Mitigations and Secure Coding

Security 1 2018-19

Università Ca' Foscari Venezia

www.dais.unive.it/~focardi

secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Stack overflow

A buffer overflow **occurring on the stack,** also known as *stack smashing*

Right after the local variables, the stack contains

- The old base pointer (EBP)
- The return address

A stack overflow can overwrite these control data to run **arbitrary code**

Mitigations

A number of **mitigations** have been added in compilers and operating systems to make buffer overflow exploitation harder

- Non eXecutable stack (**NX**)
  Support: hardware (NX bit)/operating system
- Address space layout randomization (**ASLR**)
  Support: operating system
- Stack protector (**canary**)
  Support: operating system

Limitations of NX

NX prevents execution of injected code on the stack

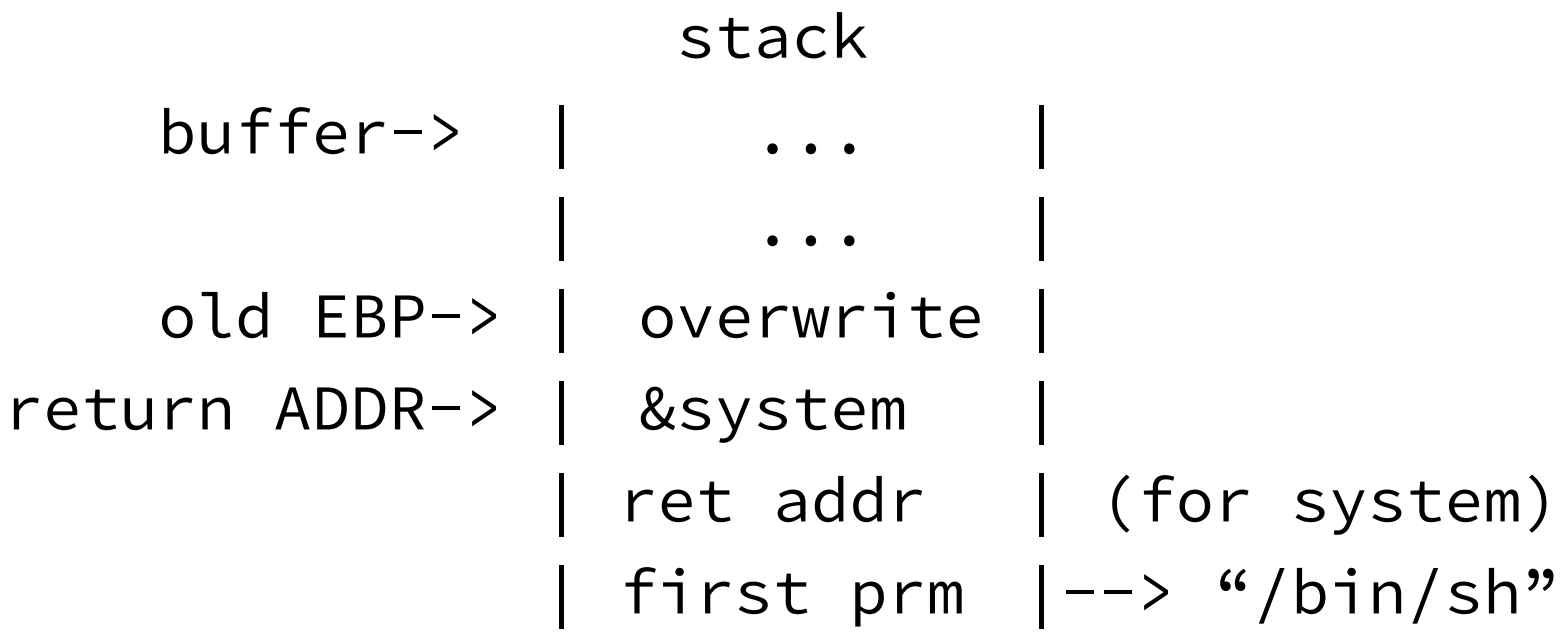
(Programs might **disable** it if they need to execute code on the stack)

Even with NX enabled, an attacker can:

- Return to **program code**
- Return to **library code**

In general, NX does not prevent **returning to code** in segments that are (necessarily) executable!

Example: return to `system`



NOTE: `system` "thinks" it has been called and looks for its parameter on the stack ("/bin/sh")

Address space layout randomization

ASLR randomizes the address space of

- stack
- library functions

This requires **brute-forcing** to get useful addresses

Limitations:

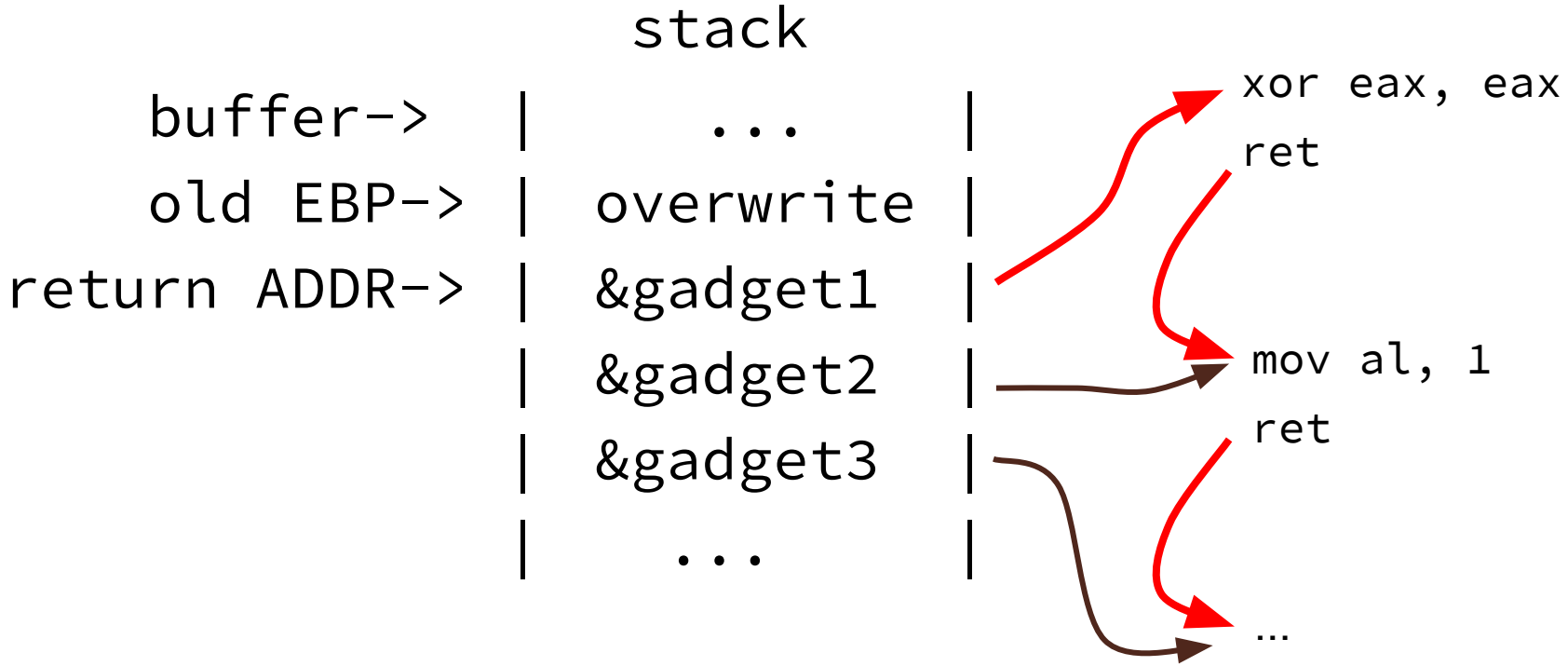
- It does NOT prevent jumping to **program code** (for example ROP, described later)
- If addresses are **leaked** (e.g. recent timing side-channels attacks) it becomes void

Examples

1. Brute-forcing with ASLR
2. Jumping to a particular instruction in a program

Return Oriented Programming (ROP)

ROP "composes" a shellcode by putting together small pieces of code (*gadgets*) that ends with `ret`



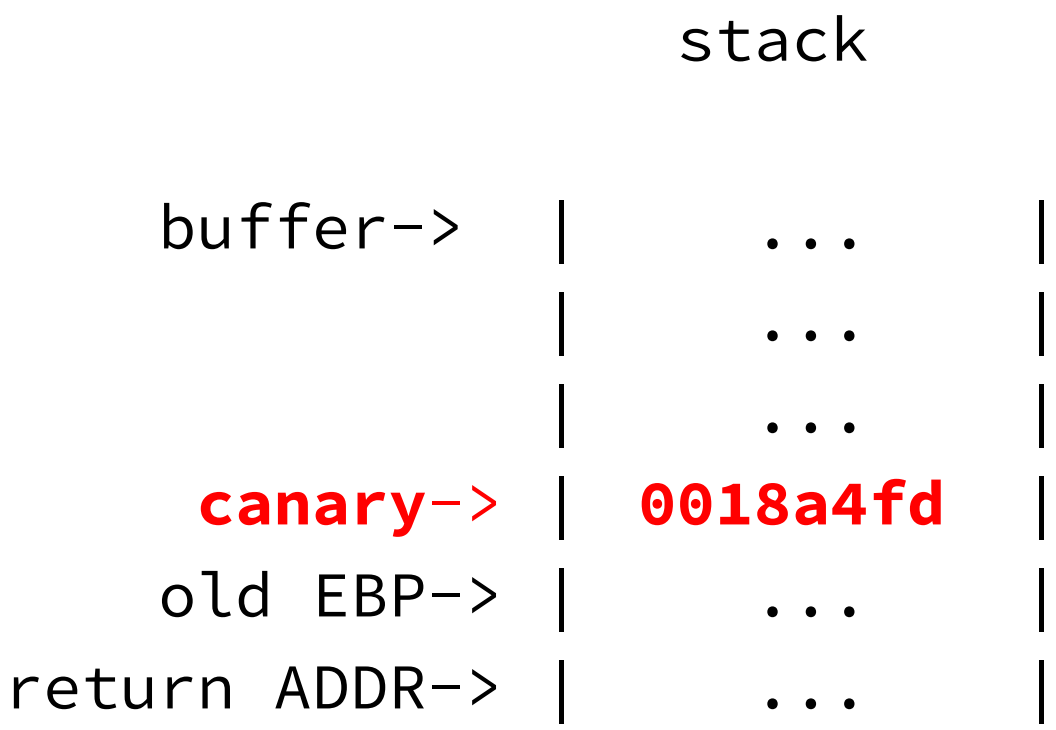
Stack protector (canary)

Stack protector

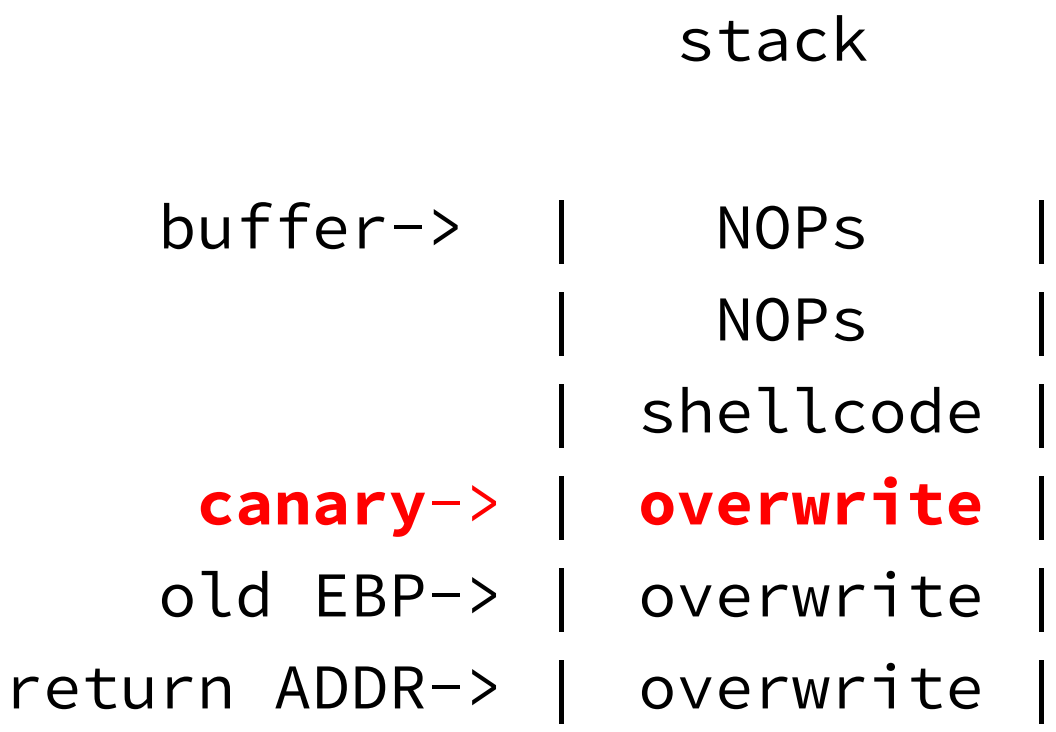
- Re-arranges variable layout on the stack to mitigate overflow (see previous class)
- Adds a random value (the **canary**) to check whether an overflow occurred

The canary mechanism requires support from the operating system that provides a **random value** when a process is started

The canary



The canary



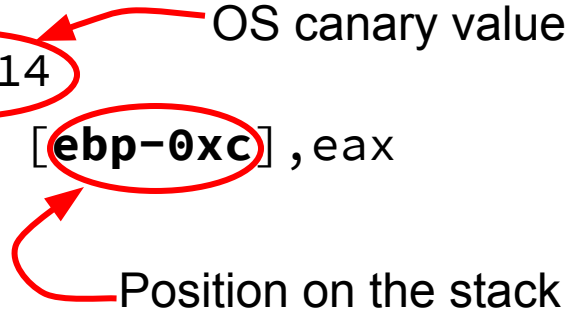
Aborts before return if canary value has changed!

Example: canary in gcc

```
0x0804850a <+26>:    mov    eax,gs:0x14
0x08048510 <+32>:    mov    DWORD PTR [ebp-0xc],eax

...

0x0804857d <+141>:   mov    edx,DWORD PTR [ebp-0xc]
0x08048580 <+144>:   xor    edx,DWORD PTR gs:0x14
0x08048587 <+151>:   je     0x804858e <main+158>
0x08048589 <+153>:   call   0x8048360 <__stack_chk_fail@plt>
...
0x08048597 <+167>:   ret
```



Limitations

Canary is a **very effective** mitigation technique

However, similarly to ASLR Canary is void **if its value is leaked**

- because of **another vulnerability**
- because the program spontaneously dumps the stack (unlikely but not impossible)

Canary is also void in case of “random” access to the stack (eg. overflowing a buffer index)

More protections

- **Fortify**: the compiler replaces unsafe functions with "fortified" ones when possible
- **PIE** (Position Independent Executable): makes it possible ASLR also for program code
- **RELRO** (RELocation Read-Only): makes it impossible to overwrite relocation address of library functions (that would cause control-flow modification)

Secure Coding

The SEI CERT C Coding Standard provides **rules** and **recommendation** from the security coding community

- **Rules** provide normative **requirements** for code
- **Recommendations** provide **guidance** to improve the **safety**, **reliability**, and **security** of software systems.

A violation of a recommendation does not necessarily indicate the presence of a defect in the code.

Risk assessment

An indication of

- potential **consequences** of not addressing a particular rule or recommendation
- the expected remediation costs

Each rule and recommendation has an assigned **priority**

Three values are assigned for each rule on a scale of 1 to 3 for **severity**, **likelihood**, and **remediation cost**

Severity

How **serious** are the **consequences** of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	Low	Denial-of-service attack, abnormal termination
2	Medium	Data integrity violation, unintentional information disclosure
3	High	Run arbitrary code

Likelihood

How likely is it that a flaw introduced by violating the rule can lead to an **exploitable vulnerability**?

Value	Meaning
1	Unlikely
2	Probable
3	Likely

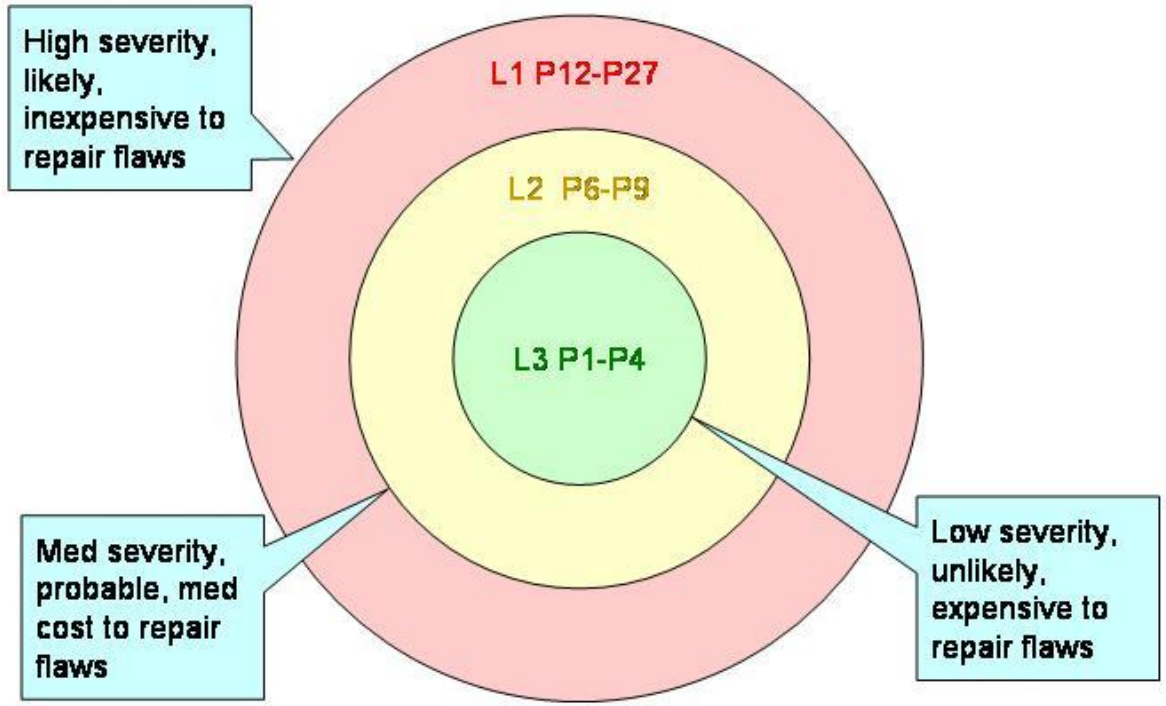
Remediation cost

How **expensive** is it to comply with the rule?

Value	Meaning	Detection	Correction
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

Priorities and levels

Severity, **likelihood**, and **remediation cost** are multiplied



Rule 06. Arrays (ARR)

Do not form or use out-of-bounds pointers or array subscripts

It is crucial that array indexes are always checked

```
1  int const TABLESIZE = 100;
2  #include <stddef.h>
3
4  static int table[TABLESIZE];
5
6  int *f(int index) {
7      if (index < TABLESIZE) {
8          return table + index;
9      }
10     return NULL;
11 }
```

Rule 06. Arrays (ARR)

Previous code is **noncompliant!**

What if `index` becomes negative?

```
1  int const TABLESIZE = 100;
2  #include <stddef.h>
3
4  static int table[TABLESIZE];
5
6  int *f(int index) {
7      if (index >=0 && index < TABLESIZE) {
8          return table + index;
9      }
10     return NULL;
11 }
```

Rule 06. Arrays (ARR)

Evaluation of this rule is:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Likely	Medium	**P18**	**L1**

Rule 07. Characters and Strings (STR)

Do not pass a non-null-terminated character sequence to a library function that expects a string

Wrong:

```c
#include <stdio.h>

void func(void) {
  char c_str[3] = "abc";
  printf("%s\n", c_str);
}
```

Correct:

```c
#include <stdio.h>

void func(void) {
  char c_str[] = "abc";
  printf("%s\n", c_str);
}
```

Rule 07. Characters and Strings (STR)

Evaluation of this rule is:

Severity	Likelihood	Remediation Cost	Priority	Level
High	Probable	Medium	**P12**	**L1**

More examples

- **Rule 07. Characters and Strings (STR)**: *Guarantee that storage for strings has sufficient space for character data and the null terminator*
- **Rec. 07. Characters and Strings (STR)**: *Use the bounds-checking interfaces for string manipulation.* For example BSD `strlcpy` and `strlcat` (`strncpy` and `strncat` might leave the string unterminated)
- **Rule 10. Environment (ENV)**: *Do not call `system()`.* Use of the `system()` function can result in exploitable vulnerabilities

Vulnerabilities due to `system()`

- When passing an **unsanitized** or improperly sanitized command string originating from a **tainted** source
- If a command is specified **without a path name** and the command processor path name resolution mechanism is accessible to an attacker
- If a **relative path** to an executable is specified and control over the current working directory is accessible to an attacker
- If the specified executable program can be **spoofed** by an attacker

Exercise

Analyse the compliance to rules and recommendations of the program at the bottom of the notes, and rewrite it to make it compliant

(NOTE: one of the vulnerabilities in the code will be presented tomorrow!)