Countermeasures

Michele LORETI
Università di Camerino

https://cybersecnatlab.it

License & Disclaimer

License Information

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.

Goal

- In this lecture some countermeasures will be presented that permits mitigating the impact of memory corruption attacks.

Prerequisites

- Lecture:
  - Basic knowledge of C

Outline

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

Non eXecutable stack (NX)

- C*ode 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)

- Below the output of *readelf –e progname*

```
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)


```
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
```

- Above we see that the stack is enabled for execution

Non eXecutable stack (NX)

- *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)

- 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)

- 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

- 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

- 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

- 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

Without Canaries

```
0804843b <welcome>:
 804843b:       55                      push   %ebp
 804843c:       89 e5                   mov    %esp,%ebp
 804843e:       83 ec 18                sub    $0x18,%esp
 8048441:       83 ec 08                sub    $0x8,%esp
 8048444:       ff 75 08                pushl  0x8(%ebp)
 8048447:       8d 45 ee                lea    -0x12(%ebp),%eax
 804844a:       50                      push   %eax
 804844b:       e8 c0 fe ff ff          call   8048310 <strcpy@plt>
 8048450:       83 c4 10                add    $0x10,%esp
 8048453:       83 ec 08                sub    $0x8,%esp
 8048456:       8d 45 ee                lea    -0x12(%ebp),%eax
 8048459:       50                      push   %eax
 804845a:       68 20 85 04 08          push   $0x8048520
 804845f:       e8 9c fe ff ff          call   8048300 <printf@plt>
 8048464:       83 c4 10                add    $0x10,%esp
 8048467:       90                      nop
 8048468:       c9                      leave
 8048469:       c3                      ret
```

With Canaries

```
0804849b <welcome>:
 804849b:       55                      push   %ebp
 804849c:       89 e5                   mov    %esp,%ebp
 804849e:       83 ec 28                sub    $0x28,%esp
 80484a1:       8b 45 08                mov    0x8(%ebp),%eax
 80484a4:       89 45 e4                mov    %eax,-0x1c(%ebp)
 80484a7:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80484ad:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b0:       31 c0                   xor    %eax,%eax
 80484b2:       83 ec 08                sub    $0x8,%esp
 80484b5:       ff 75 e4                pushl  -0x1c(%ebp)
 80484b8:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484bb:       50                      push   %eax
 80484bc:       e8 af fe ff ff          call   8048370 <strcpy@plt>
 80484c1:       83 c4 10                add    $0x10,%esp
 80484c4:       83 ec 08                sub    $0x8,%esp
 80484c7:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484ca:       50                      push   %eax
 80484cb:       68 a0 85 04 08          push   $0x80485a0
 80484d0:       e8 7b fe ff ff          call   8048350 <printf@plt>
 80484d5:       83 c4 10                add    $0x10,%esp
 80484d8:       90                      nop
 80484d9:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484dc:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 80484e3:       74 05                   je     80484ea <welcome+0x4f>
 80484e5:       e8 76 fe ff ff          call   8048360 <__stack_chk_fail@plt>
 80484ea:       c9                      leave
 80484eb:       c3                      ret
```

Stack Canaries

Without Canaries

```
0804843b <welcome>:
 804843b:       55                      push   %ebp
 804843c:       89 e5                   mov    %esp,%ebp
 804843e:       83 ec 18                sub    $0x18,%esp
 8048441:       83 ec 08                sub    $0x8,%esp
 8048444:       ff 75 08                pushl  0x8(%ebp)
 8048447:       8d 45 ee                lea    -0x12(%ebp),%eax
 804844a:       50                      push   %eax
 804844b:       e8 c0 fe ff ff          call   8048310 <strcpy@plt>
 8048450:       83 c4 10                add    $0x10,%esp
 8048453:       83 ec 08                sub    $0x8,%esp
 8048456:       8d 45 ee                lea    -0x12(%ebp),%eax
 8048459:       50                      push   %eax
 804845a:       68 20 85 04 08          push   $0x8048520
 804845f:       e8 9c fe ff ff          call   8048300 <printf@plt>
 8048464:       83 c4 10                add    $0x10,%esp
 8048467:       90                      nop
 8048468:       c9                      leave
 8048469:       c3                      ret
```

With Canaries

Initialization

```
0804849b <welcome>:
 804849b:       55                      push   %ebp
 804849c:       89 e5                   mov    %esp,%ebp
 80484a1:       8b 45 08                mov    0x8(%ebp),%eax
 80484a4:       89 45 e4                mov    %eax,-0x1c(%ebp)
 80484a7:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80484ad:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b0:       31 c0                   xor    %eax,%eax
 80484b2:       83 ec 08                sub    $0x8,%esp
 80484b5:       ff 75 e4                pushl  -0x1c(%ebp)
 80484b8:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484bb:       50                      push   %eax
 80484bc:       e8 af fe ff ff          call   8048370 <strcpy@plt>
 80484c1:       83 c4 10                add    $0x10,%esp
 80484c4:       83 ec 08                sub    $0x8,%esp
 80484c7:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484ca:       50                      push   %eax
 80484cb:       68 a0 85 04 08          push   $0x80485a0
 80484d0:       e8 7b fe ff ff          call   8048350 <printf@plt>
 80484d5:       83 c4 10                add    $0x10,%esp
 80484d8:       90                      nop
 80484d9:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484dc:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 80484e3:       74 05                   je     80484ea <welcome+0x4f>
 80484e5:       e8 76 fe ff ff          call   8048360 <__stack_chk_fail@plt>
 80484ea:       c9                      leave
 80484eb:       c3                      ret
```

Stack Canaries

Without Canaries

```
0804843b <welcome>:
 804843b:       55                      push   %ebp
 804843c:       89 e5                   mov    %esp,%ebp
 804843e:       83 ec 18                sub    $0x18,%esp
 8048441:       83 ec 08                sub    $0x8,%esp
 8048444:       ff 75 08                pushl  0x8(%ebp)
 8048447:       8d 45 ee                lea    -0x12(%ebp),%eax
 804844a:       50                      push   %eax
 804844b:       e8 c0 fe ff ff          call   8048310 <strcpy@plt>
 8048450:       83 c4 10                add    $0x10,%esp
 8048453:       83 ec 08                sub    $0x8,%esp
 8048456:       8d 45 ee                lea    -0x12(%ebp),%eax
 8048459:       50                      push   %eax
 804845a:       68 20 85 04 08          push   $0x8048520
 804845f:       e8 9c fe ff ff          call   8048300 <printf@plt>
 8048464:       83 c4 10                add    $0x10,%esp
 8048467:       90                      nop
 8048468:       c9                      leave
 8048469:       c3                      ret
```

With Canaries

```
0804849b <welcome>:
 804849b:       55                      push   %ebp
 804849c:       89 e5                   mov    %esp,%ebp
 804849e:       83 ec 28                sub    $0x28,%esp
 80484a1:       8b 45 08                mov    0x8(%ebp),%eax
 80484a4:       89 45 e4                mov    %eax,-0x1c(%ebp)
 80484a7:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80484ad:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b0:       31 c0                   xor    %eax,%eax
 80484b2:       83 ec 08                sub    $0x8,%esp
 80484b5:       ff 75 e4                pushl  -0x1c(%ebp)
 80484b8:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484bb:       50                      push   %eax
 80484bc:       e8 af fe ff ff          call   8048370 <strcpy@plt>
 80484c1:       83 c4 10                add    $0x10,%esp
 80484c4:       83 ec 08                sub    $0x8,%esp
 80484c7:       8d 45 ea                lea    -0x16(%ebp),%eax
 80484ca:       50                      push   %eax
 80484cb:       68 a0 85 04 08          push   $0x80485a0
 80484d0:       e8 7b fe ff ff          call   8048350 <printf@plt>
 80484d5:       83 c4 10                add    $0x10,%esp
 80484d8:       90                      nop
 80484d9:       8b 45 f4                mov    -0xc(%ebp),%eax
 80484dc:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 80484e3:       74 05                   je     80484ea <welcome+0x4f>
 80484e5:       e8 76 fe ff ff          call   8048360 <__stack_chk_fail@plt>
 80484ea:       c9                      leave
 80484eb:       c3                      ret
```

Test

Terminator Canaries

- 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

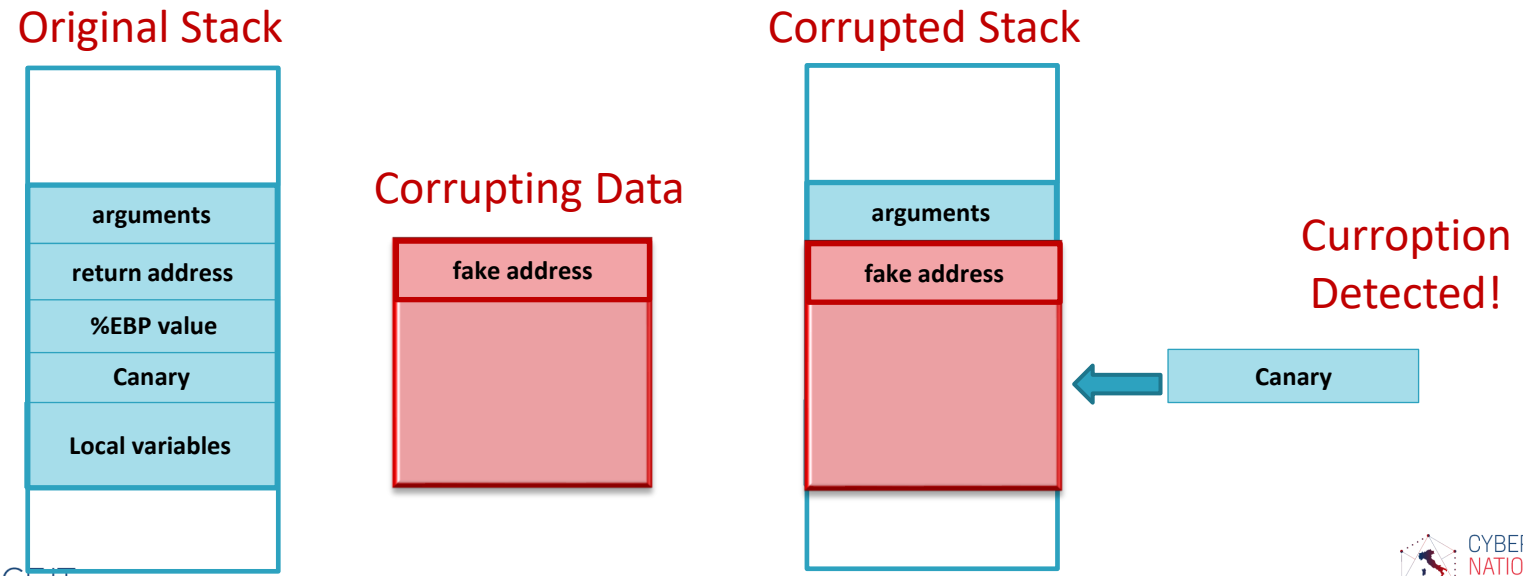
- Values are randomly selected with the goal of making it *exceedingly difficult* for attackers to find the right value
- The random value is taken from */dev/urandom* if available, otherwise by hashing the time of day
- The randomness is sufficient to prevent most prediction attempts

Bypass Canaries

- 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

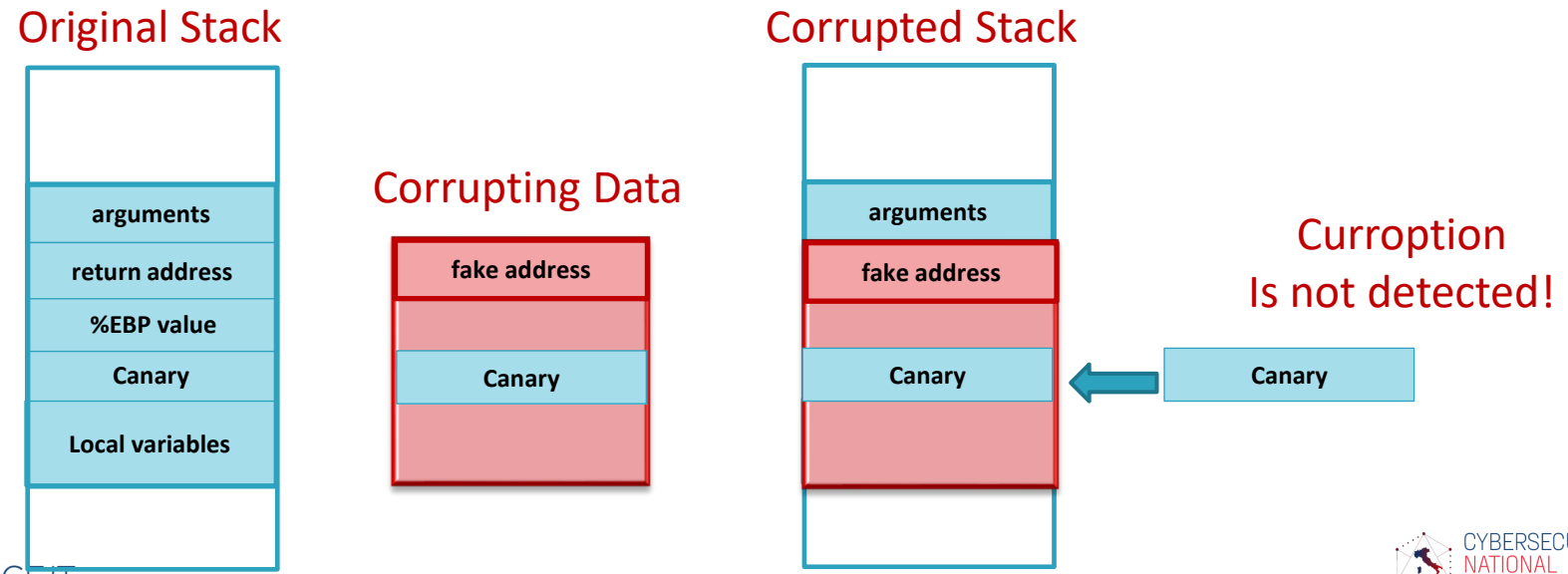
Stack Canary Leaking

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



Stack Canary Leaking

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



Stack Canary Leaking

- 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

- 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

- 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

- 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

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

```c
#include <stdio.h>

void main(int argc, char **argv) {

   int aVariable = 0;

   printf("Global is stored ad %p\n",&aVariable);

}
```

Address Space Layout Randomization

- 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 0x7fffffffe4a4
[CC >./testaslr
 Global is stored ad 0x7fffffffe4a4
[CC >./testaslr
 Global is stored ad 0x7fffffffe4a4
[CC >./testaslr
 Global is stored ad 0x7fffffffe4a4
 CC >
```

Address Space Layout Randomization

- 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 0x7ffdd36e4bc4
[CC >./testaslr
Global is stored ad 0x7fffa73e9a34
[CC >./testaslr
Global is stored ad 0x7fffa8aa7e14
CC >
```

Bypassing ASLR

- 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)

- 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)

- 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

Countermeasures

Michele LORETI
Università di Camerino

CYBER CHALLENGE.IT

CYBERSECURITY NATIONAL LABORATORY

https://cybersecnatlab.it