA775
Process Technology	90 nanometer, SOI 65 nanometer, SOI	90 nanometer
64-bit Instruction Set Support	Yes, AMD64 technology	Depends, EM64T on some Pentium® 4 series
Enhanced Virus Protection for Windows® XP SP2*	Yes	Depends

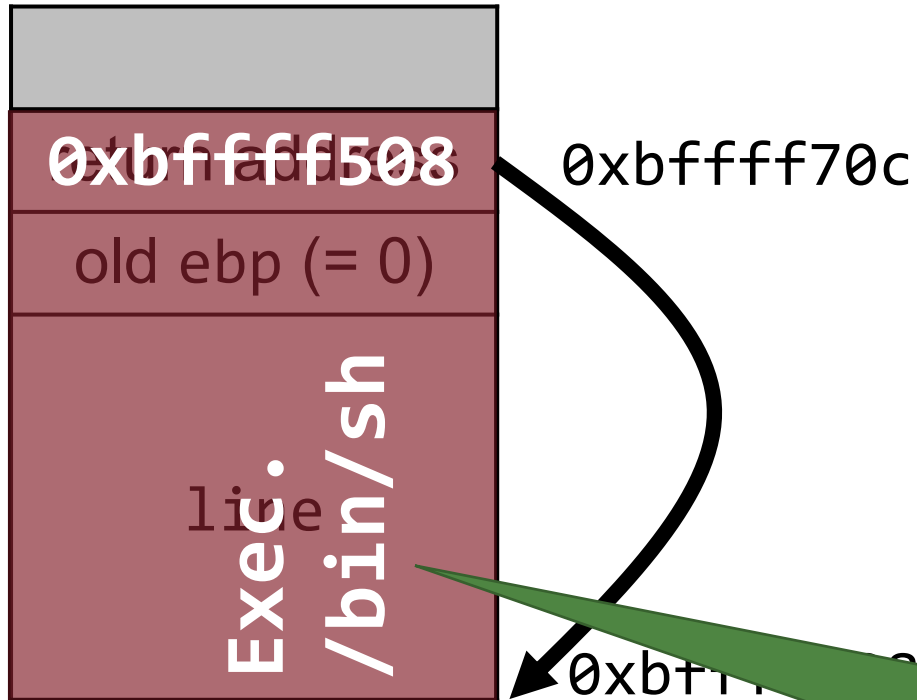
W \oplus X (Write XOR eXecute) Policy

On Linux, it is called W \oplus X

- Every page should be either writable or executable, but **NOT both**
- Even though we can put a shellcode to a writable buffer, we cannot execute it if this policy is enabled

Mitigating Control Flow Hijack with DEP

66



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

DEP on Stack using execstack

- Tool to set, clear, or query NX stack flag of binaries

```
$ /usr/sbin/execstack -s <filename> ; clear NX flag
```

```
$ /usr/sbin/execstack -c <filename> ; set NX flag
```

```
$ /usr/sbin/execstack -q <filename> ; query NX flag
```

When NX is set, return-to-stack exploit will fail
(i.e., the program will crash)

But,



DEP does not prevent buffer overflows. It prevents return-to-stack exploits, though

Any other ways to exploit buffer overflows?

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***)

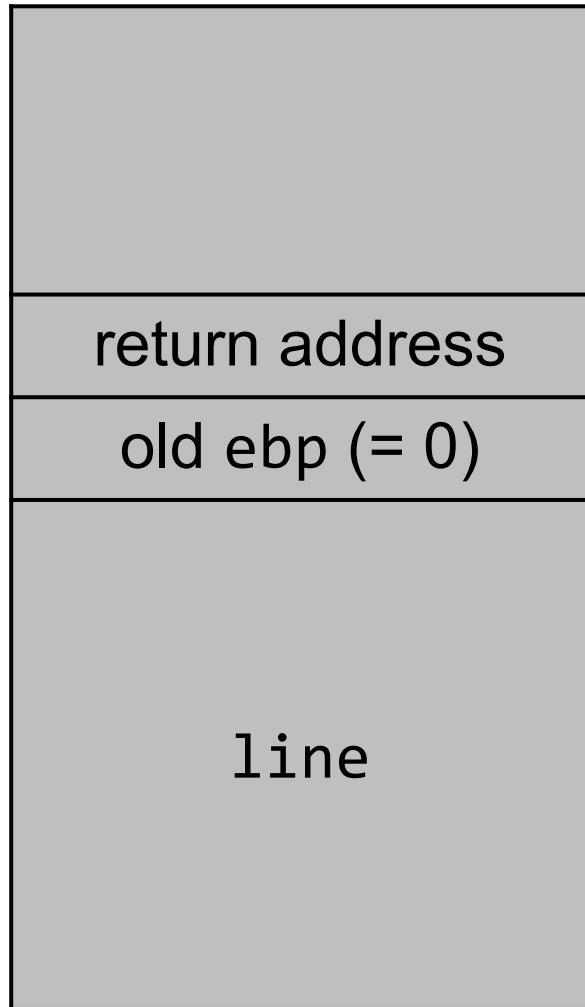
Code Reuse Attack #1: Return-to-Libc



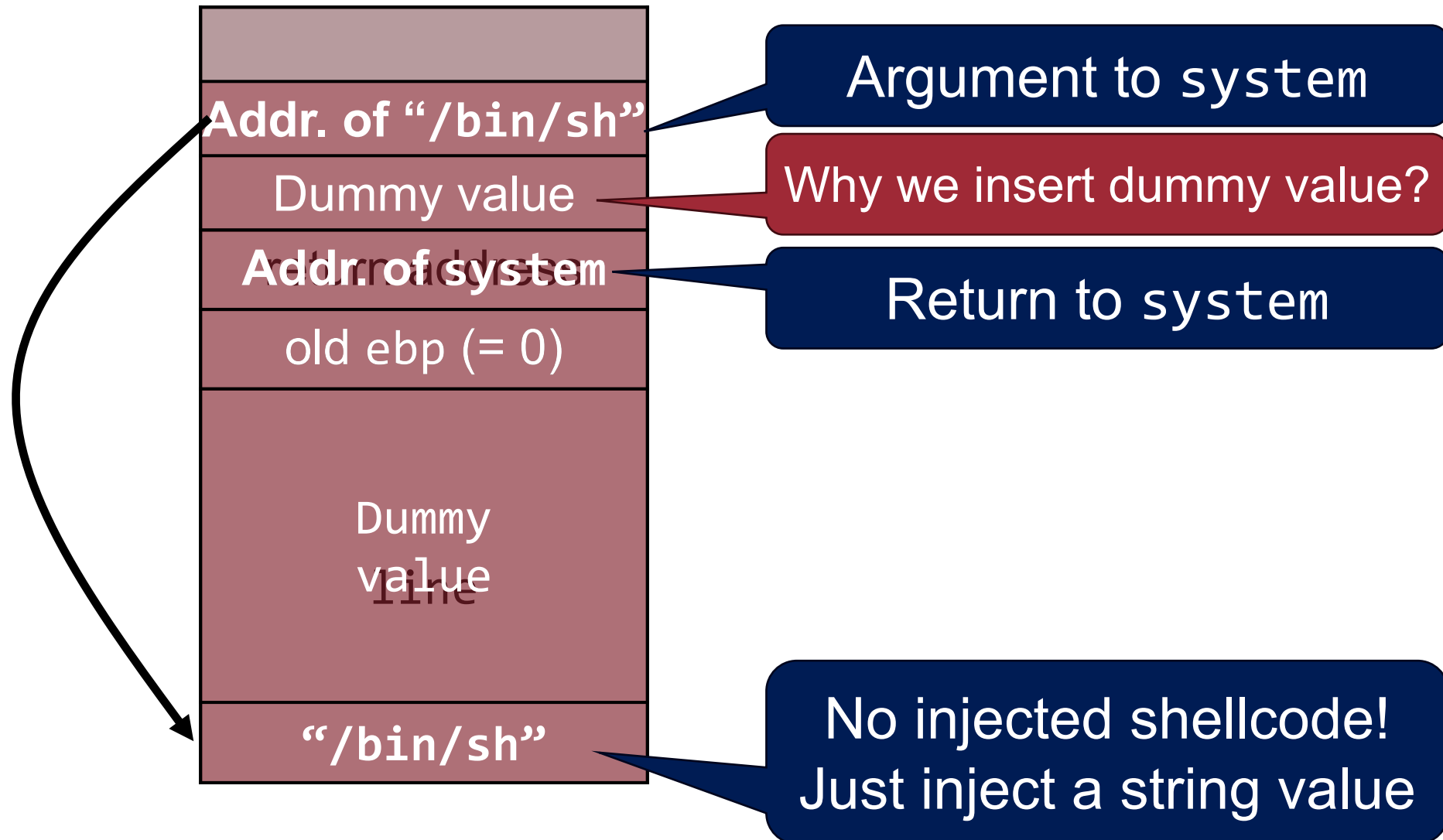
- LIBC (LIBrary C) is a standard library that most programs commonly use
 - For example, printf is in LIBC
- Many useful functions in LIBC to execute
 - exec family: `execl`, `execvp`, `execle`, ...
 - `system`
 - `mprotect`
 - `mmap`

Code Reuse Attack #1: Return-to-Libc

72



Code Reuse Attack #1: Return-to-Libc

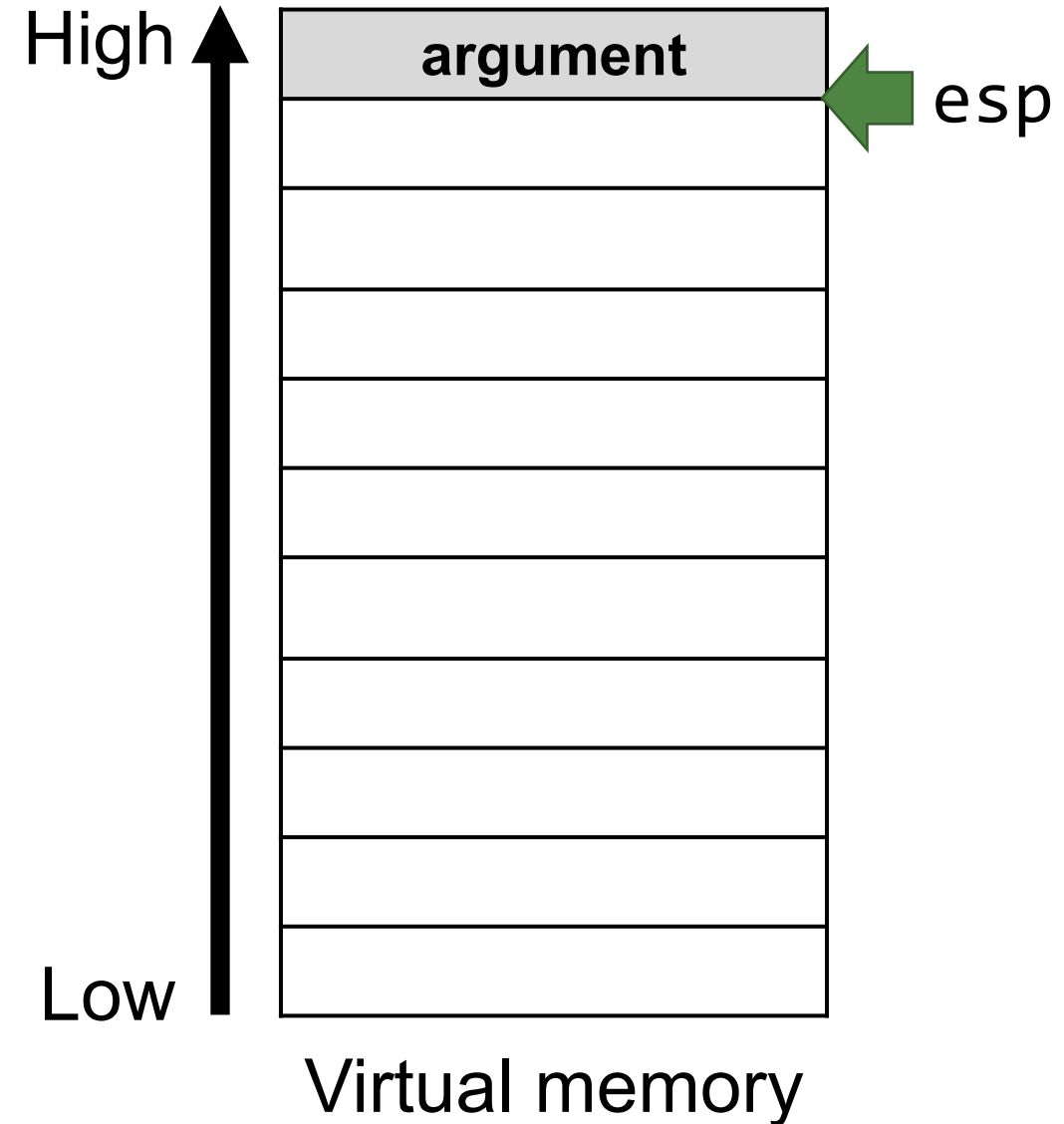


Recap: Function Call (call)

74

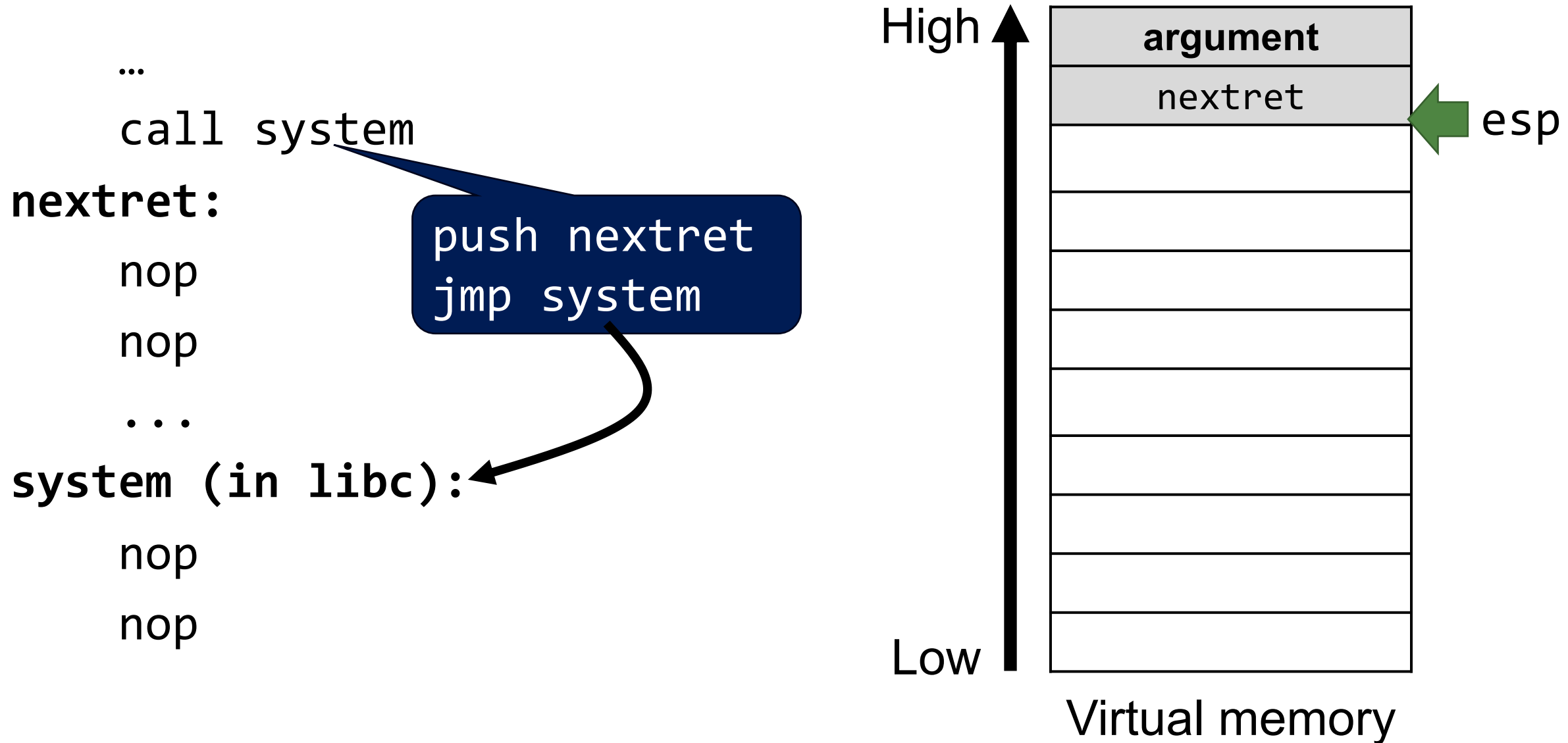
```
...  
call system  
nextret:  
  nop  
  nop  
...  
system (in libc):  
  nop  
  nop
```

push nextret
jmp system



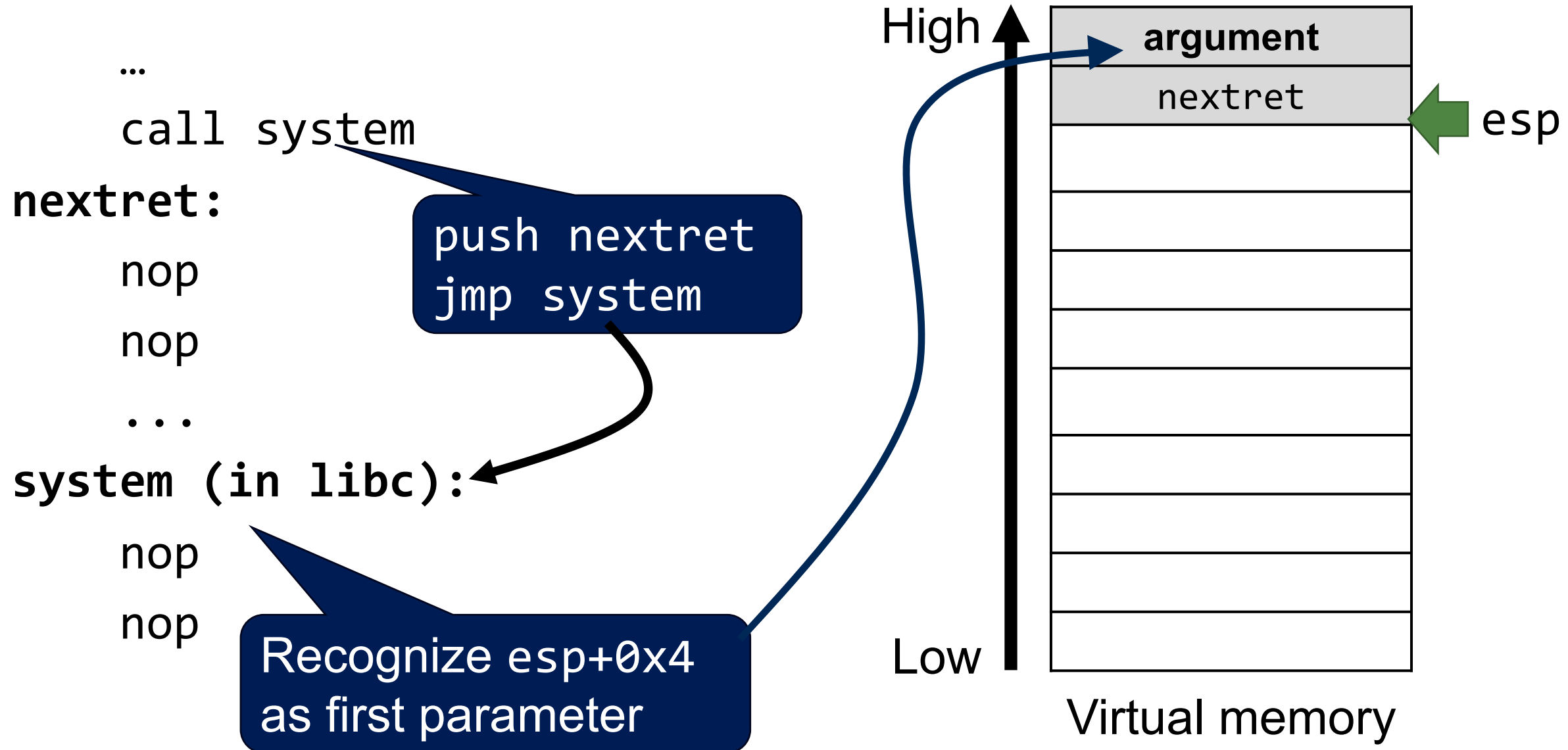
Recap: Function Call (call)

75

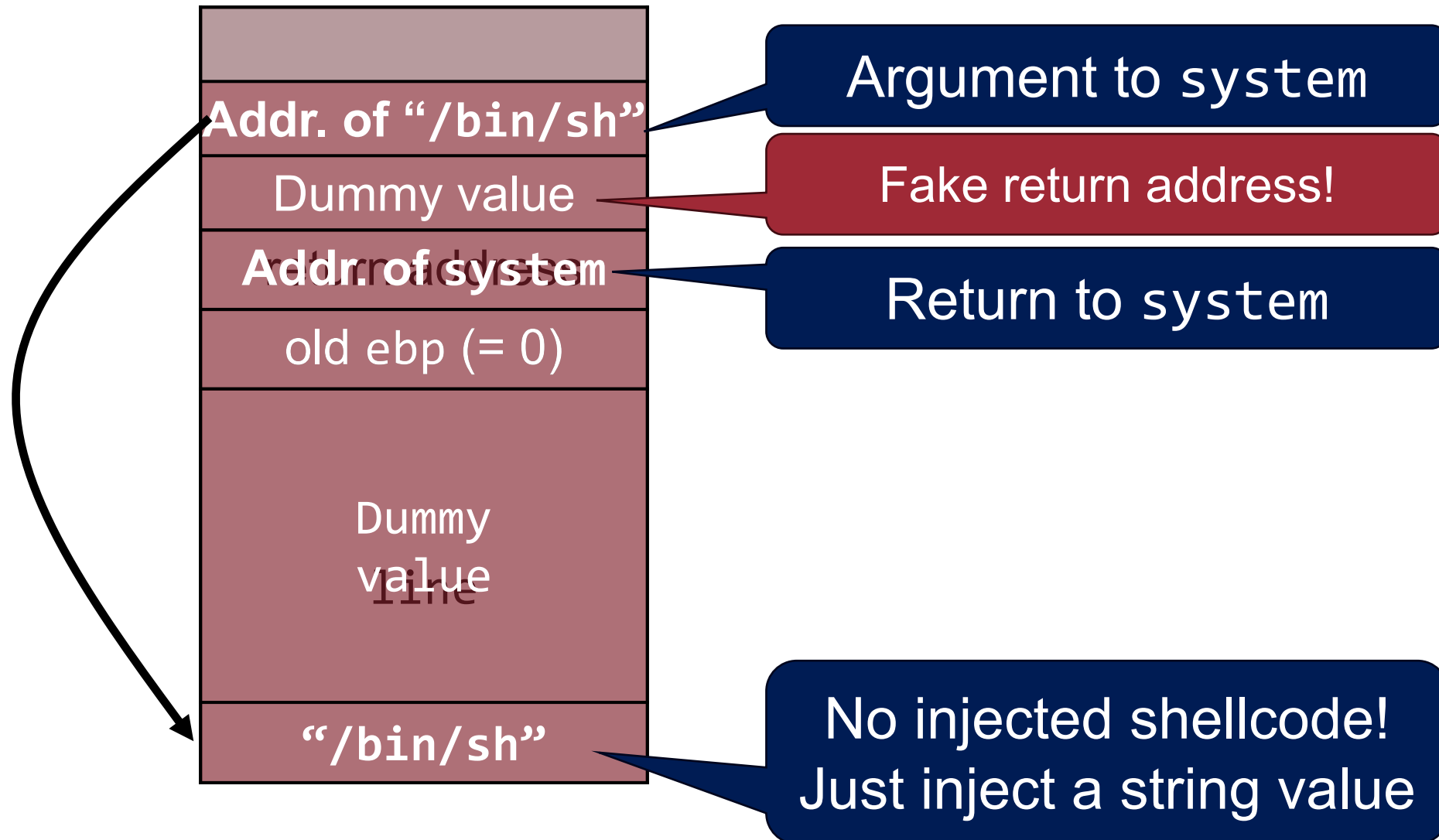


Recap: Function Call (call)

76



Code Reuse Attack #1: Return-to-Libc



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)”

Motivation of ROP



Return-to-Libc requires LIBC function calls, but can **we spawn a shell without the use of LIBC functions?**

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

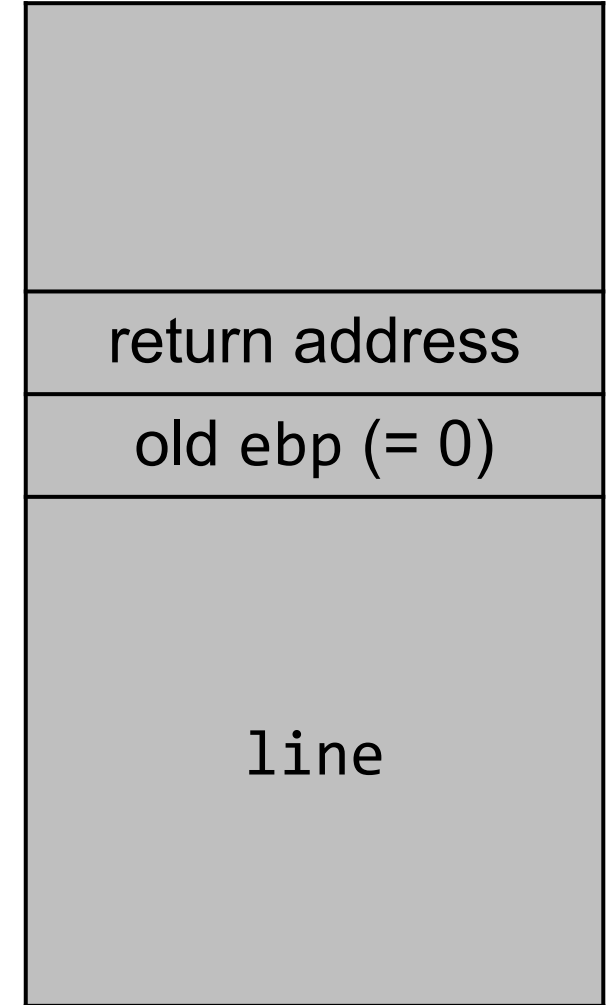
Return (ret) Chaining



Attacker's goal:

execute following instructions

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



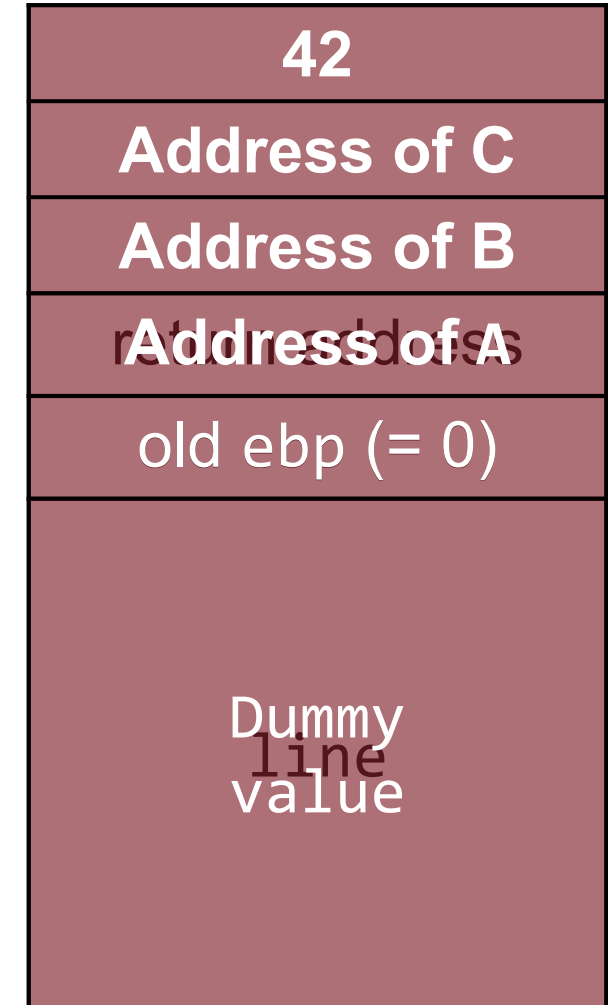
Return (ret) Chaining



Attacker's goal:

execute following instructions

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



Return (ret) Chaining

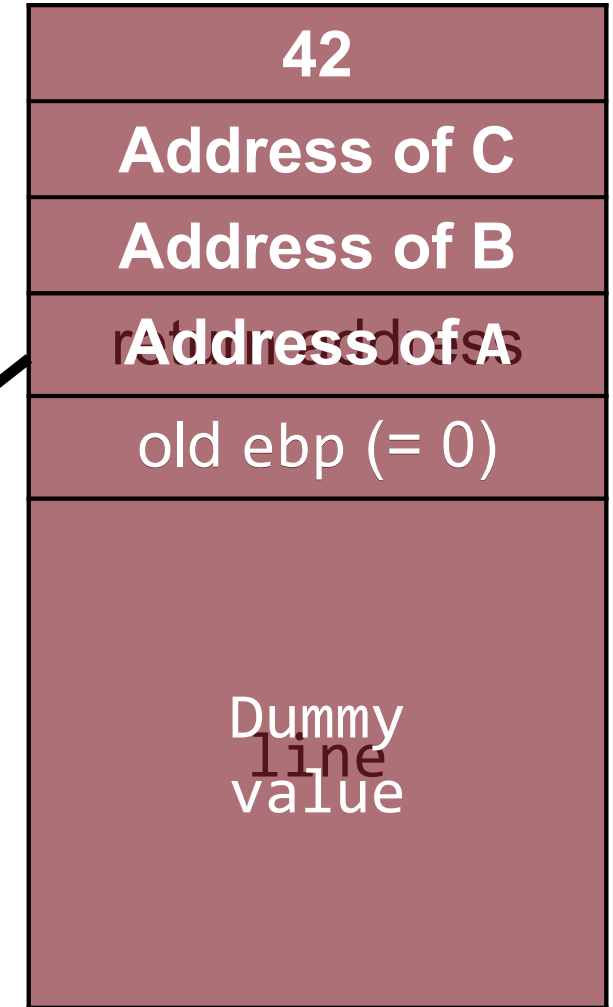
Attacker's goal:

execute following instructions

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

Somewhere in the
binary code

A | add eax, ebx
ret



Return (ret) Chaining



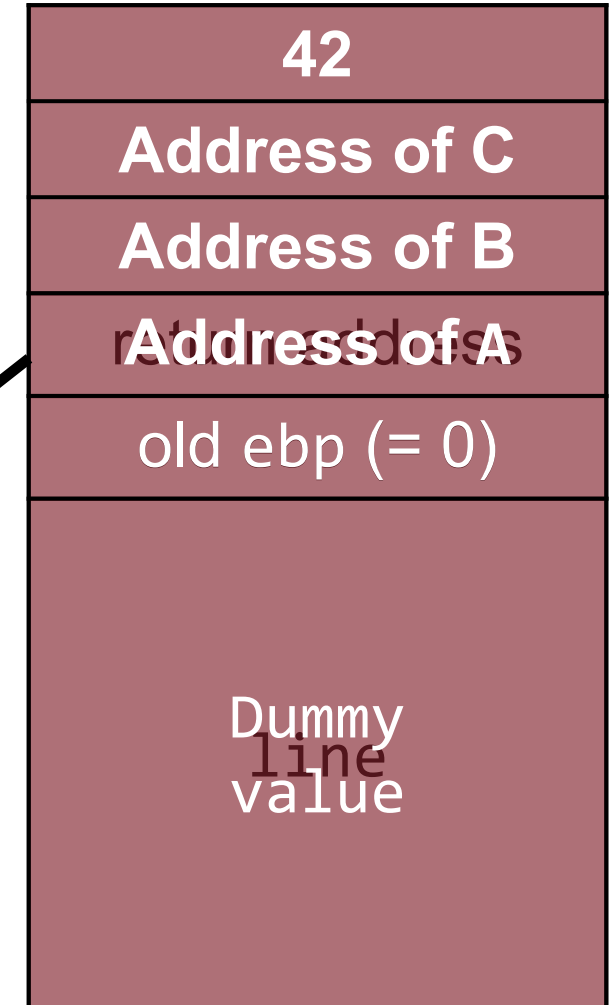
Attacker's goal:

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



Return (ret) Chaining



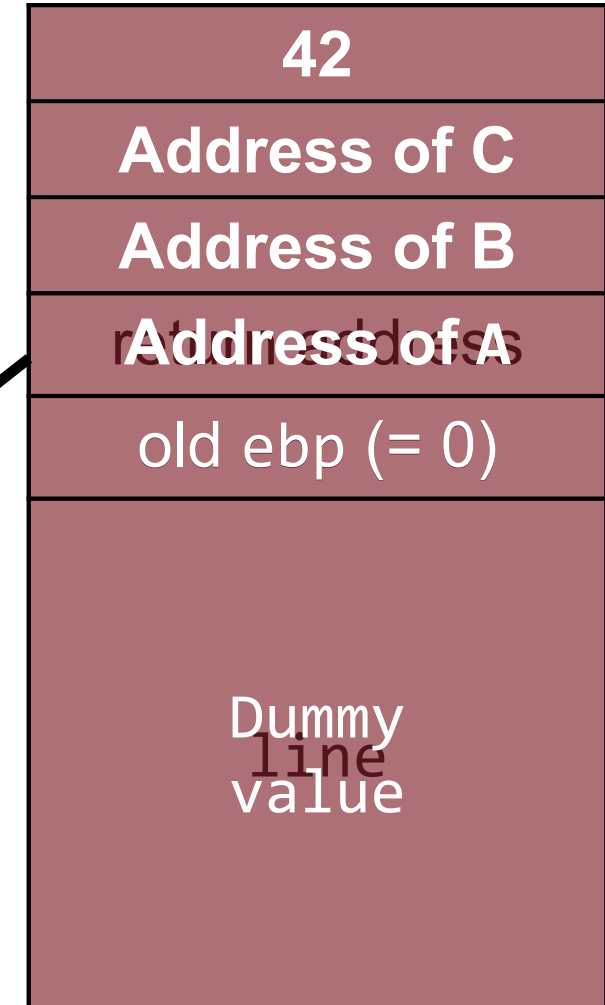
Attacker's goal:

execute following instructions

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

A

```
add eax, ebx
ret
```



Return (ret) Chaining

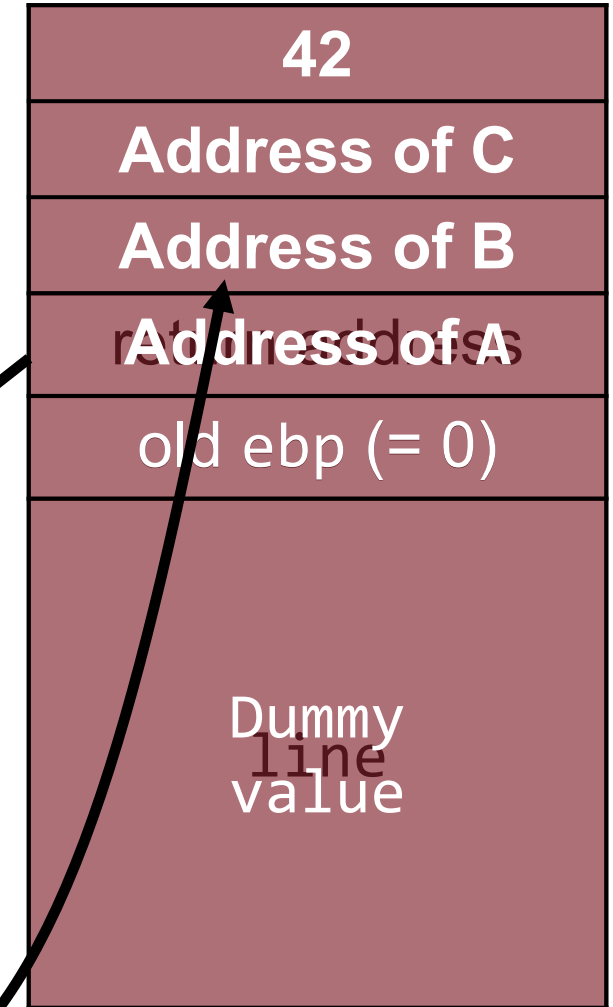
Attacker's goal:

execute following instructions

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

pop eip
= jump to another
gadget

A | add eax, ebx
ret



Return (ret) Chaining

Attacker's goal:

execute following instructions

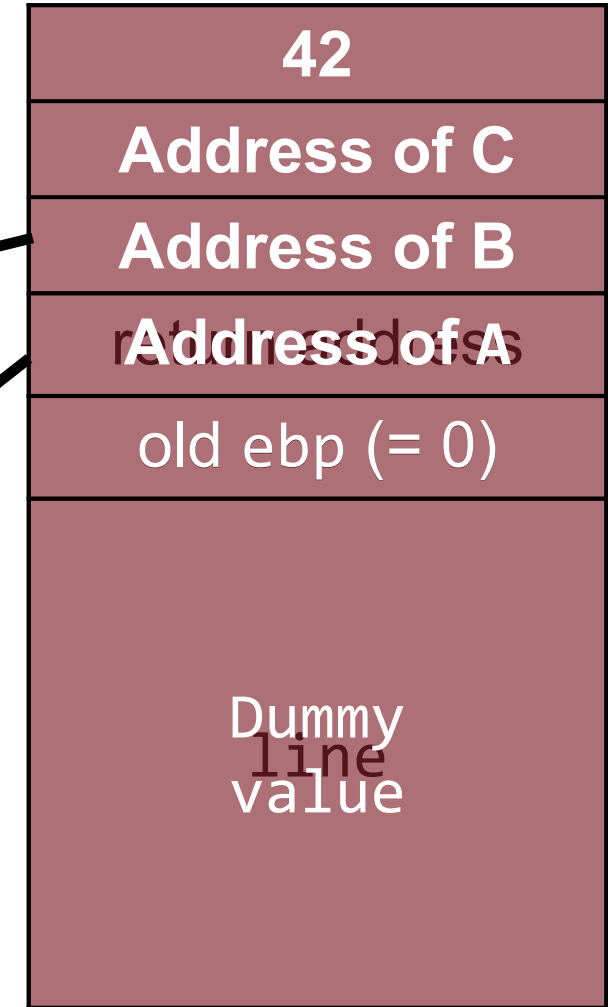
```
add eax, ebx  
mov ecx, eax  
inc ecx  
mov edx, 42
```

B

```
mov ecx, eax  
ret
```

A

```
add eax, ebx  
ret
```

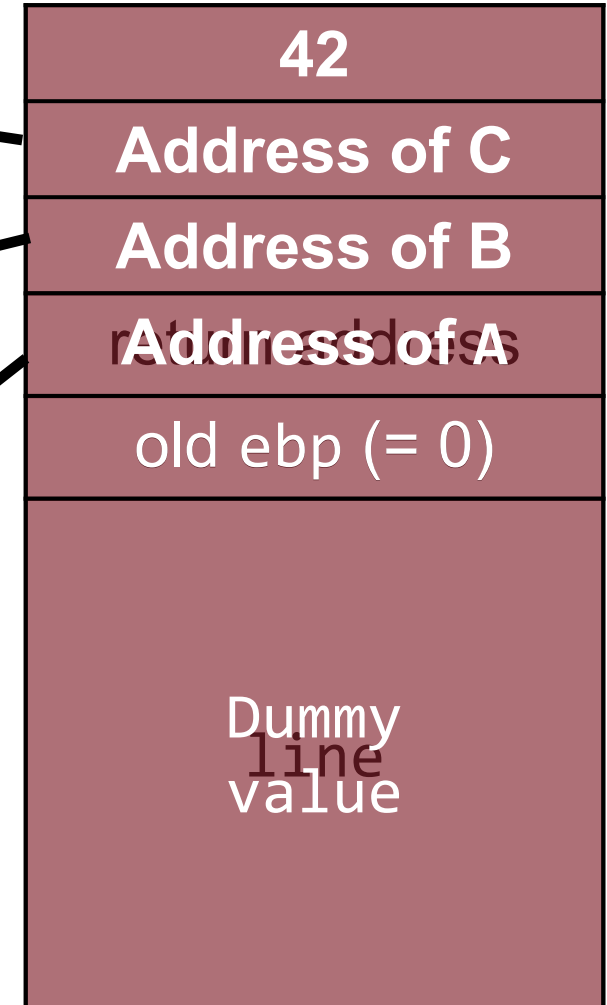
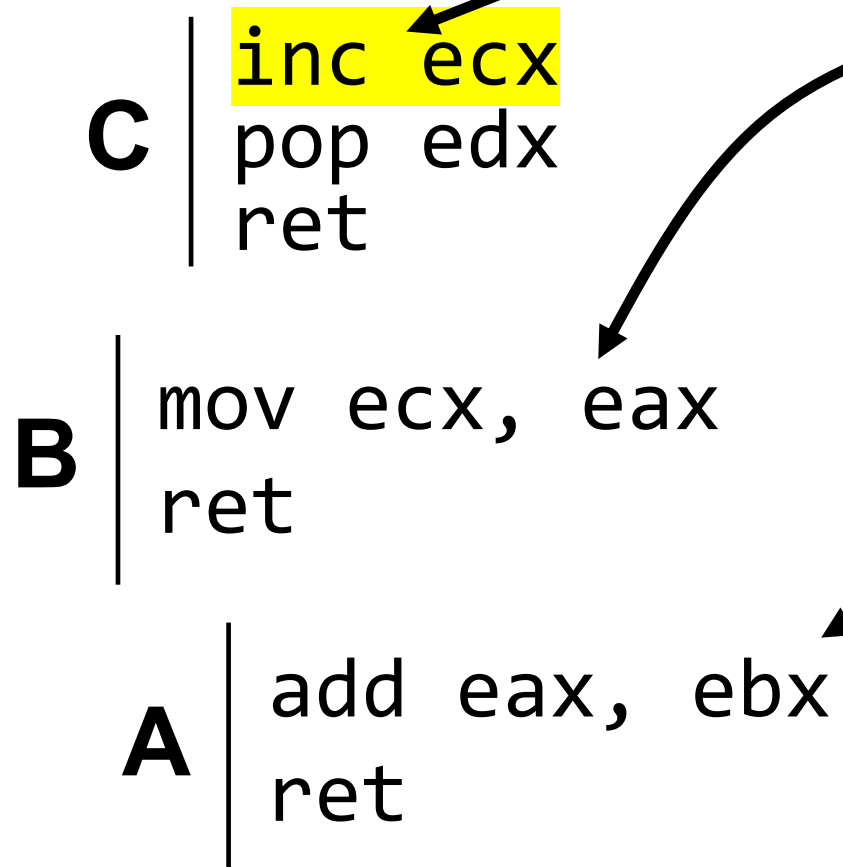


Return (ret) Chaining

Attacker's goal:

execute following instructions

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



Return (ret) Chaining

Attacker's goal:

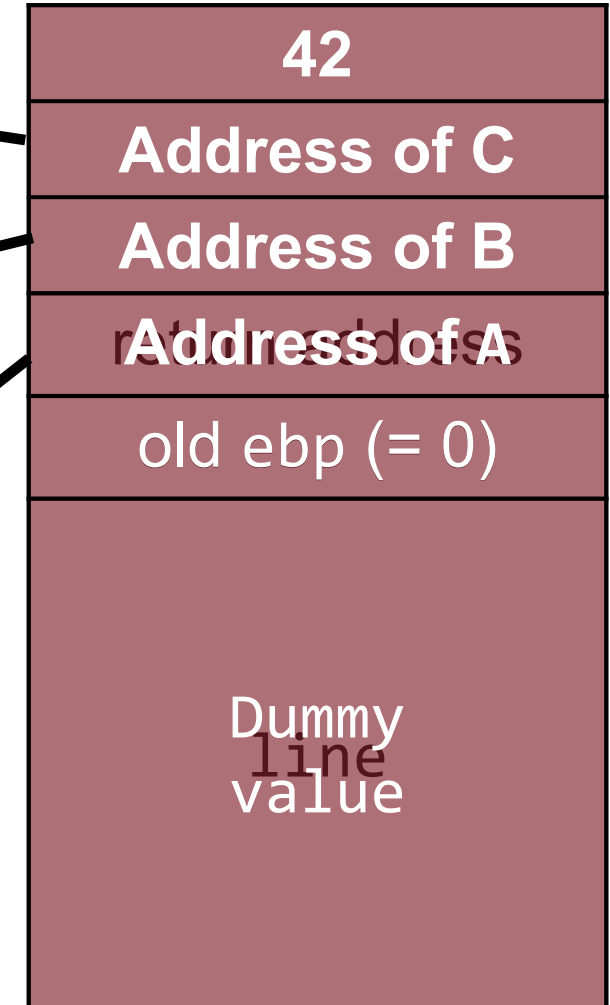
execute following instructions

```
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`



Return (ret) Chaining

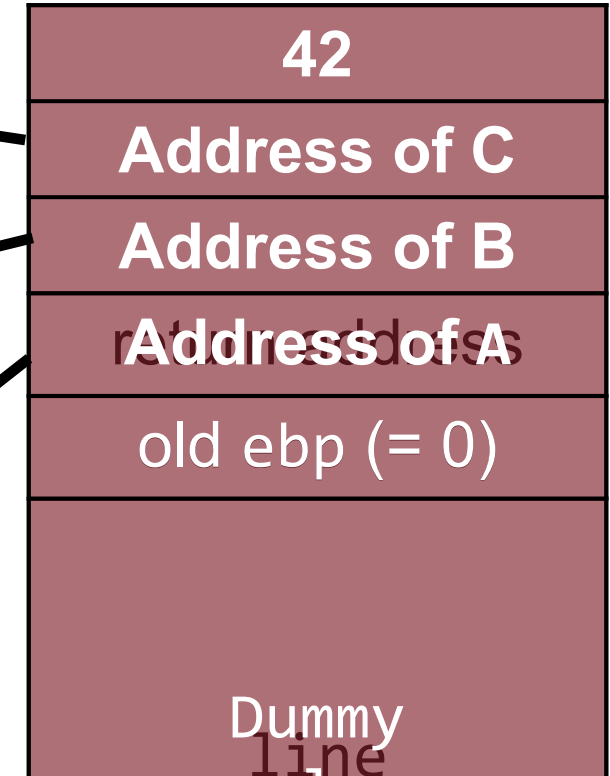
Attacker's goal:

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



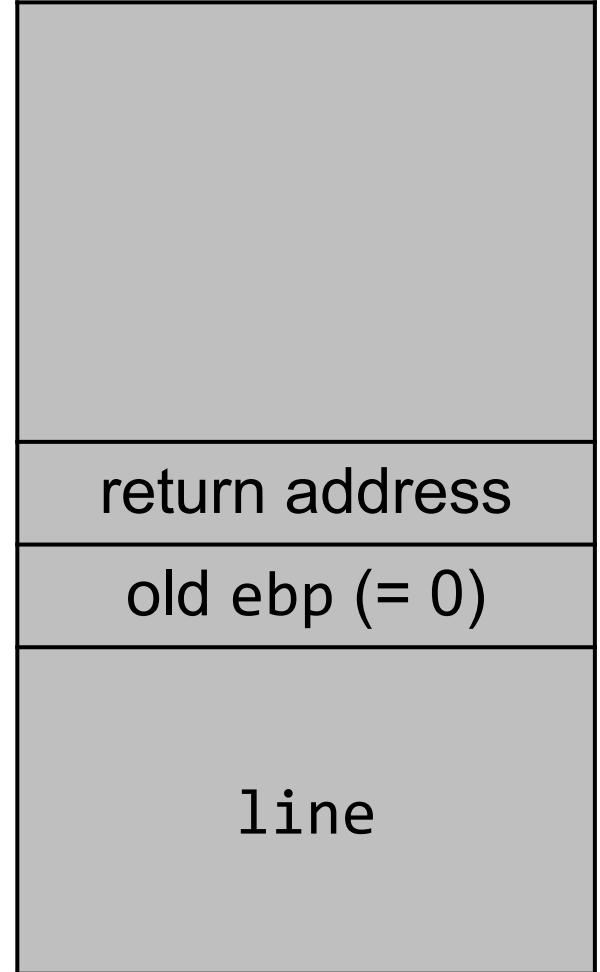
Goal: Modify ptr to be 0x42424242 with ROP

```
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***

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

Disassembling x86



eip



e8 05 ff ff ff

81 c3 59 12 00 00

call 8048330

add ebx,0x1259

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

Unintended ret Instruction



eip



e8 05 ff ff ff
81 c3 59 12 00 00

add eax, 0x81ffffff
ret

Unintended ret Instruction



eip



e8 05 ff ff ff
81 c3 59 12 00 00

add eax, 0x81ffffff
ret

Unintended ret Instruction

100

eip



e8	05	ff	ff	ff	
81	c3	59	12	00	00

add	eax,	0x81ffffff
ret		

Many Gadgets in Regular Binaries?



Also, program size may matter!

Larger code \Rightarrow More chance to get useful gadgets

Question

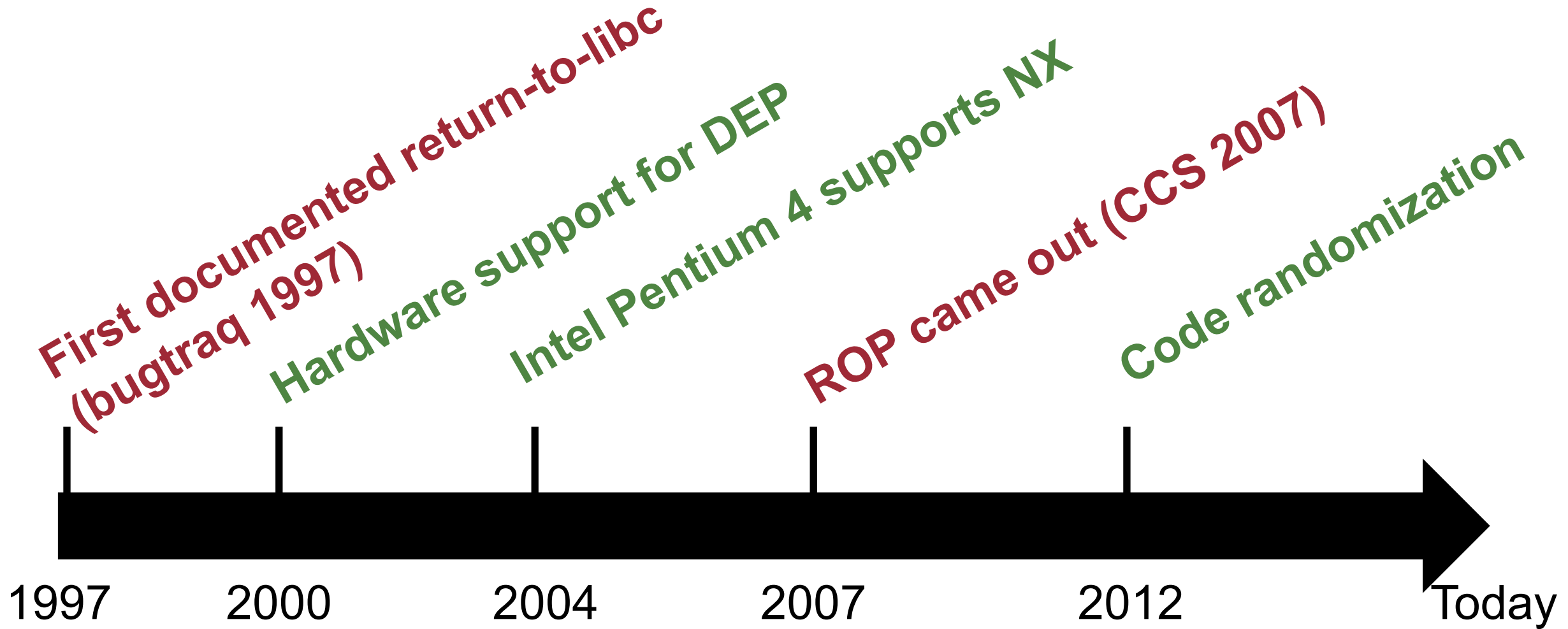


How can we mitigate code reuse attacks (ROP)?

Address randomization (ASLR)!
(next lecture)

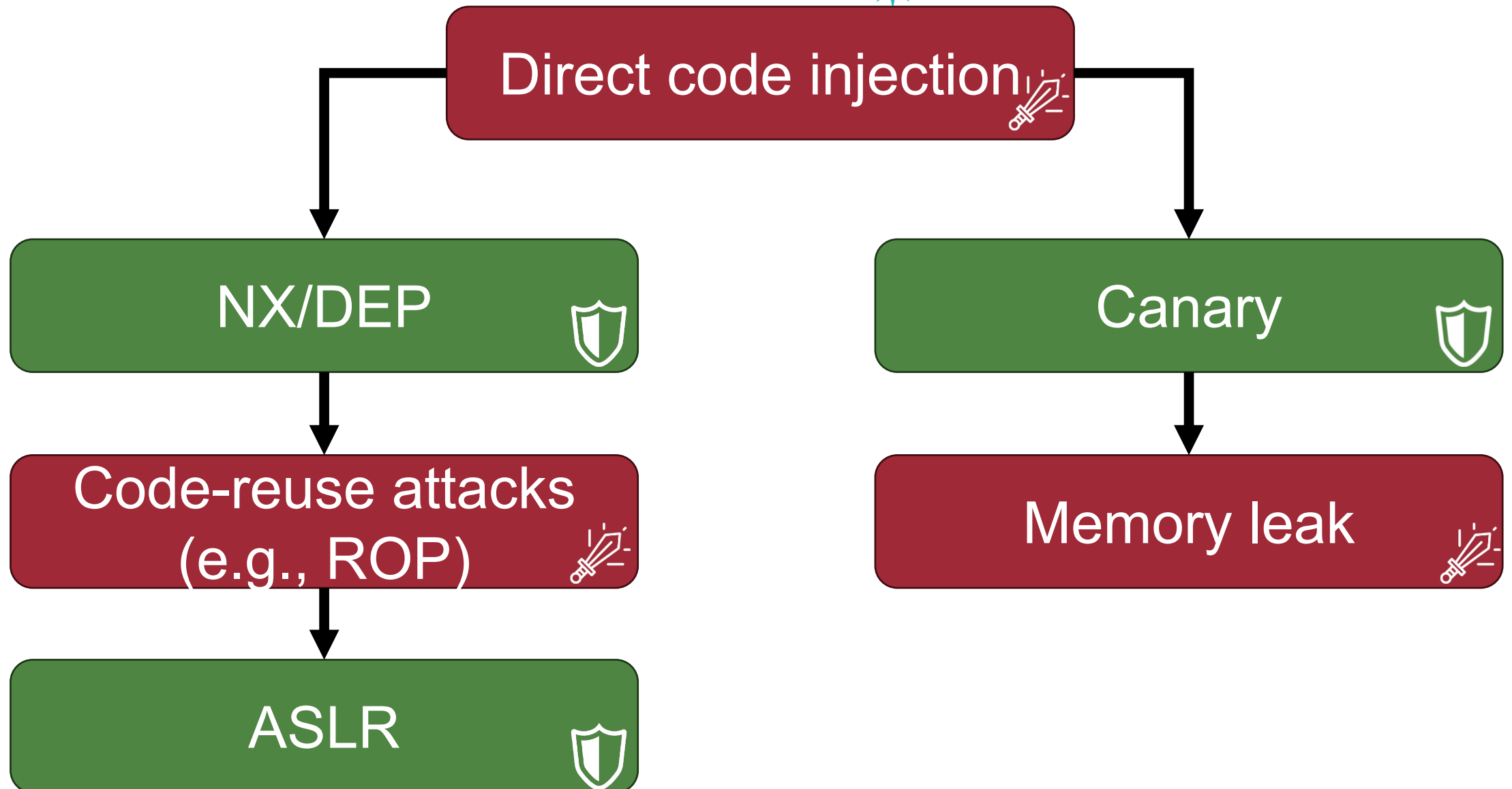
DEP and Code Reuse Attacks

103

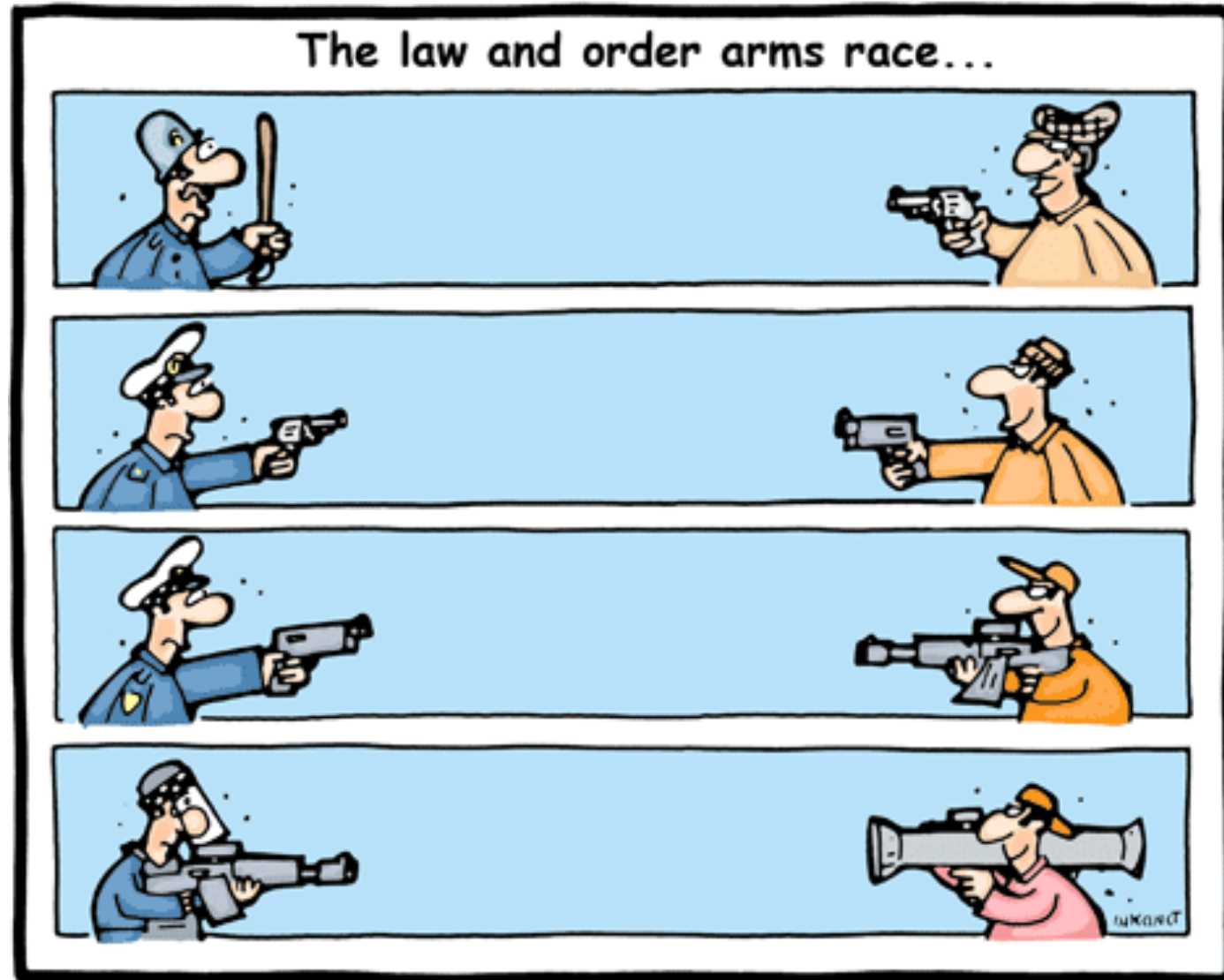


Control Hijack Attack / Defense So Far

104



Arms Race in Security



Summary



- Two mitigation techniques against control flow hijacks
 - Stack canary
 - NX (or DEP)
- Code reuse attacks allow an attacker to bypass DEP
- Many mitigation techniques are proposed for code reuse attacks, which will be covered next.

Question?