Software Vulnerabilities: Exploitation and Mitigation

—

Lab 2

Alexandre Bartel

The report for the lab should consist of a single pdf file. Please use the following filename:

`lab2_${FIRSTNAME}_${LASTNAME}.pdf`

Send the report to alexandre.bartel@uni.lu with the following subject:

`MICS-SOFTVULN2019 Lab2 ${FIRSTNAME}_${LASTNAME}`

The deadline is the day before the next lecture at 23:59.

1 Lab2 (46 P.)

In this lab you will exploit a buffer overflow on the stack to execute arbitrary code. Download the following qcow2 Debian image (user:user, root:root) here. The image does only has terminal access. If you prefer to have a graphical environment you can install the following package as root:

```
# apt-get install xfce4
```

Install qemu. To launch the image with qemu use the following command:

```
$ qemu-system-x86_64 -hda debian-svem.qcow -m 512
```

1.1 Stack Overflow

Write the following C program in file `test.c`.

```
#include <stdio.h>
#include <string.h>

void function1(char * arg) {
  char buffer[100];
  strcpy(buffer, arg);
```

```
  printf("buffer is: '%s' \n");
}

int main(int argc, char** argv) {
  printf("Welcome to this vulnerable program!\n");
  printf("argv[0]: '%s' argv[1]: '%s'\n", argv[0], argv[1]);
  function1(argv[1]);
  return 0;
}
```

Compile `test.c` using the following command:

```
$ gcc -o test -fno-stack-protector -z execstack -no-pie test.c
```

Question 1.1 Explain options `-fno-stack-protector`, `-z execstack` and `-no-pie`. 3 P.

Run the program with the following command line:

```
$ ./test "hi there!"
```

Question 1.2 What is the output? 1 P.

Question 1.3 What input should you give to the program to generate a *Segmentation Fault*? 2 P.

You can look at the where the different libraries (code) and the code of the main program is loaded in memory (virtual memory) by looking at the content of the `maps` file in the `proc` filesystem:

```
$ cat /proc/PID/maps
```

Launch the program with `gdb`:

```
$ gdb ./test
```

Run the program with the input causing a segmentation fault:

```
(gdb) run YOUR_INPUT_HERE
```

Gdb stops when a segmentation fault is generated.

Question 1.4 Explain what a segmentation fault is. 1 P.

Question 1.5 Find the `test` program PID (process ID). List the addresses at which code or data is placed in the virtual memory. Is the stack executable? Why is this important information to the attacker? 4 P.

In `gdb` you can look at the current back trace with the following command:

```
(gdb) bt
```

Question 1.6 Explain the current back trace at the program crash. 2 P.

In `gdb` you can look at conent of memory at a specific address with the following command:

```
(gdb) x/30gx ADDRESS
(gdb) x/30gx $rsp
```

Question 1.7 At what address start the local variable `buffer` on the stack? At what address ends the local variable `buffer` on the stack? 2 P.

Question 1.8 At what address is located the return address of function `function1`? 2 P.

In `gdb` you can set a breakpoint (at the start of a function or a specific memory location) with the following command:

```
(gdb) break function1
(gdb) break *0x7ffff7f708c0
(gdb) break *$rsp
```

Question 1.9 Are return addresses encoded in memory in little-endian of big-endian? Why is that important (think how the overflow overwrites the stack and the constraints imposed by strcpy)? 4 P.

Question 1.10 At what position(s) in your input to program test should you put the return address you control? 2 P.

You now know how to change the return address! In the following section, you will generate the assembly code to execute.

1.2 Shellcode

Question 1.11 Why is the code an attacker wants to execute called a "shellcode"? 2 P.

Suppose the attacker wants to execute a bash script. Write the following code to `system.c`.

```
#include <stdlib.h>

int main(int argc, char** argv){
  system("/bin/sh");
}
```

Compile `system.c` using the following command:

```
$ gcc -o system -fno-stack-protector -z execstack -no-pie system.c
```

Disassemble the `test` binary using the following command:

```
$ objdump -d test
```

Question 1.12 Explain the assembly code of the `main` function. Explain in details how parameters are passed to the `system` function. 4 P.

In theory, the attacker should generate an assembly script based on the disassembly code you have just analyzed. In practice, one can already find optimized shellcodes on the Internet. Write the following assembly code in `command.nasm`.

```
; 22 byte execve("/bin//sh", 0, 0) for linux/x86-64
; source: https://modexp.wordpress.com/2016/03/31/x64-shellcodes-linux/

    bits 64

    push    59
    pop     rax     ; eax = 59
    cdq             ; edx = 0
    push    rdx     ; NULL
    pop     rsi     ; esi = NULL
    mov     rcx, '/bin//sh'
    push    rdx     ; 0
    push    rcx     ; "/bin//sh"
    push    rsp
    pop     rdi     ; rdi="/bin//sh",0
    syscall
```

Question 1.13 Explain the assembly code of the `main` function. Draw a picture of the stack just before instruction `syscall` is executed. What are the values of the different registers just before instruction `syscall` is executed? 6 P.

Assemble and link the program with the following commands:

```
$ nasm -f elf64 command.nasm
$ ld -m elf_x86_64 -s -o command command.o
```

Execute the resulting binary to see that it works (it pops a shell).
Look at the assembly code bytes:

```
$ objdump -d command
```

Question 1.14 Is there any byte with the value zero? Why is that important in our case with `strcpy`? 3 P.

1.3 Vulnerability Exploitation

You now have:

- the address of the local variable buffer,
- the address of the return address of function1 on the stack and
- a working shell code.

Question 1.15 Generate an input to the `test` program containing the shellcode. Make sure that the return address you overwrite, jumps to your shellcode. Execute the `test` program with your input to see if it pops a shell as expected. Explain your process and the difficulties you have encountered. 8 P.

Note that to print special characters you can use the `printf` command in bash:

```
$ echo "`printf '0x41'`"
A
```

Note on plagiarism

Plagiarism is the misrepresentation of the work of another as your own. It is a serious infraction. Instances of plagiarism or any other cheating will at the very least result in failure of this course. To avoid plagiarism, always properly cite your sources.