



**CYBER
CHALLENGE**
CyberChallenge.it



SPONSOR PLATINUM



SPONSOR GOLD



SPONSOR SILVER



Software Security 3

Mitigations and Countermeasures

2

Michele LORETI

Univ. di Camerino

michele.loreti@unicam.it



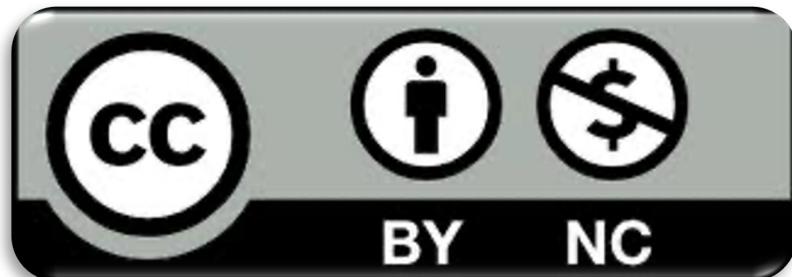
<https://cybersecnatlab.it>

License & Disclaimer

3

License Information

This presentation is licensed under the Creative Commons BY-NC License



To view a copy of the license, visit:

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Outline

4

- Non eXecutable stack (NX)
- Stack Canaries
- Attacks against canaries
- Address Space Layout Randomization (ASLR)
- ASLR bypassing techniques
- Control Flow Integrity

Non eXecutable stack (NX)

5

- *Code injection* attacks typically consist of two steps:
 - Add malicious code in the stack;
 - Jump to it.
- To contrast this kind of attacks, memory areas can be marked as *non executable*.
- The command *readelf* can be used to check if a given section is executable or not.

Non eXecutable stack (NX)

6

Below the output of *readelf -e proname*

```
Program Headers:
Type          Offset      VirtAddr     PhysAddr     FileSiz  MemSiz Flg Align
PHDR          0x000034  0x08048034  0x08048034  0x00120  0x00120 R E 0x4
INTERP        0x000154  0x08048154  0x08048154  0x00013  0x00013 R   0x1
                  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD          0x000000  0x08048000  0x08048000  0x0064c  0x0064c R E 0x1000
LOAD          0x000f08  0x08049f08  0x08049f08  0x00118  0x0011c RW  0x1000
DYNAMIC       0x000f14  0x08049f14  0x08049f14  0x000e8  0x000e8 RW  0x4
NOTE          0x000168  0x08048168  0x08048168  0x00044  0x00044 R   0x4
GNU_EH_FRAME  0x00052c  0x0804852c  0x0804852c  0x00034  0x00034 R   0x4
GNU_STACK     0x000000  0x00000000  0x00000000  0x00000  0x00000 RWE 0x10
GNU_RELRO    0x000f08  0x08049f08  0x08049f08  0x000f8  0x000f8 R   0x1
```

Non eXecutable stack (NX)

7

```
Program Headers:
Type          Offset    VirtAddr   PhysAddr   FileSiz MemSiz Flg Align
PHDR          0x000034  0x08048034  0x08048034  0x00120  0x00120 R E 0x4
INTERP        0x000154  0x08048154  0x08048154  0x00013  0x00013 R  0x1
                  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD          0x000000  0x08048000  0x08048000  0x0064c  0x0064c R E 0x1000
LOAD          0x000f08  0x08049f08  0x08049f08  0x00118  0x0011c RW 0x1000
DYNAMIC       0x000f14  0x08049f14  0x08049f14  0x000e8  0x000e8 RW 0x4
NOTE          0x000168  0x08048168  0x08048168  0x00044  0x00044 R  0x4
GNU_EH_FRAME  0x000520  0x08048520  0x08048520  0x00034  0x00034 R  0x4
GNU_STACK    0x000000  0x00000000  0x00000000  0x00000  0x00000 RWE 0x10
GNU_RELRO    0x00000T08 0x08049T08 0x08049T08 0x0000T8 0x0000T8 R  0x1
```

Above we see that the stack is enabled for execution.

Non eXecutable stack (NX)

8

- *gcc* allows us to *enable* or *disable* *NX flag* on building:

NX enabled

```
gcc -z noexecstack -o nxprogram myprogram.c -m32
```

NX disabled

```
gcc -z execstack -o nonxprogram myprogram.c -m32
```

Non eXecutable stack (NX)

9

- Pwntools can be also used to read ELF binary file and check the status of NX:

```
>>> elf = ELF('nxprogram')
[*] '/home/loreti/CC/SS3/nxprogram'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX enabled
    PIE:       No PIE (0x8048000)
>>>
```

```
>>> elf = ELF('nonxprogram')
[*] '/home/loreti/CC/SS3/nonxprogram'
    Arch:      i386-32-little
    RELRO:     Partial RELRO
    Stack:     Canary found
    NX:        NX disabled
    PIE:       No PIE (0x8048000)
    RWX:       Has RWX segments
>>>
```

Non eXecutable stack (NX)

10

- The use of NX allow us to mitigate *code injection*.
- However other form of ACE, like for instance *return-to-libc*, are not affected.
- This because the malicious code (a function in the *Standard C Library*) is not stored in the stack.

Stack Canaries

11

- Stack Canaries are secret values placed on the stack which change every time the program is executed.
- Prior to a function return, the stack canary is checked and if it appears to be modified, the program exits immediately.
- The name is related to the birds used in the coal mines to check the presence of dangerous gases.



Stack Canaries

12

- The *pseudocode* of a function that uses *canaries* is:

```
void afunction(...) {
    long canary = CANARY_VALUE; //Canary initialization
    ...
    ...
    if (canary != CANARY_VALUE) {
        exit(CANARY_DEAD);
    }
}
```

- Gcc compiler can generate a more efficient code for us.
 - Option *-fstack-protector* must be used.

Stack Canaries

13

- The basic methodology is to place a filler word, the canary, between local variables, or buffer contents in general, and the return address.
- The canary may consist of a mix of *random* and *terminator* values.
- At the end of the function, just before the return instruction, the integrity of the canary is checked:
 - If no alteration is found, execution resumes normally.
 - If an alteration is detected, program execution is terminated immediately.

Stack Canaries

14

Without Canaries

```
0804843b <welcome>:  
0804843b:    55          push  %ebp  
0804843c:  89 e5        mov   %esp,%ebp  
0804843e:  83 ec 18      sub   $0x18,%esp  
08048441:  83 ec 08      sub   $0x8,%esp  
08048444: ff 75 08      pushl 0x8(%ebp)  
08048447: 8d 45 ee      lea   -0x12(%ebp),%eax  
0804844a: 50            push  %eax  
0804844b: e8 c0 fe ff ff call  8048310 <strcpy@plt>  
08048450: 83 c4 10      add   $0x10,%esp  
08048453: 83 ec 08      sub   $0x8,%esp  
08048456: 8d 45 ee      lea   -0x12(%ebp),%eax  
08048459: 50            push  %eax  
0804845a: 68 20 85 04 08 push  $0x8048520  
0804845f: e8 9c fe ff ff call  8048300 <printf@plt>  
08048464: 83 c4 10      add   $0x10,%esp  
08048467: 90            nop  
08048468: c9            leave  
08048469: c3            ret
```

With Canaries

```
0804849b <welcome>:  
0804849b:    55          push  %ebp  
0804849c:  89 e5        mov   %esp,%ebp  
0804849e:  83 ec 28      sub   $0x28,%esp  
080484a1: 8b 45 08      mov   0x8(%ebp),%eax  
080484a4: 89 45 e4      mov   %eax,-0x1c(%ebp)  
080484a7: 65 a1 14 00 00 00 mov   %gs:0x14,%eax  
080484ad: 89 45 f4      mov   %eax,-0xc(%ebp)  
080484b0: 31 c0          xor   %eax,%eax  
080484b2: 83 ec 08      sub   $0x8,%esp  
080484b5: ff 75 e4      pushl -0x1c(%ebp)  
080484b8: 8d 45 ea      lea   -0x16(%ebp),%eax  
080484bb: 50            push  %eax  
080484bc: e8 af fe ff ff call  8048370 <strcpy@plt>  
080484c1: 83 c4 10      add   $0x10,%esp  
080484c4: 83 ec 08      sub   $0x8,%esp  
080484c7: 8d 45 ea      lea   -0x16(%ebp),%eax  
080484ca: 50            push  %eax  
080484cb: 68 a0 85 04 08 push  $0x80485a0  
080484d0: e8 7b fe ff ff call  8048350 <printf@plt>  
080484d5: 83 c4 10      add   $0x10,%esp  
080484d8: 90            nop  
080484d9: 8b 45 f4      mov   -0xc(%ebp),%eax  
080484dc: 65 33 05 14 00 00 00 xor   %gs:0x14,%eax  
080484e3: 74 05          je    80484ea <welcome+0x4f>  
080484e5: e8 76 fe ff ff call  8048360 <__stack_chk_fail@plt>  
080484ea: c9            leave  
080484eb: c3            ret
```

Stack Canaries

15

Without Canaries

```
0804843b <welcome>:  
0804843b:    55          push  %ebp  
0804843c:    89 e5       mov   %esp,%ebp  
0804843e:    83 ec 18    sub   $0x18,%esp  
08048441:    83 ec 08    sub   $0x8,%esp  
08048444:    ff 75 08    pushl 0x8(%ebp)  
08048447:    8d 45 ee    lea   -0x12(%ebp),%eax  
0804844a:    50          push  %eax  
0804844b:    e8 c0 fe ff ff  call  8048310 <strcpy@plt>  
08048450:    83 c4 10    add   $0x10,%esp  
08048453:    83 ec 08    sub   $0x8,%esp  
08048456:    8d 45 ee    lea   -0x12(%ebp),%eax  
08048459:    50          push  %eax  
0804845a:    68 20 85 04 08  push  $0x8048520  
0804845f:    e8 9c fe ff ff  call  8048300 <printf@plt>  
08048464:    83 c4 10    add   $0x10,%esp  
08048467:    90          nop  
08048468:    c9          leave  
08048469:    c3          ret
```

With Canaries

```
0804849b <welcome>:  
0804849b:    55          push  %ebp  
0804849c:    89 e5       mov   %esp,%ebp  
0804849d:    83 ec 0c    sub   $0x1c,%esp  
080484a1:    8b 45 08    mov   0x8(%ebp),%eax  
080484a4:    89 45 e4    mov   %eax,-0x1c(%ebp)  
080484a7:    65 a1 14 00 00 00  mov   %gs:0x14,%eax  
080484ad:    89 45 f4    mov   %eax,-0xc(%ebp)  
080484b0:    31 c0       xor   %eax,%eax  
080484b2:    83 ec 08    sub   $0x8,%esp  
080484b5:    ff 75 e4    pushl -0x1c(%ebp)  
080484b8:    8d 45 ea    lea   -0x16(%ebp),%eax  
080484bb:    50          push  %eax  
080484bc:    e8 af fe ff ff  call  8048370 <strcpy@plt>  
080484c1:    83 c4 10    add   $0x10,%esp  
080484c4:    83 ec 08    sub   $0x8,%esp  
080484c7:    8d 45 ea    lea   -0x16(%ebp),%eax  
080484ca:    50          push  %eax  
080484cb:    68 a0 85 04 08  push  $0x80485a0  
080484d0:    e8 7b fe ff ff  call  8048350 <printf@plt>  
080484d5:    83 c4 10    add   $0x10,%esp  
080484d8:    90          nop  
080484d9:    8b 45 f4    mov   -0xc(%ebp),%eax  
080484dc:    65 33 05 14 00 00 00  xor   %gs:0x14,%eax  
080484e3:    74 05       je    80484ea <welcome+0x4f>  
080484e5:    e8 76 fe ff ff  call  8048360 <__stack_chk_fail@plt>  
080484ea:    c9          leave  
080484eb:    c3          ret
```

Initialization

Stack Canaries

16

Without Canaries

```
0804843b <welcome>:  
0804843b: 55  
0804843c: 89 e5  
0804843e: 83 ec 18  
08048441: 83 ec 08  
08048444: ff 75 08  
08048447: 8d 45 ee  
0804844a: 50  
0804844b: e8 c0 fe ff ff  
08048450: 83 c4 10  
08048453: 83 ec 08  
08048456: 8d 45 ee  
08048459: 50  
0804845a: 68 20 85 04 08  
0804845f: e8 9c fe ff ff  
08048464: 83 c4 10  
08048467: 90  
08048468: c9  
08048469: c3  
  
push %ebp  
mov %esp,%ebp  
sub $0x18,%esp  
sub $0x8,%esp  
pushl 0x8(%ebp)  
lea -0x12(%ebp),%eax  
push %eax  
call 8048310 <strcpy@plt>  
add $0x10,%esp  
sub $0x8,%esp  
lea -0x12(%ebp),%eax  
push %eax  
push $0x8048520  
call 8048300 <printf@plt>  
add $0x10,%esp  
nop  
leave  
ret
```

With Canaries

```
0804849b <welcome>:  
0804849b: 55  
0804849c: 89 e5  
0804849e: 83 ec 28  
080484a1: 8b 45 08  
080484a4: 89 45 e4  
080484a7: 65 a1 14 00 00 00  
080484ad: 89 45 f4  
080484b0: 31 c0  
080484b2: 83 ec 08  
080484b5: ff 75 e4  
080484b8: 8d 45 ea  
080484bb: 50  
080484bc: e8 af fe ff ff  
080484c1: 83 c4 10  
080484c4: 83 ec 08  
080484c7: 8d 45 ea  
080484ca: 50  
080484cb: 68 a0 85 04 08  
080484d0: e8 7b fe ff ff  
080484d5: 83 c4 10  
080484d8: .  
080484d9: 8b 45 f4  
080484dc: 65 33 05 14 00 00 00  
080484e3: 74 05  
080484e5: e8 76 fe ff ff  
080484e8: c9  
080484eb: c3  
  
push %ebp  
mov %esp,%ebp  
sub $0x28,%esp  
mov 0x8(%ebp),%eax  
mov %eax,-0x1c(%ebp)  
mov %gs:0x14,%eax  
mov %eax,-0xc(%ebp)  
xor %eax,%eax  
sub $0x8,%esp  
pushl -0x1c(%ebp)  
lea -0x16(%ebp),%eax  
push %eax  
call 8048370 <strcpy@plt>  
add $0x10,%esp  
sub $0x8,%esp  
lea -0x16(%ebp),%eax  
push %eax  
call 8048350 <printf@plt>  
add $0x10,%esp  
nop  
mov -0xc(%ebp),%eax  
xor %gs:0x14,%eax  
je 80484ea <welcome+0x4f>  
call 8048360 <__stack_chk_fail@plt>  
leave  
ret
```

Test

Terminator Canaries

17

- Terminator Canaries are special sequence of bytes that *blocks* execution of functions such as *strcpy* or *gets*:
 - *NULL*, 0x00
 - *CR*, 0x0d
 - *LR*, 0x0a
 - *EOF*, 0xff
- The use of these 2-byte characters do terminate most string operations and makes the overflow attempt harmless.

Random Canaries

18

- Values are randomly selected with the goal of making it *exceedingly difficult* for attackers to find the right value.
- The random value is taken from `/dev/urandom` if available, otherwise by hashing the time of day.
- The randomness is sufficient to prevent most prediction attempts.

Bypass Canaries

19

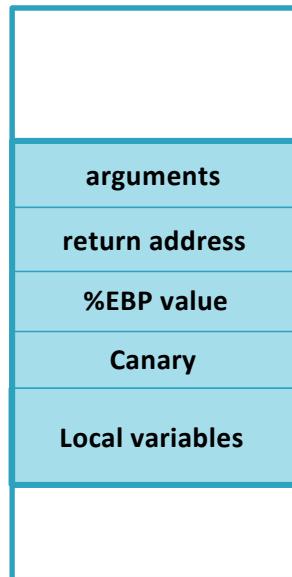
- Stack Canaries seem an efficient mechanism to mitigate any stack smashing.
- However, there are two techniques that can be used to bypass them:
 - Stack Canary Leaking
 - Bruteforcing attack

Stack Canary Leaking

20

- When stack canaries are used, we can check if data in the stack have been corrupted:

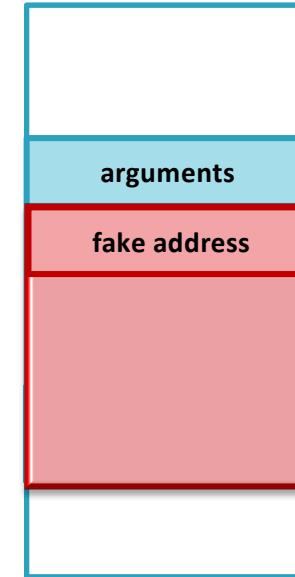
Original Stack



Corrupting Data



Corrupted Stack



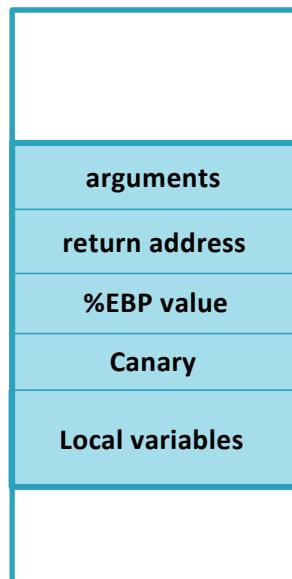
Curroption
Detected!

Stack Canary Leaking

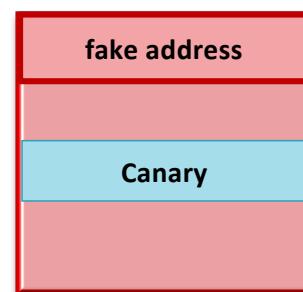
21

- If an attacker knows or is able to read the data in the stack canary, that value can be used in the *corrupting* data:

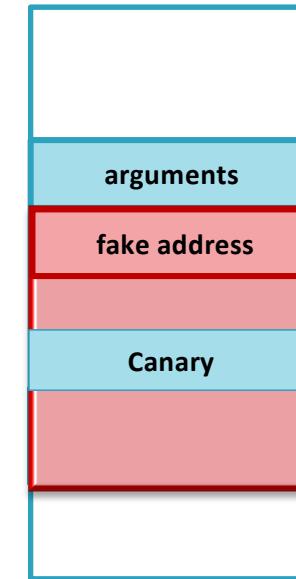
Original Stack



Corrupting Data



Corrupted Stack



Curroption
Is not detected!

Stack Canary Leaking

22

- When *terminator canaries* are used, their use in a corrupting data is not easy
 - For instance, the first byte of the stack canary could be NULL, meaning that string functions will stop when they hit it.
- In this case an attacker would perform two actions:
 - A first action to overwrite stack content;
 - A second one to put the *termination canary* back.

Bruteforcing a Stack Canary

23

- The canary is determined when the program starts up for the first time...
 - If the program forks, it keeps the same canaries in the child process.
 - Each child is used as an *oracle* to test *one canary value*.
 - The attack is replicated (with different values) on multiple children.
 - This method can be used on fork-and-accept servers where connections are spun off to child processes.

Address Space Layout Randomization

24

- The *Address Space Layout Randomization* (ASLR) is a computer security technique which involves randomly positioning:
 - the base address of an executable;
 - the position of libraries, heap, and stack, in a process's address space.

Address Space Layout Randomization

25

- Operating Systems may support different kinds of ASLRs
- Linux OS offers three ASLR modes:
 - No randomization, everything is static;
 - Conservative randomization, main process components (shared libraries, stack, mmap,...) are randomized;
 - Full randomization.

Address Space Layout Randomization

26

- Let us consider the following simple C program that just prints the address of a local variable:

```
#include <stdio.h>

void main(int argc, char **argv) {
    int aVariable = 0;
    printf("Global is stored ad %p\n", &aVariable);
}
```

Address Space Layout Randomization

27

- This is the result of some executions with ASLR disabled:

```
[CC >echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
[CC >./testaslr
Global is stored ad 0xfffffffffe4a4
CC >]
```

Address Space Layout Randomization

28

- The same executions with ASLR enabled:

```
[CC >echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
[CC >./testaslr
Global is stored ad 0x7ffc1d5666f4
[CC >./testaslr
Global is stored ad 0x7ffd36e4bc4
[CC >./testaslr
Global is stored ad 0x7fffa73e9a34
[CC >./testaslr
Global is stored ad 0x7fffa8aa7e14
CC >]
```

Bypassing ASLR

29

- **Brute Force:** the exploitation is repeated with different addresses until success.
 - In some cases, like for *code injection*, extra *dummy instructions (NOP)* are used to increase the probability that the used address hits the injected code.
- **Information leakage:** collected info can be used to instrument an attack based on ROP or JOP.
- **OS specific attack:** since ASLR is specific for each OS, specific attacks and vulnerabilities can be used.

Control-Flow Integrity (CFI)

30

- Control-Flow Integrity (CFI) is a technic that aims to prevent ACE attacks.
- To control the *correct execution*, CFI uses a Control-Flow Graph (CFG) determined ahead of time (via static analysis).
- At runtime, it is checked that control flow remains within a given CFG.

Control-Flow Integrity (CFI)

31

- Control-Flow Integrity (CFI) is an efficient defense against arbitrary code execution in general.
- Unfortunately, it could be very costly or hard to implement.
- Details about CFI can be found in:

Control-flow integrity principles, implementations, and applications

Abadi, Budiu, Erligsson, Ligatti

ACM Transactions on Information and System Security, November 2009

Software Security 3

Mitigations and Countermeasures

32

Michele LORETI

Univ. di Camerino

michele.loreti@unicam.it



<https://cybersecnatlab.it>



**CYBER
CHALLENGE**
CyberChallenge.IT



SPONSOR PLATINUM



SPONSOR GOLD



SPONSOR SILVER

