# CSE467: Computer Security

## 15. Canary & DEP

Seongil Wi

Department of Computer Science and Engineering

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

# How to defend against buffer overflows?

# Defense: 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?



Do NOT use C/C++!

C is the root of evil!

# Easy to Prevent Buffer Overflows!

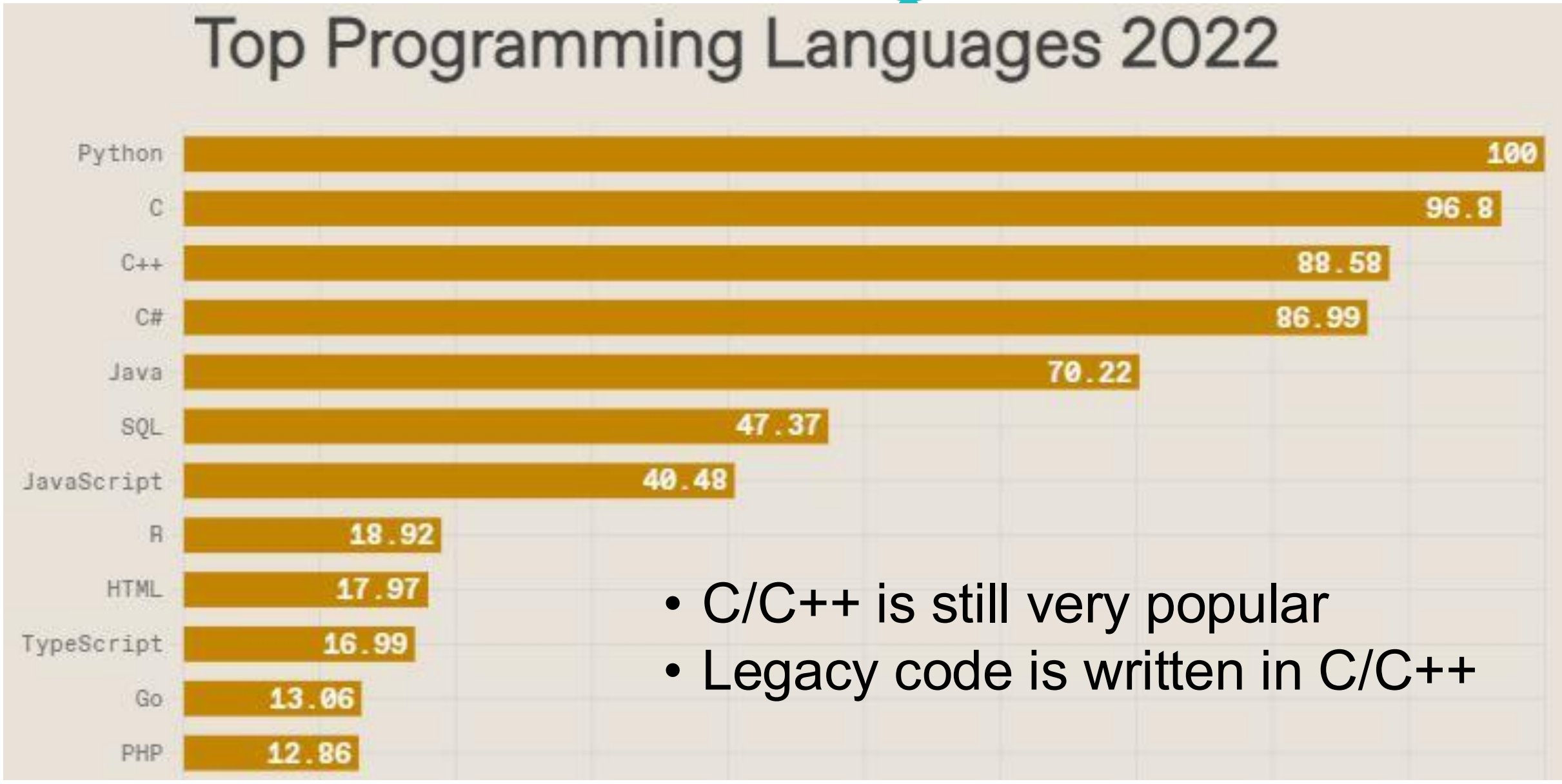


Have you ever seen buffer overflows in other safe languages such as F#, OCaml, Haskell, Python, etc.?

```
>>> 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 ...






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

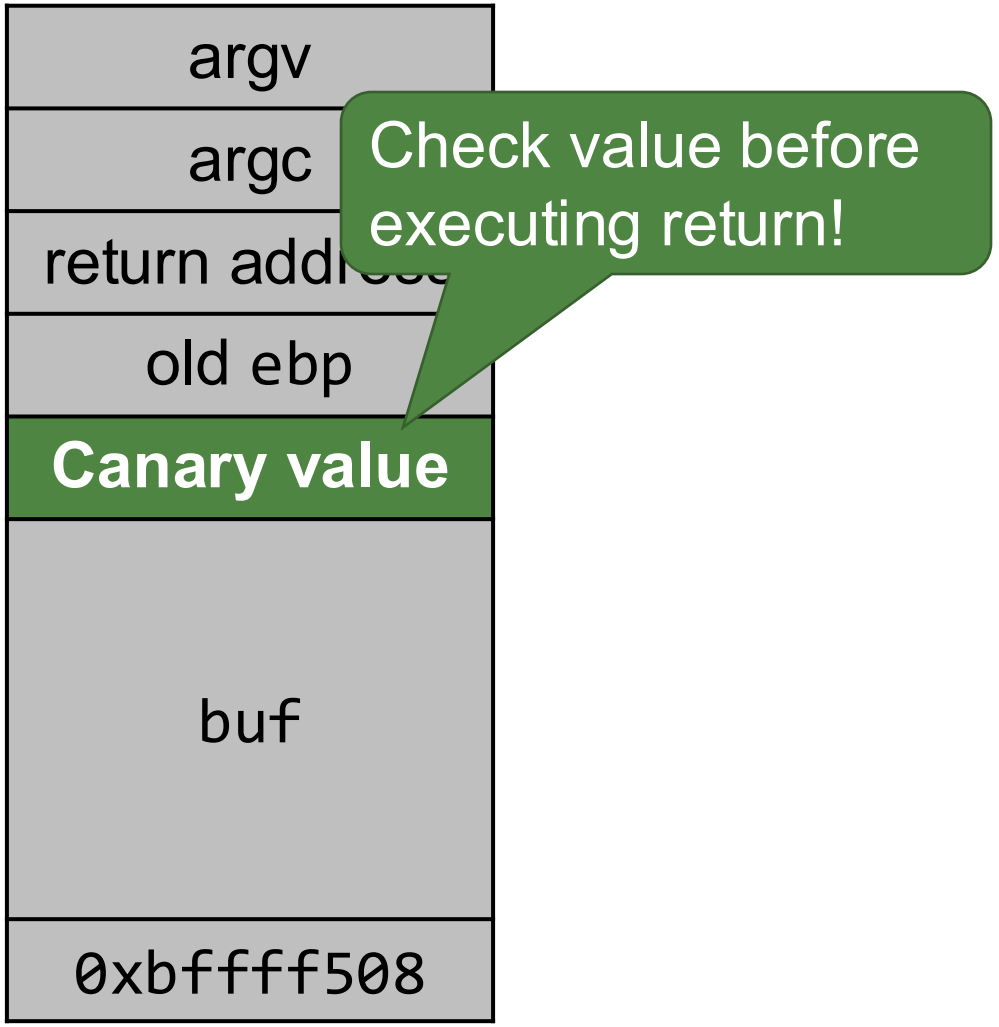
# Okay …



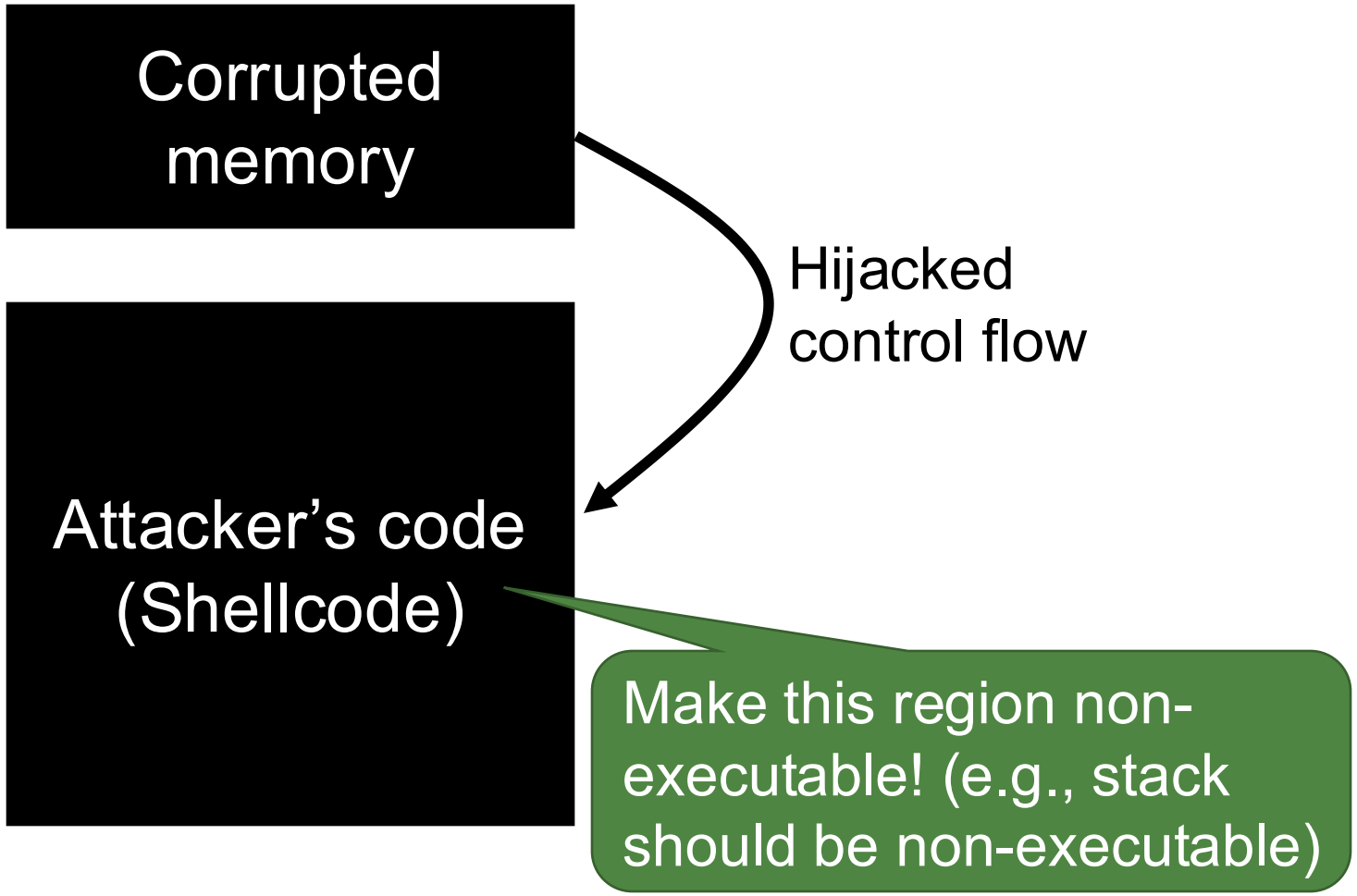
Let’s mitigate it then ☹

# Preview: Mitigating Memory Corruption Bugs

**Mitigation #1**: Canary




**Mitigation #2**: NX (No eXcute)



# Buffer Overflow Mitigation #1:
# Canary

Image from https://naturalworldpets.co.uk/blogs/pet-advice/guide-to-caring-for-canaries

# Canary in a Cole Mine



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

Image from https://www.forbes.com/sites/kionasmith/2019/12/31/the-canary-in-the-coal-mine-isnt-ancient-history

# Mitigating Buffer Overflows with Canary



- Early warnings of buffer overflows

- First introduced in 1998

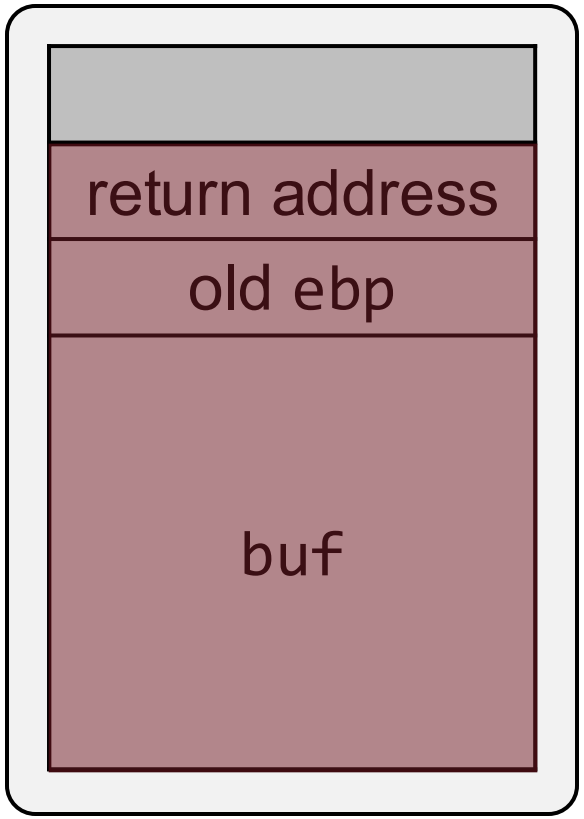
StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, ***USENIX Security 1998***

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

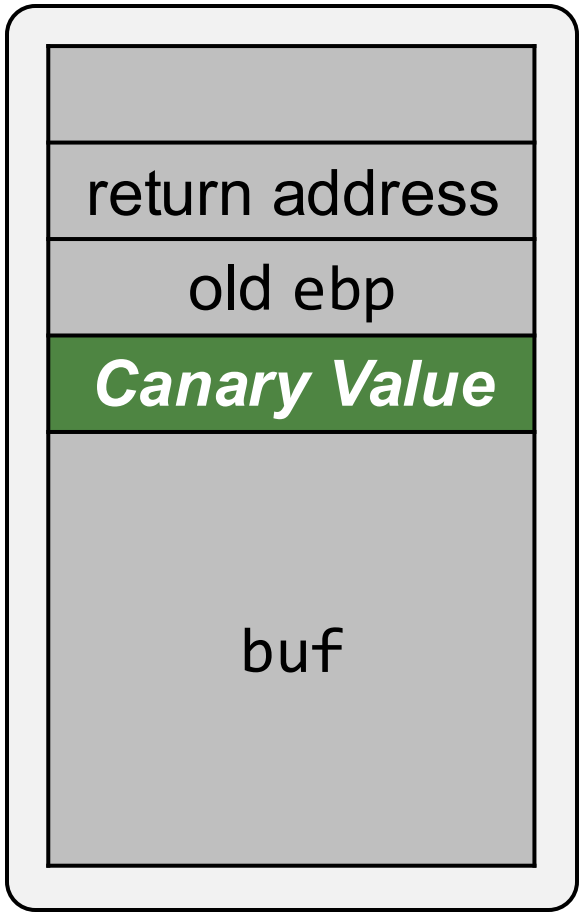
# Stack Canary (a.k.a. Stack Cookie)



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




Without stack canary

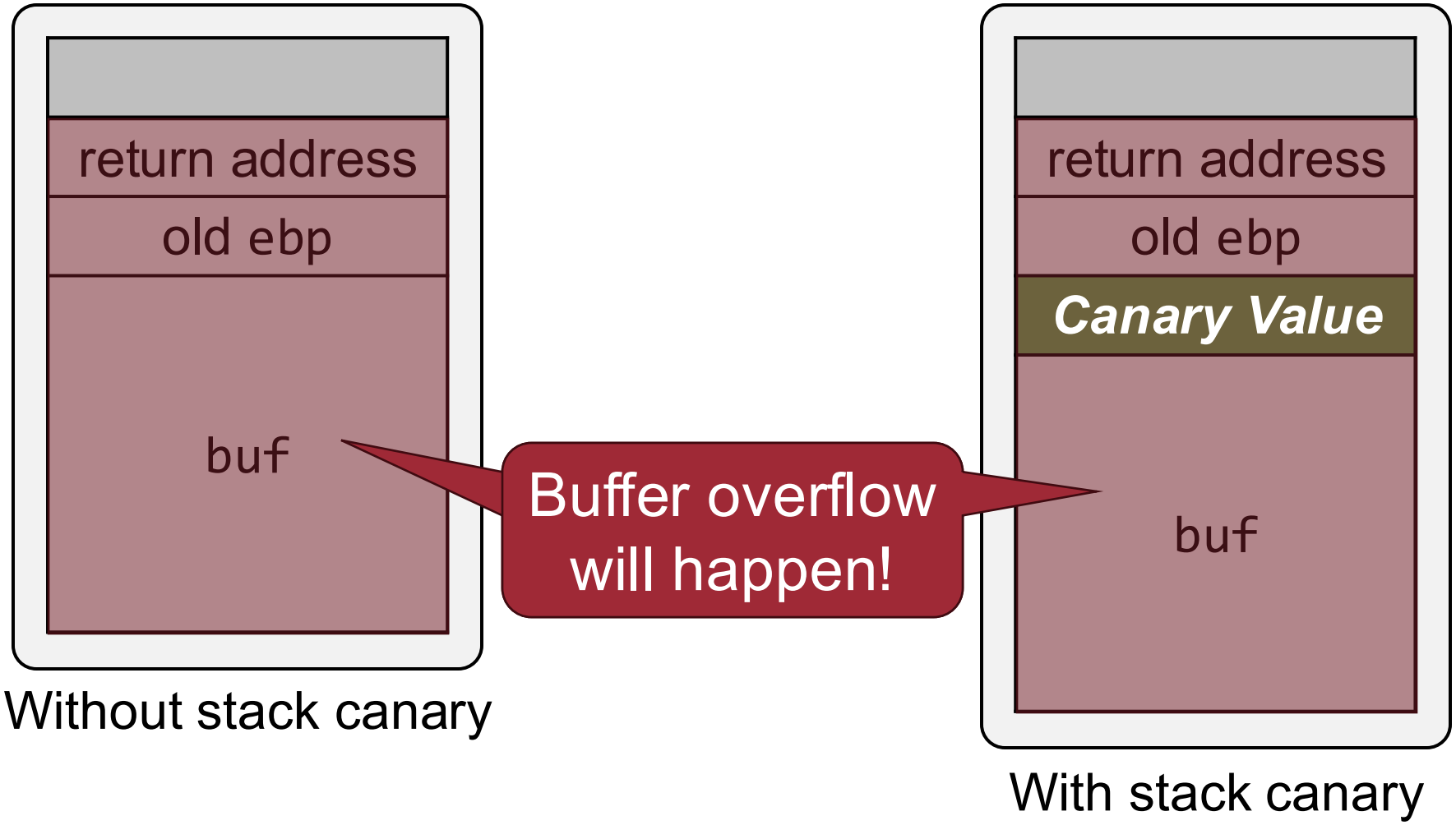



With stack canary

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



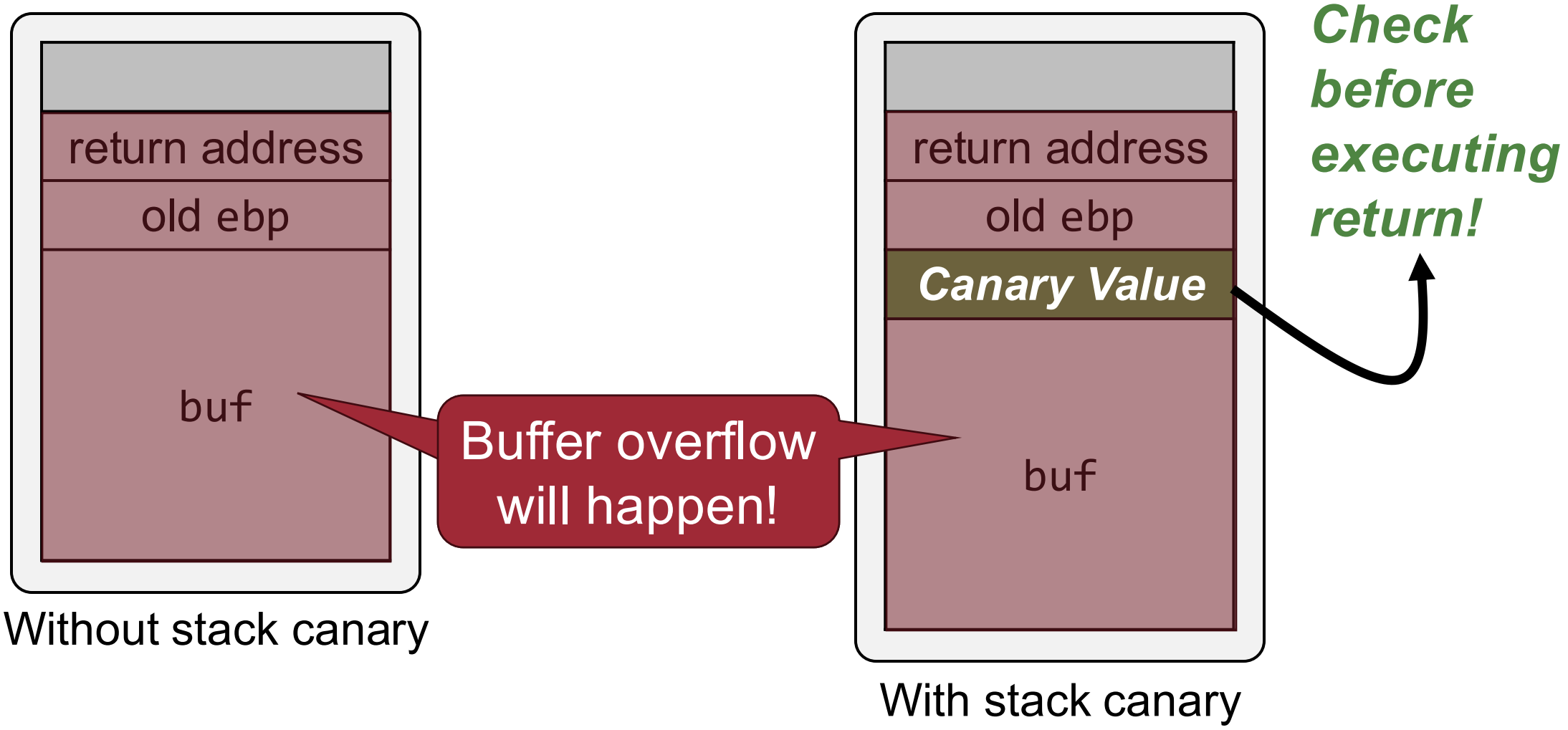
- Key idea: insert a checking value before the return address



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



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



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



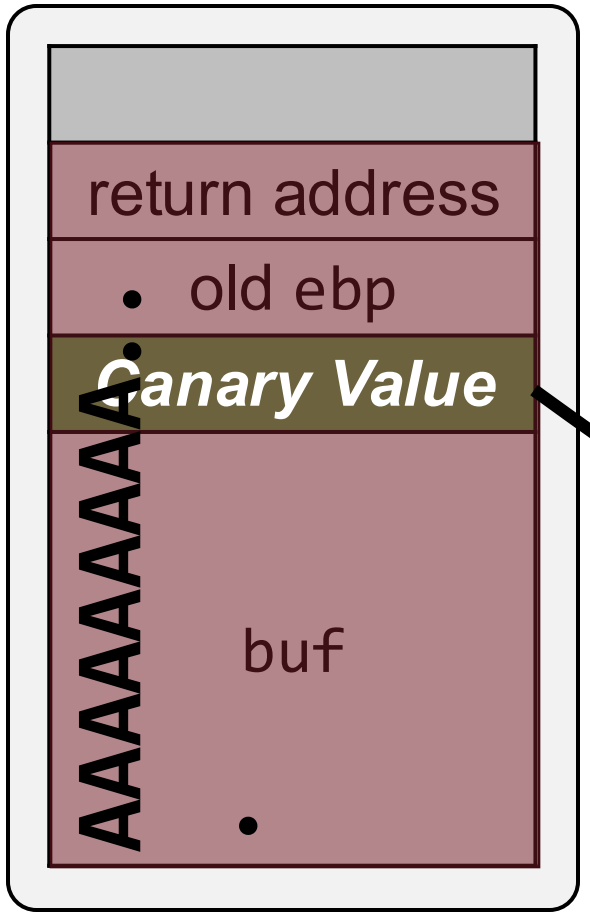
- Key idea: insert a checking value before the return address

Before executing return, check…

(Inserted canary value) ≠ (Current canary value)

***Canary Value*** ≠ **0x41414141**

Overflow is occurred!
Stop the program



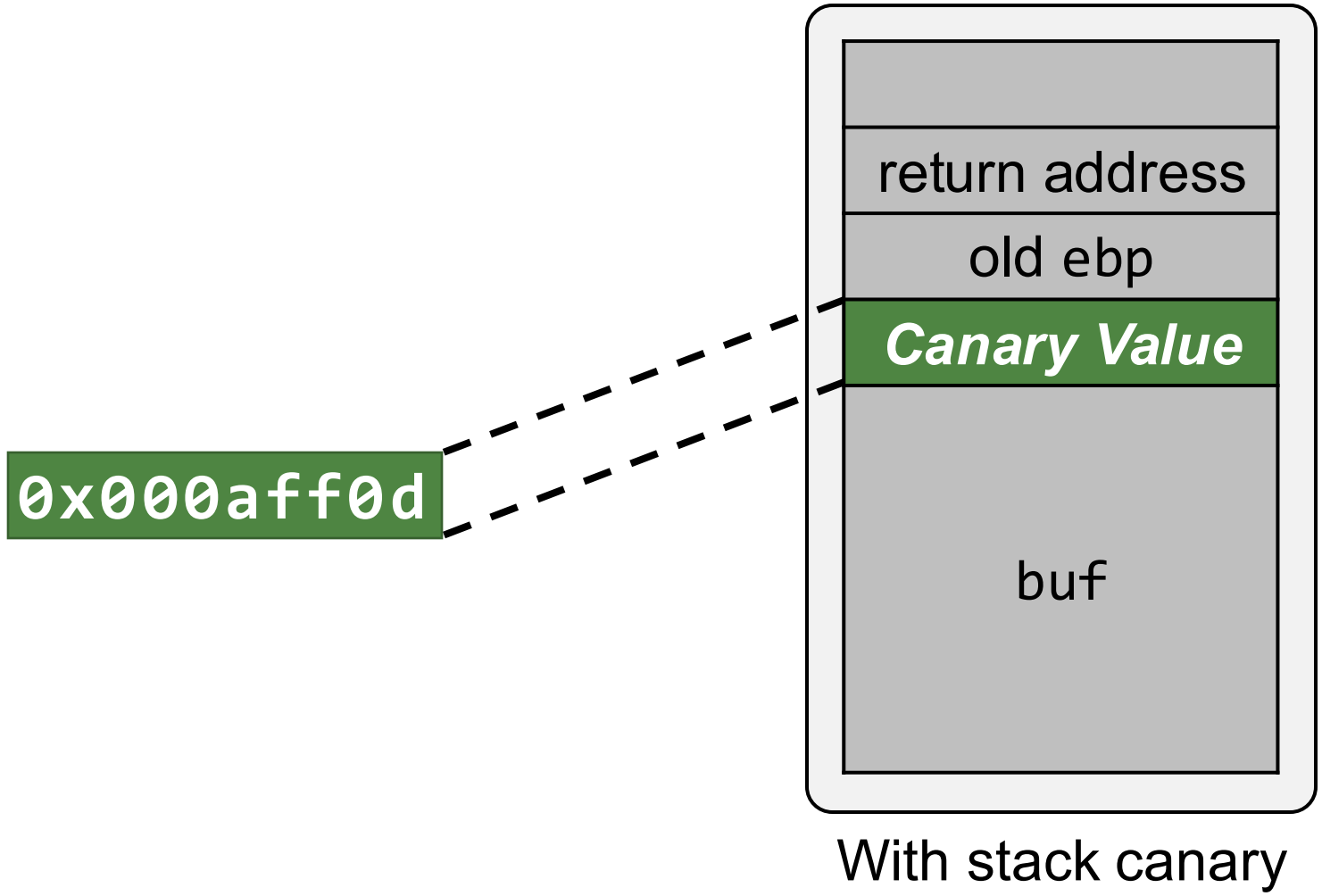

With stack canary

***Check before executing return!***

# StackGuard (1998)



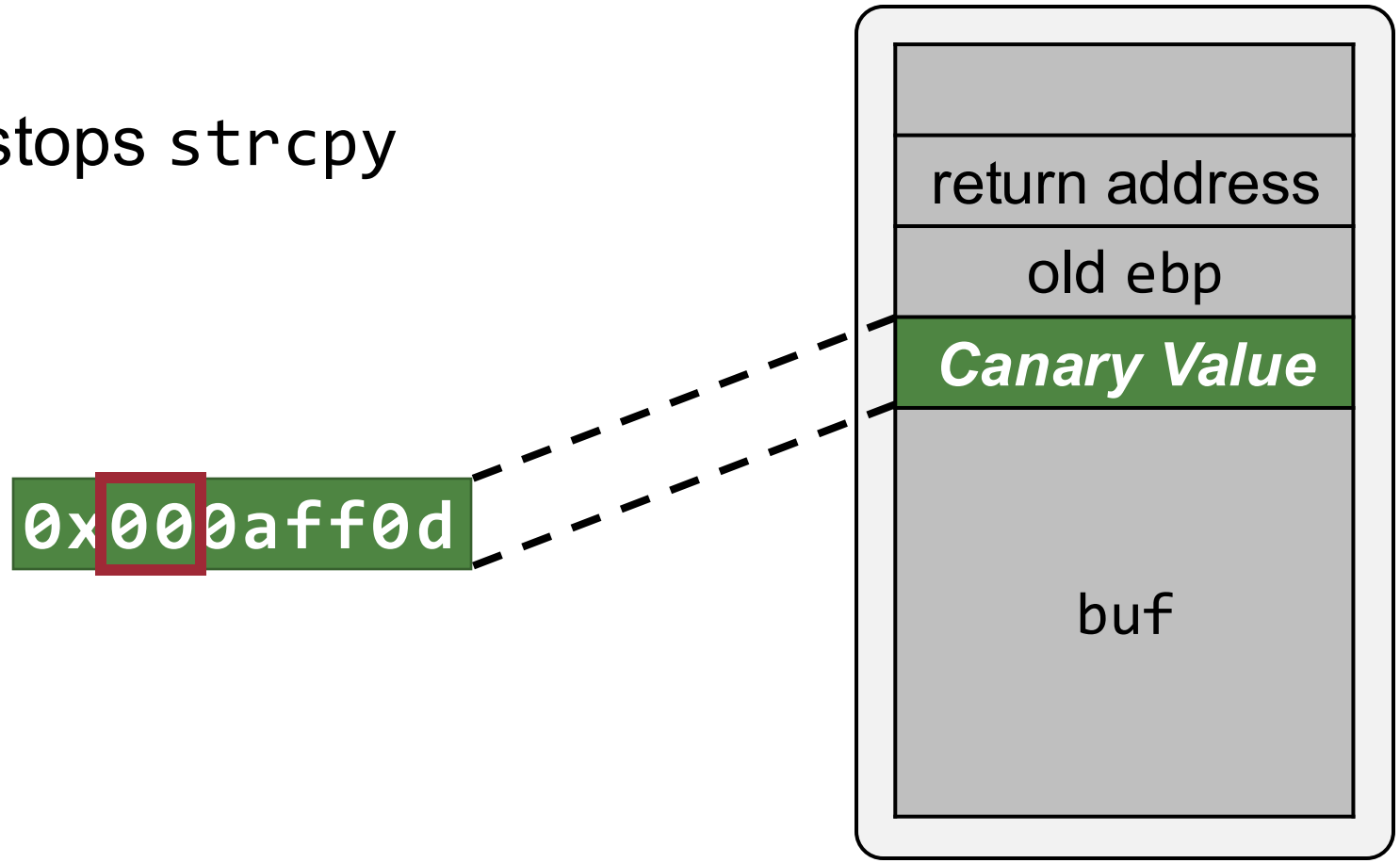
- Uses a constant canary value `0x000aff0d`



# StackGuard (1998)

- Uses a constant canary value `0x000aff0d`

  - ✓ `0x00` stops `strcpy`

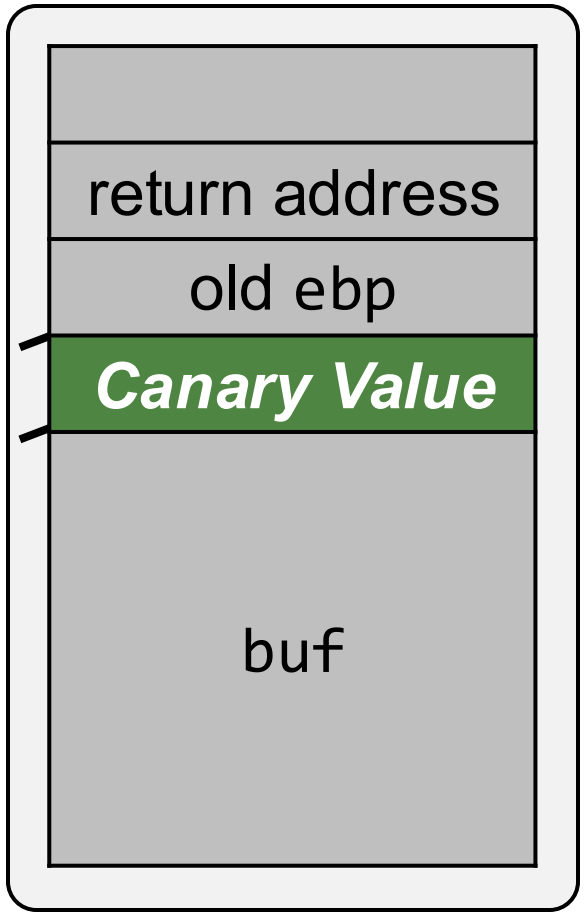



With stack canary

# StackGuard (1998)

- Uses a constant canary value `0x000aff0d`

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

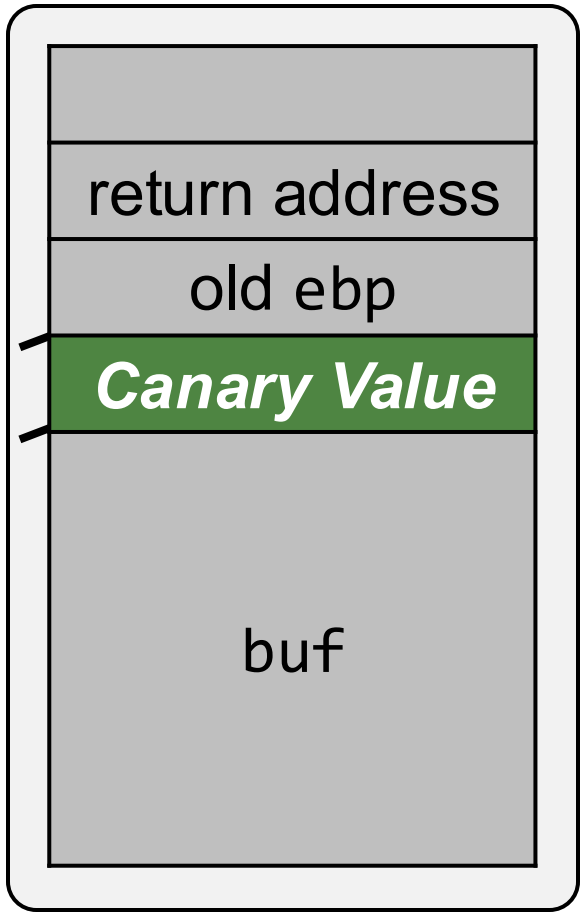




With stack canary

# StackGuard (1998)

- Uses a constant canary value `0x000aff0d`

✓ `0x00` stops `strcpy`
✓ `0x0a` and `0x0d` stop `fgets`
✓ `0xff` stops `EOF` checks


With stack canary

# Problem of Using a Constant Canary Value



memcpy?

# Problem of Using a Constant Canary Value



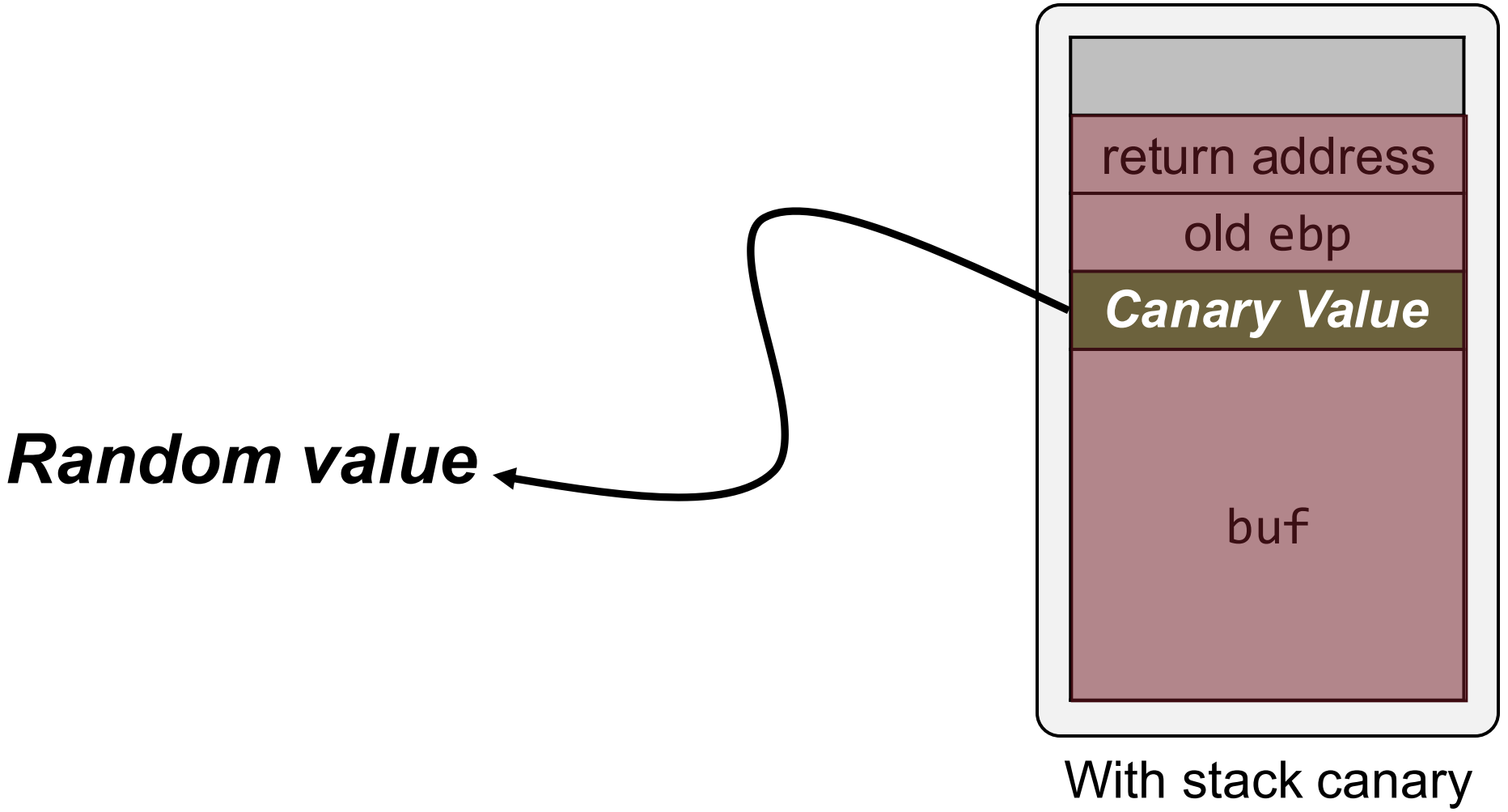
```
memcpy(void dest, void src, size_t n)
```

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

# Random Canaries

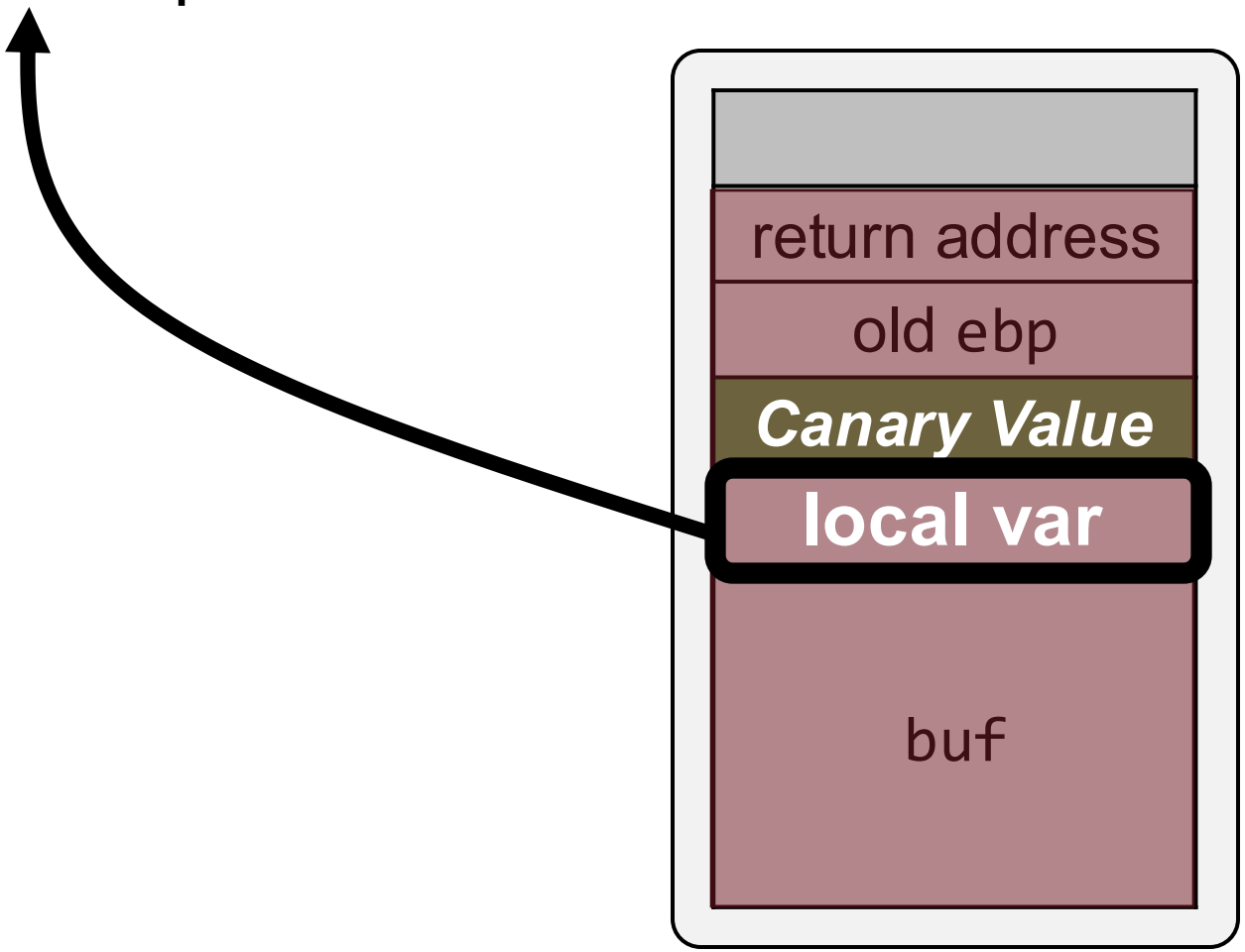


- Pick a ***random value*** at process initialization, put it on the stack



# 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

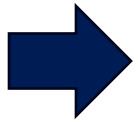
# GCC Stack Canary Implementation



```
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
```

Without stack canary

gcc -fno-stack-protector



```
804844b: push ebp
804844c: mov  ebp, esp
804844e: sub  esp,0 x108
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
```

With **stack canary**

gcc -fstack-protector

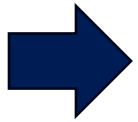
# GCC Stack Canary Implementation



```
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
```

Without stack canary

gcc -fno-stack-protector



```
804844b: push ebp
804844c: mov  ebp, esp
804844e: sub  esp,0 x108
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
```

With **stack canary**

gcc -fstack-protector

# GCC Stack Canary Implementation



Random canary value at gs:0x14

```
                                    esp
                                    x108
                                    eax,DWORD PTR [ ebp+0x8 ]
8048457: mov                        RD 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?

```
                                esp
                                x108
                                eax,DWORD PTR [ ebp+0x8 ]
8048457: mov                    RD 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



```
804844b: push ebp
804844c: mov  ebp, esp
804844e: sub  esp,0 x108
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 ]
                          bp-0x104 ]
804847a: add              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
```

Get current canary value from stack

Compare to the original canary value

Jump to the `leave` instruction if equal

With **stack canary**

`gcc -fstack-protector`

# GCC Stack Canary Implementation



```
804844b: push ebp
804844c: mov  ebp, esp
804844e: sub  esp,0 x108
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
```

With **stack canary**

gcc -fstack-protector

***Stack smashing detected!***
***(terminated)***

# Demo

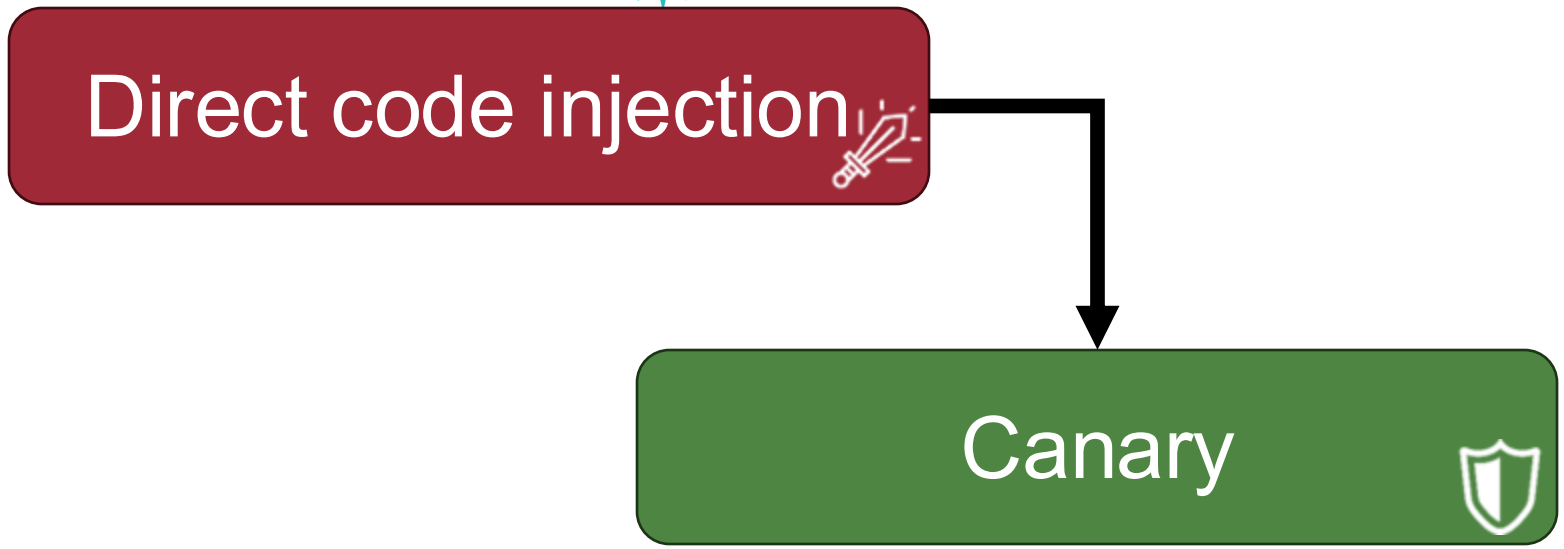
# GCC Canary Implementation



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

# Control Hijack Attack / Defense So Far

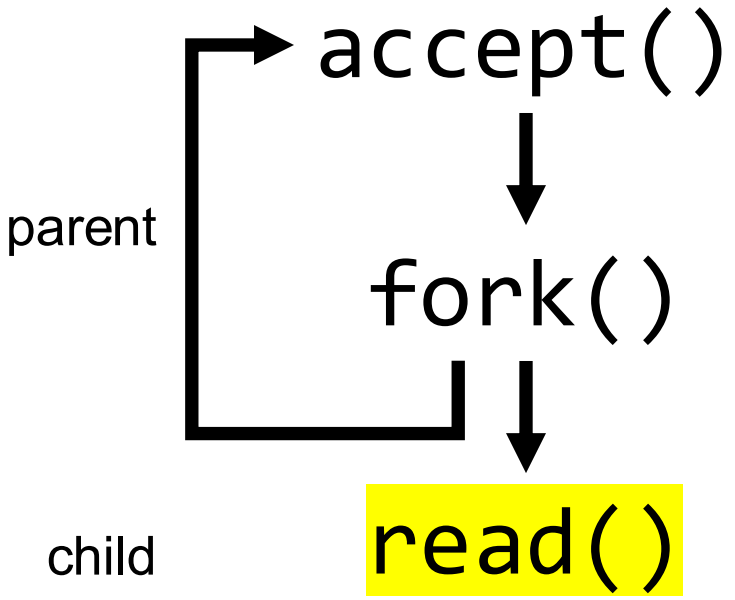




# Bypassing Canary Protection

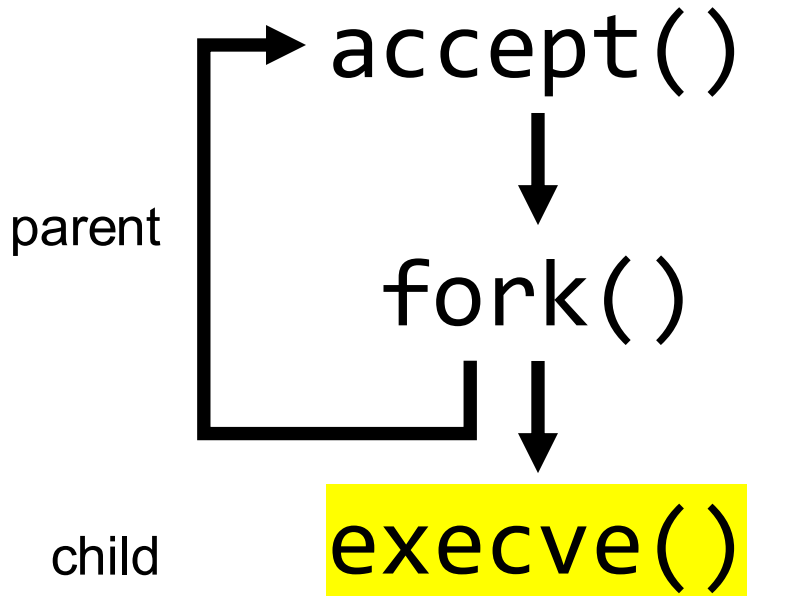
# Reused Canary Value

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




Server Type #1

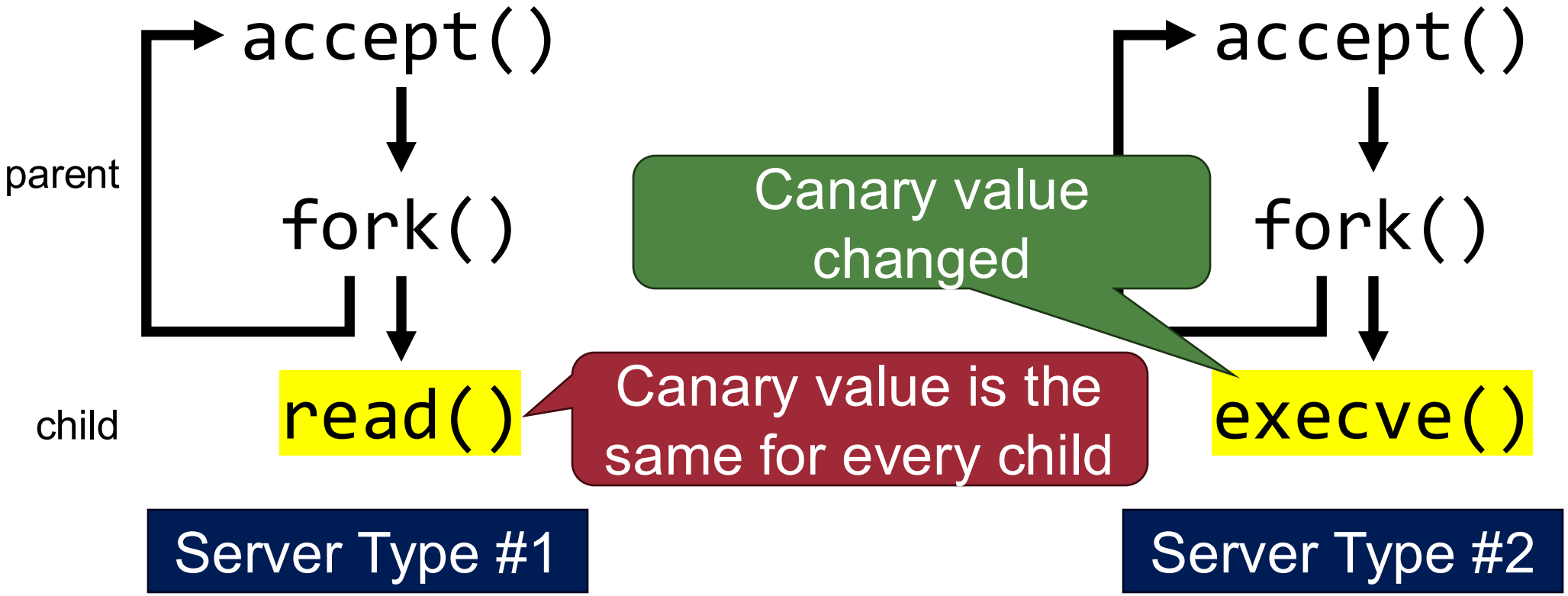
e.g., OpenSSH does this




Server Type #2

# Reused Canary Value

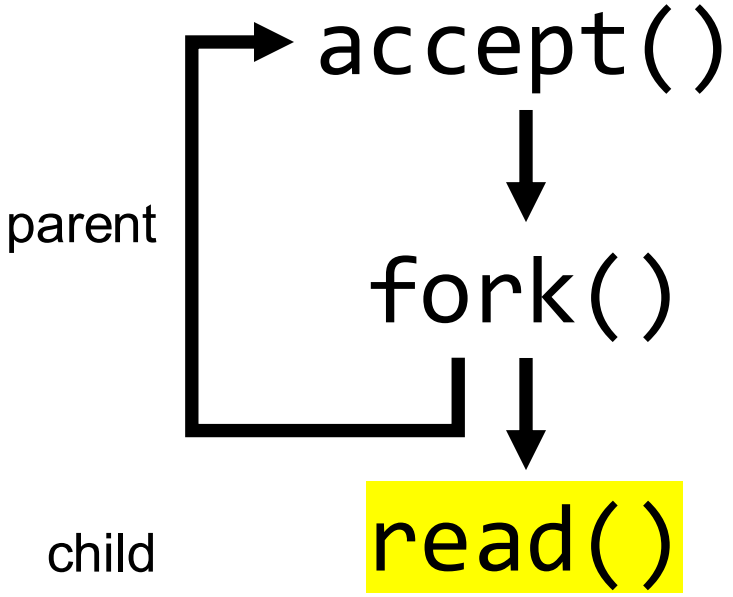
- Uses a random canary value for ***every process creation***



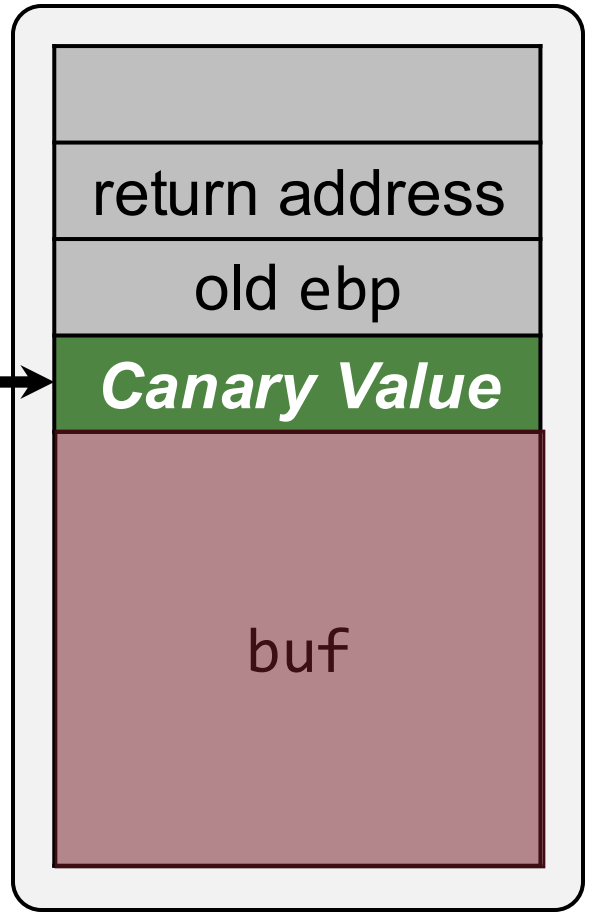

e.g., OpenSSH does this

# Attack #1: Byte-by-Byte Brute Forcing






Random canary:
0x429af70c

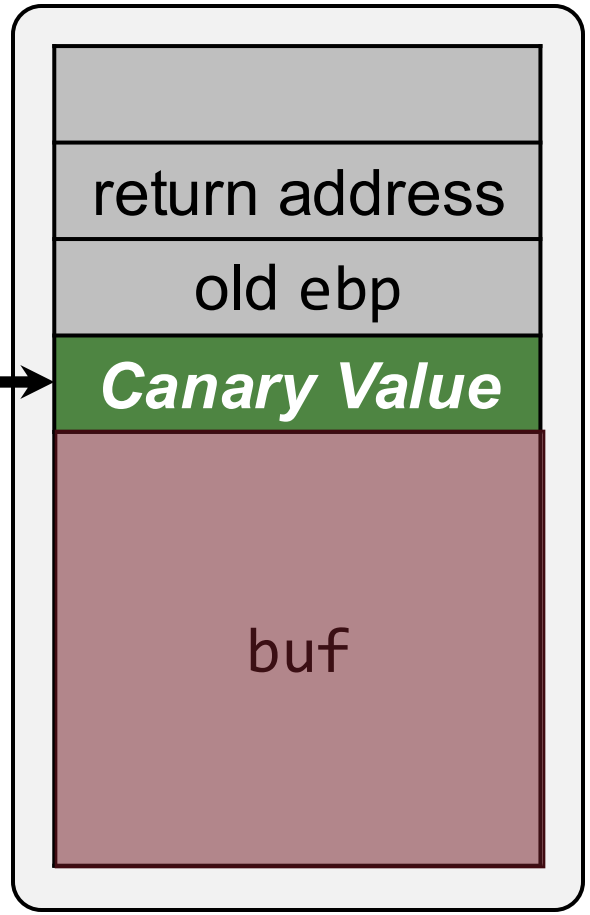



With stack canary

# 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`

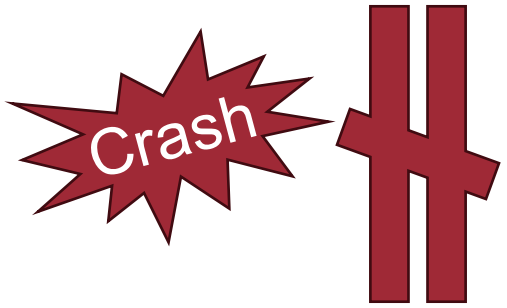



With stack canary

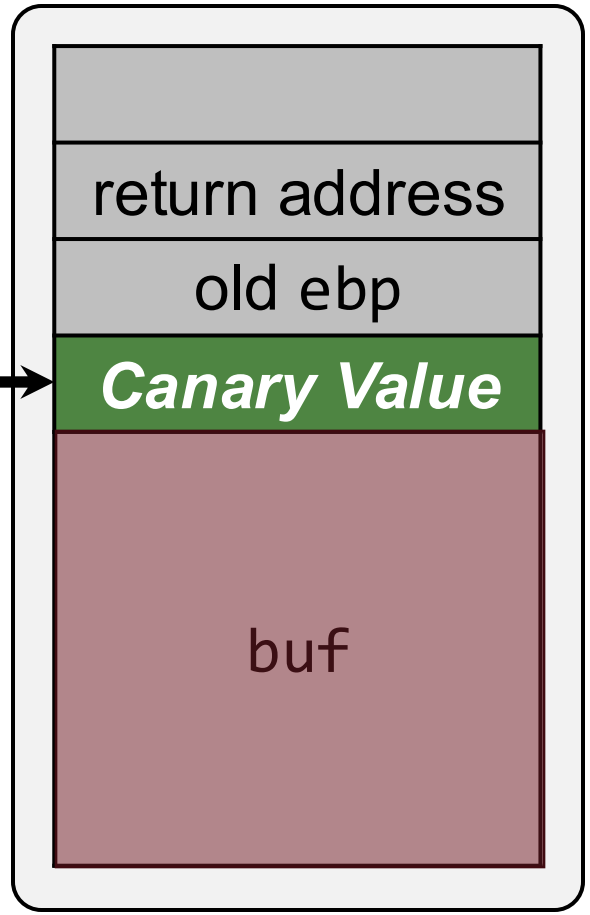
# Attack #1: Byte-by-Byte Brute Forcing

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`

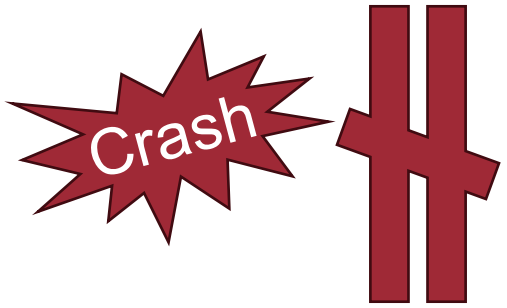




With stack canary

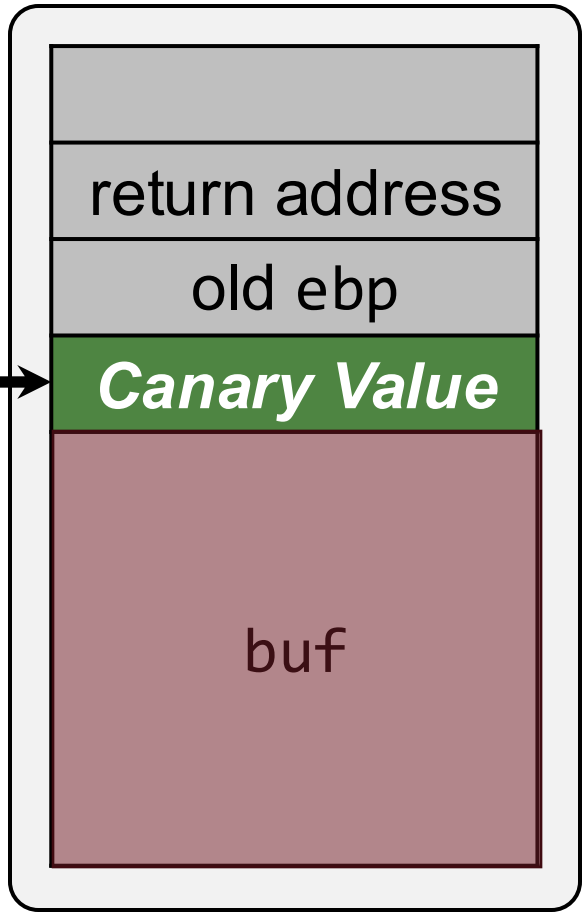
# Attack #1: Byte-by-Byte Brute Forcing

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`

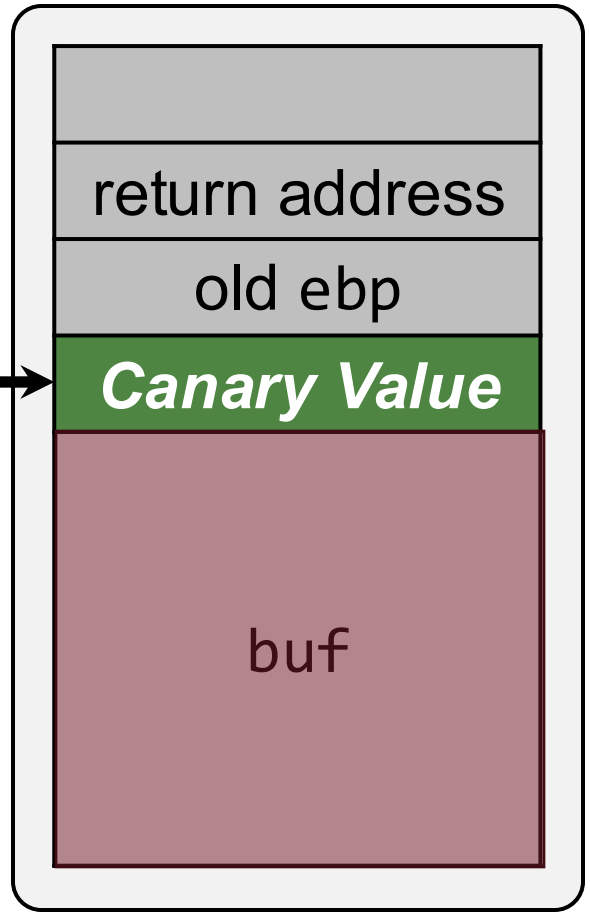




With stack canary

# 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`




With stack canary

**67th try**: insert `\x42`

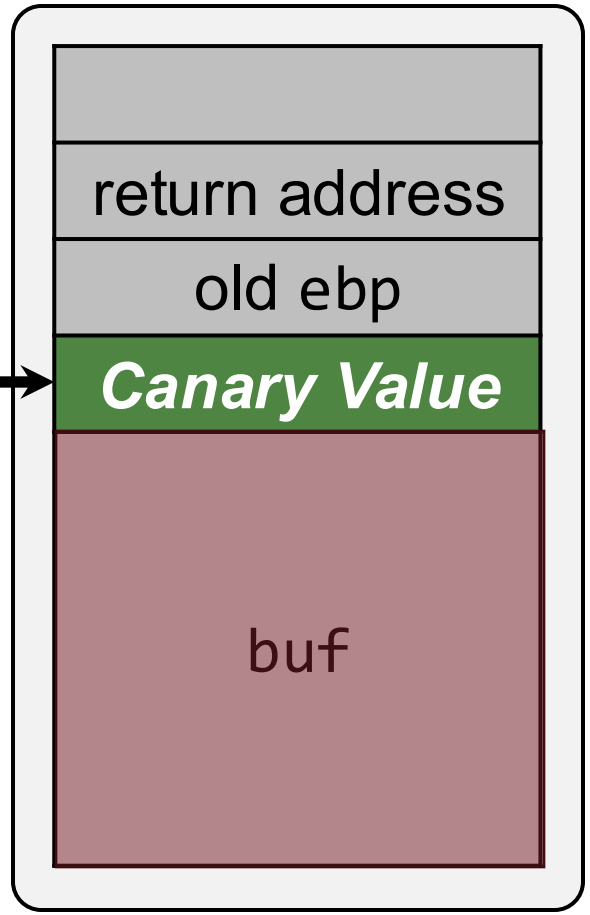
# Attack #1: Byte-by-Byte Brute Forcing

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

Random canary:
`0x429af70c`

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

**67th try**: insert `\x42`


With stack canary

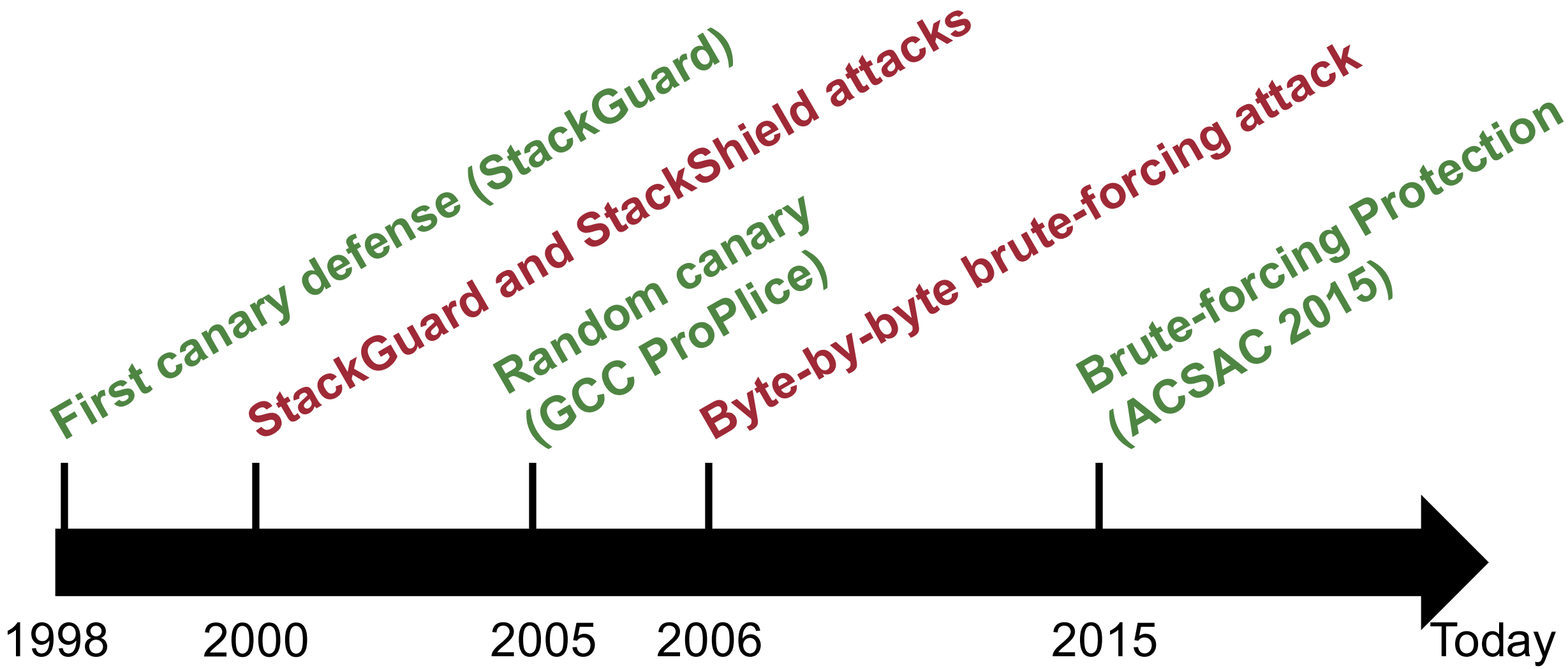
# Protecting Canary Brute-Forcing Attack



(Optional Reading)

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

# Canary Attack and Defense Timeline



# Attack #2: Leaking Canary Value

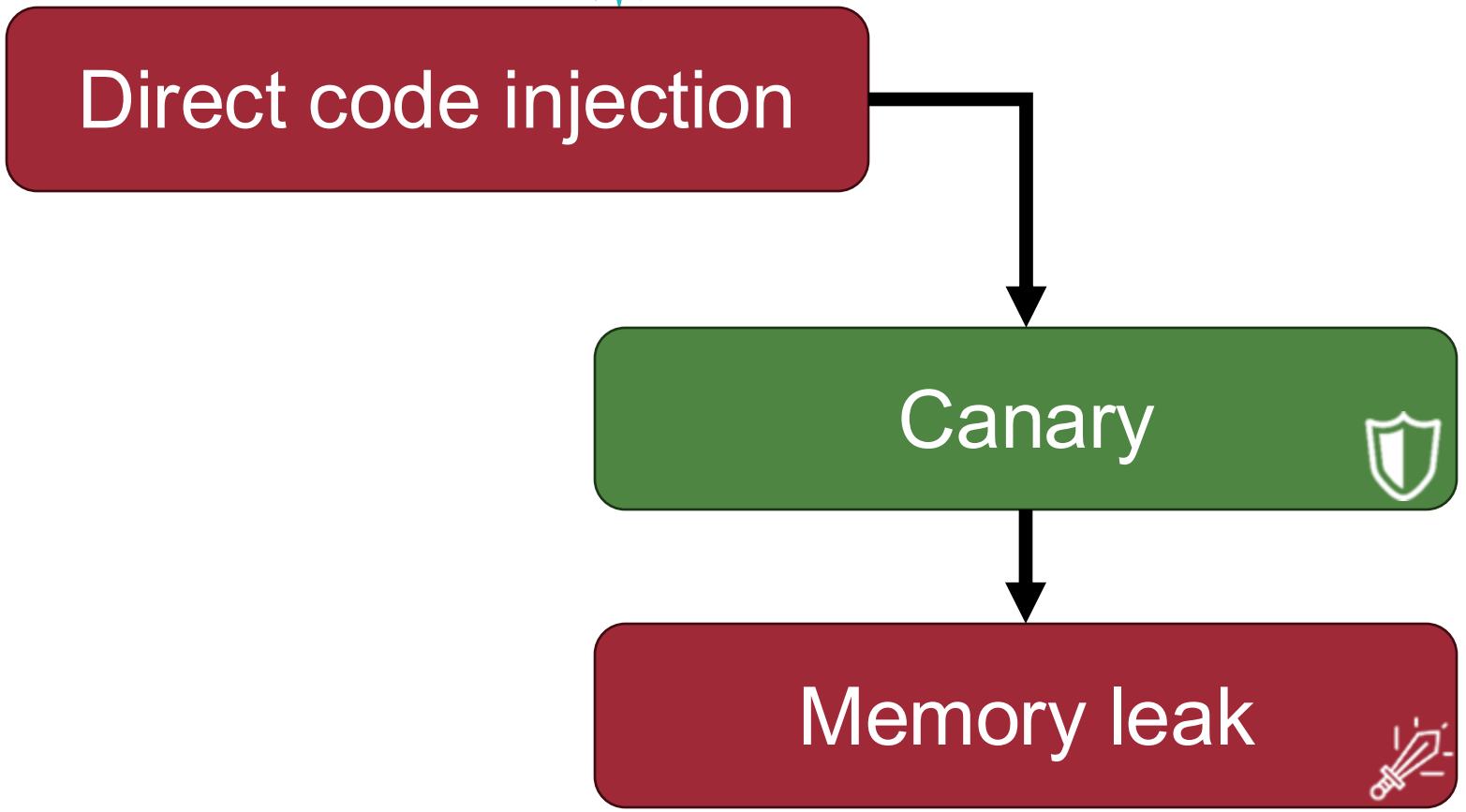


- If there is another vulnerability that allows us to ***leak*** stack contents, then we can easily bypass the canary check

- Canary is inherently vulnerable to *format string attacks*

# Control Hijack Attack / Defense So Far





# 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

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

# W ⊕ X (Write XOR eXecute) Policy

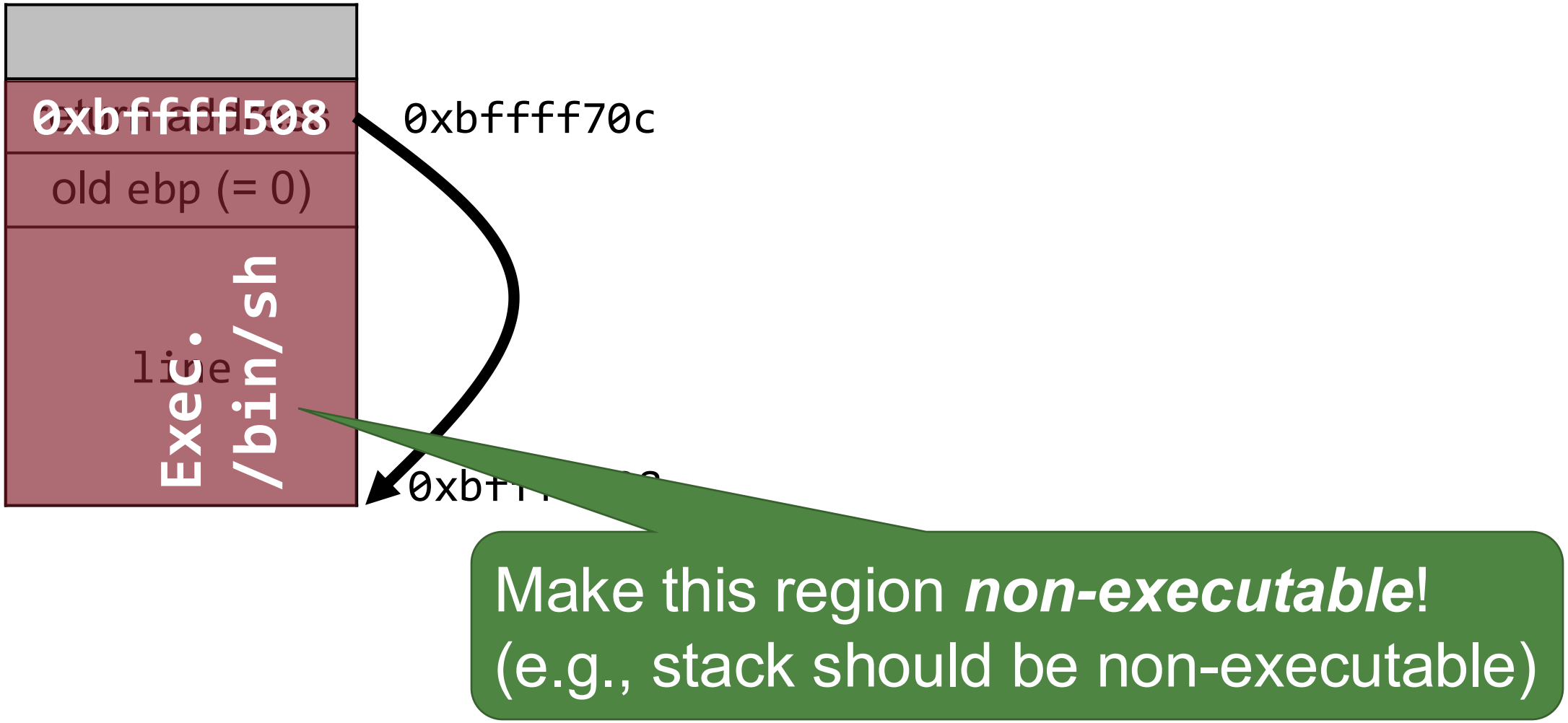


On Linux, it is called W ⊕ X

- 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





# DEP on Stack using execstack

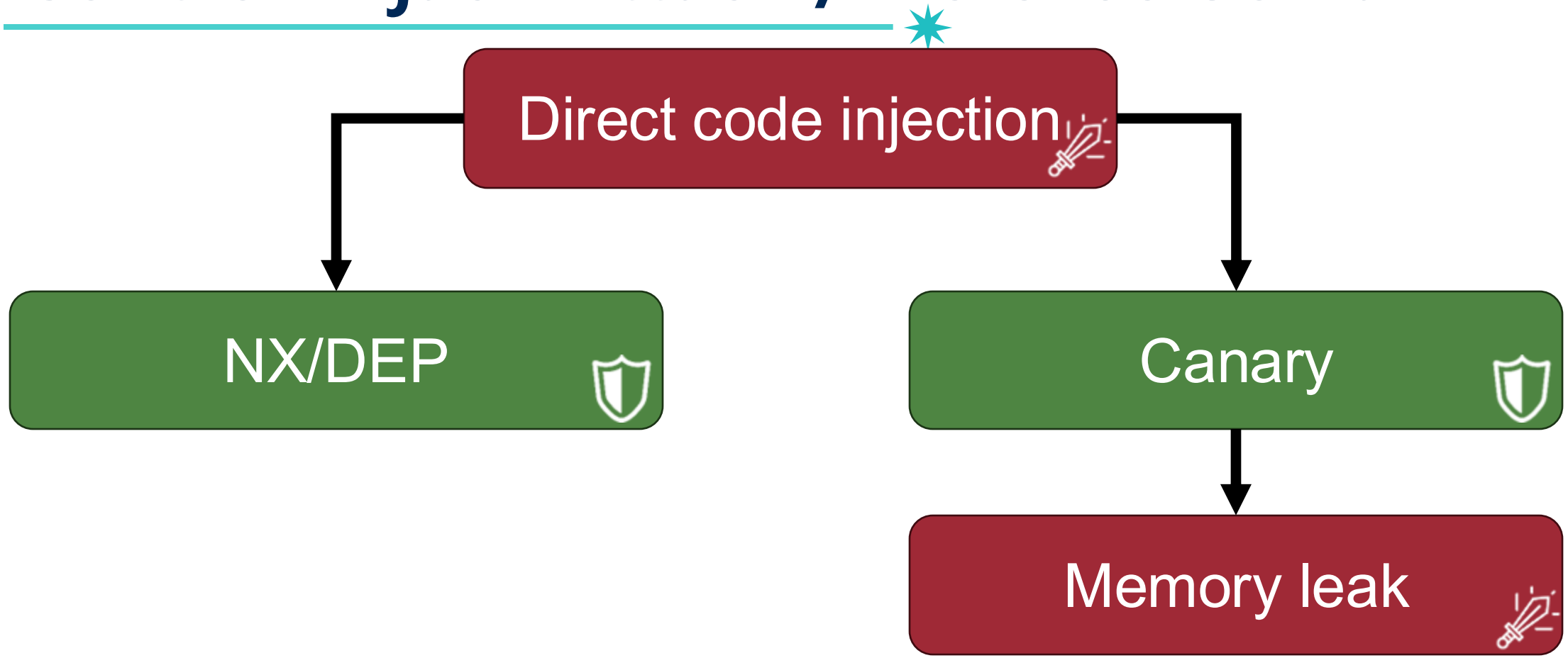


- Tool to set, clear, or query NX stack flag of binaries

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

# Control Hijack Attack / Defense So Far



# But,

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

Any other ways to exploit buffer overflows?

***Next topic!***

# Summary



- Two mitigation techniques against control flow hijacks
  - Stack canary
  - NX (or DEP)

# Question?