INFO8013: Advanced Computer Security Exploits – Format String Vulnerability

B. Donnet, L. Mathy, S. Ben Mariem, V. Jacquot

1 Context

The objective of this lab is for students to gain the first-hand experience on format-string vulnerability by putting what they have learned about the vulnerability from class into actions. Students will be given a program with a format-string vulnerability ; their task is to exploit the vulnerability to achieve the following damage : (1) crash the program, (2) read the internal memory of the program, (3) modify the internal memory of the program, and most severely, (4) inject and execute malicious code using the victim program's privilege.

In this lab, you will be given a program with a buffer-overflow vulnerability. You will be guided to develop a scheme in order to exploit this vulnerability. Your objective will be to modify the behavior of the program, and execute some malicious code.

Important : The theory on this subject may be quite complex to understand at first. However, the exercises have been incrementally defined so that you can put the theory into practice. So don't worry, if this is a bit abstract at first.

This lab must be done **individually**.

2 Background

A format string vulnerability is a bug where user input is passed as the format argument to `printf`, `scanf`, or another function in that family.

Here we will learn how to use `printf` not only to inspect the stack but also, remarkably, to change it as well.

2.1 Format String vulnerability

Recall that `printf` is a function that takes a variable number of arguments. The first should be a format string, which, in addition to plain text, may contain any number n of format specifiers, which are treated as holes in the plain text. The remaining arguments are expected to be n values whose printed representations, as determined by the corresponding specifiers, will fill corresponding holes. Here are some of the format specifiers :

Parameter	Meaning	Passed as
`%d`	decimal	value
`%u`	unsigned decimal (unsigned int)	value
`%x`	hexadecimal (unsigned int)	value
`%s`	string ((const) (unsigned) char *)	reference
`%n`	number of bytes written so far, (* int)	reference

TABLE 1 – format parameters

2.2 What if there is a miss-match ?

The behavior of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack. But what happens if there is a miss-match between the format string and the actual arguments ?

```
printf ("a␣=␣%d,␣b␣=␣%d,␣c␣is␣at␣address:␣%08x\n", a, b);
```

In this example, the format string requires 3 arguments, but the program actually provides only two (i.e. `a` and `b`). Can `printf` detect the miss-match ? In theory, the function `printf` fetches the arguments from the stack. If the format string requires 3 arguments, it will fetch 3 elements from the stack. Unless the stack is marked with a boundary, `printf` does not know that it runs out of the arguments that are provided to it. Since there is no such a marking `printf` will continue fetching data from the stack. In a miss-match case, it will fetch some data that do not belong to this function call.

2.3 Viewing the stack

Although `printf` does not "know" how many arguments it takes, it can rely on the same aspects of the procedure calling convention used by all C functions to find their arguments : the ith argument (1-indexed) is at an offset $4(i+1)$ bytes from the base of the `printf` frame [1]. So the first argument (the format string) is 8 bytes from the base of the `printf` frame, the second argument is 12 bytes from the base, and so on. Therefore, by using the right format string, we can inspect the content of the stack.

```
printf ("%08x␣%08x␣%08x␣%08x␣%08x\n");
```

This instructs `printf` to retrieve five parameters from the stack and display them as 8-digit padded hexadecimal numbers.

2.4 Viewing Memory at Any Location

By exploiting this vulnerability, it is possible to view memory at any location. Since we cannot change the code, we can only supply format string. In the following example, the format string is stored in a buffer, which is located on the stack.

```
int main(int argc, char *argv[])
{
    char buf[100];
    // ... other variable definitions and statements

    scanf("%s", buf); // getting a string from user
    printf(buf); // vulnerable place

    return 0;
}
```

If we use `printf(%s)` without specifying a memory address, the target address will be obtained from the stack by the `printf` function. Now, if we can force `printf` to obtain the address from the format string (also on the stack), we can control the address. For example :

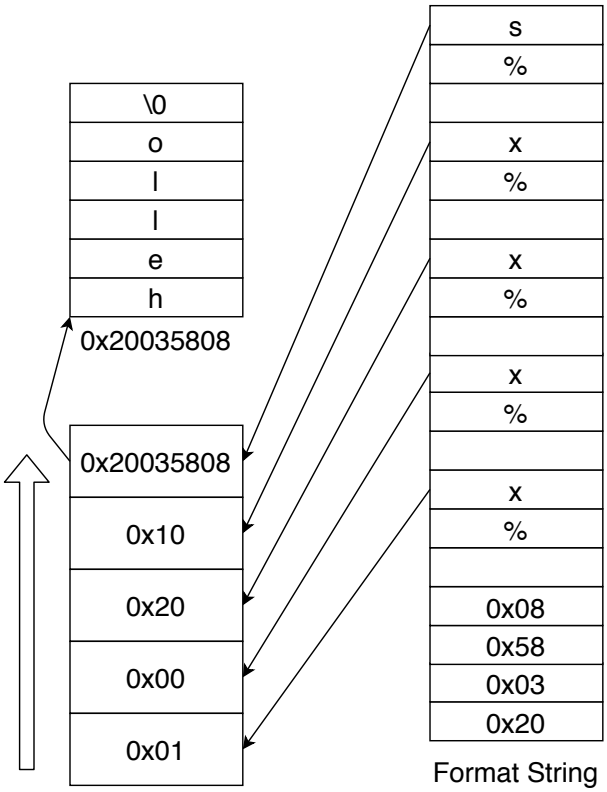
```
printf ("\x20\x03\x58\x08␣␣%x␣%x␣%x␣%x␣%s");
```

The string `\x20\x03\x58\x08` represents the four bytes of the target address. In C language, \x20 in a string tells the compiler to put a hexadecimal value `0x20` in the current position. The value will take up just one byte (`%x` causes the stack pointer to move towards the format string). Here is how the attack works if `buf[]` contains the previous format string :

Basically, we use four `%x` to move the `printf()`'s pointer towards the address that we stored in the format string. Once we reach the destination, we will give `%s` to `printf`, causing it to print out the contents in the memory address `0x20035808`. The function `printf` will treat the contents as a string, and print out the string until reaching the end of the string (i.e. 0).

The key challenge in this attack is to figure out the distance between the `buf[]` and the address passed to the `printf` function. This distance decides how many `%x` you need to insert into the format string, before giving `%s`.

1. Recall : the base of the calling frame is stored at an offset of 0 bytes and the return address at an offset of 4 bytes.



2.5 Writing an Integer to Memory

In addition to read memory, it is also possible to write integer at a specific memory location by using the `%n` format parameter. Here is the example program.

```
#include<stdio.h>

int main() {
  int i;
  printf("You␣are␣%ncool\n", &i);
  printf("%d\n", i);
}
```

The output of the program above is as follows :

```
You are cool
8
```

The first `printf` prints "You are cool". The second `printf` prints 8 as there are 8 characters printed (the 8 characters are "You are " with spaces) before $\%n$ in first `printf` and `i` is set to 8 by the first `printf`.

Therefore, by using the same approach as that for viewing memory, we can compromise `printf` to write an integer into any location. `%n`, and the contents at the address `0x20035808` will be overwritten. With this attack, we can thus overwrite variables, return addresses on the stack, function pointers, etc.

Nevertheless, it seems that the value written is determined by the number of characters printed before the `%n` is reached. We can change this status quo and write arbitrary integer values as follow (more information in exercise 5c) :

1. Use dummy output characters. To write a value of 1000, a simple padding of 1000 dummy characters would do.
2. To avoid long format strings, we can use a width specification of the format indicators.

3 Lab environment

This lab will be realized in a virtual machine (Ubuntu 16.04), emulated with VirtualBox. The credentials on the virtual device are the following ones :

1. User ID : `seed`
2. Password : `dees`

3.1 Security

Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To be able to realize your attacks, you must disable them first.

Address Space Randomization. Ubuntu, as several other Linux-based systems, uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult. However, knowing theses addresses is one of the critical steps of buffer-overflow attacks. You can disable this feature by using the following command

```
$>sudo sysctl -w kernel.randomize_va_space=0
```

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has changed now. The binary images of the different programs (and shared libraries) must declare whether they require executable stacks or not by marking a field in the program header. This marking is done automatically in the recent versions of `gcc`, and by default, the stack is set to be non-executable. To allow executable stacks, use the following option when compiling a program

```
$>gcc -z execstack -o program program.c
```

3.2 Lab Material

The different files that will be used in this lab can be found on the webpage of the course [2].

Download the archive `exploits.zip` and decompress it. It contains the victim program `server_udp.c`.

To get all files on the VM, download the archive exploits.zip from eCampus using Firefox on the VM (or alternatively on the host and tranfer them to the VM via SSH or drag & drop) and decompress it.

All necessary tools are installed on the reference virtual marchine.
Note : *GDB* is not required !

4 Lab Tasks

4.1 Task 1 - The Vulnerable Program

Open the file `server_udp.c`. This program is used by a company as simple test server. Compile it with the adequate flags and run it as *sudo*.

You can compile it by the following command :

```
$>gcc -z execstack -o server_udp server_udp.c
```

Once the server is launched, we can connect to the server using the `nc` command, where the flag `-u` means UDP (the server program is a UDP server).

```
$ nc -u 127.0.0.1 9999
```

Yon can type any message on the client side ; the server program is supposed to print out whatever is typed by you. However, a format string vulnerability exists in the server program's.

2. https://www.ecampus.ulg.ac.be

Question 1 What is the the vulnerability of the program ? Explain.

4.2 Task 2 : Understanding the Layout of the Stack

To succeed in this lab, you need to understand the stack layout when the `printf` function is invoked inside `printmsg`. Figure 1 depicts the stack layout. You need to conduct some investigation and calculation, and then answer the following questions :

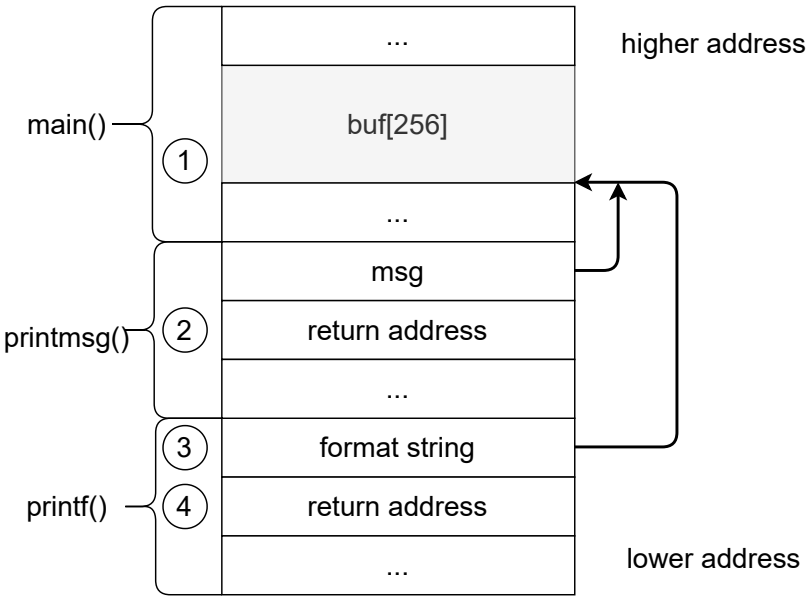


FIGURE 1 – Stack Layout

Question 2a What are the memory addresses at the locations marked by (1), (2), (3) and (4) ? Enter addresses and the offset (in hex) from the `msg` variable in the following table. Justify your answers.

Element	Address	Offset from `msg` (in hex)
`buf` (1) `msg` `Return Address` (2) `Format String` (3) `Return Address` (4)		0x00

TABLE 2 – format parameters

Question 2b What is the distance between the locations marked by (1) and (3) ?

4.3 Task 3 : Crash the Program

Try to crash this program by providing a specific input.

Question 3 Explain why your input causes the program to crash.

4.4 Task 4 : Print Out the Server Program's Memory

You task is to print specific memory location of the server program. There is a secret message stored in the heap area, and you know its address ; your job is to print out the content of the secret message.

To achieve this goal, you need to place the address (in the binary form) of the secret message in your input (i.e., the format string), but it is difficult to type the binary data inside a terminal. We can use the following commands do that.
Note : Use python2 instead of python3.

```
$>python -c 'print("\xDD\xCC\xBB\xAA%.8x%.8x")' | nc -u 127.0.0.1 9999
// Or we can save the data in a file
$>python -c 'print("\xDD\xCC\xBB\xAA%.8x%.8x")' > input
$>nc  -u 127.0.0.1 9999 < input
```

For this part, it should be noted that most computers are little-endian machines, so to store an address `0xAABBCCDD` (four bytes on a 32-bit machine) in memory, the least significant byte `0xDD` is stored in the lower address, while the most significant byte `0xAA` is stored in the higher address. Therefore, when we store the address in a buffer, we need to save it using this order : `0xDD`, `0xCC`, `0xBB`, and then `0xAA`.

Question 4 Which command did you use ? Explain your methodology.

4.5 Task 5 : Change the Server Program's Memory

The objective of this task is to modify the value of the `target` variable that is defined in the server program. This part is divided into 3 sub-tasks.

4.5.1 Change the value to a different value.

In this sub-task, you must change the content of the `target` variable to something else (any value is correct).

Question5a Which command did you use ? Explain your methodology.

4.5.2 [Bonus] Task 5.B : Change the value to `0x999`.

In this sub-task, you must change the content of the `target` variable to a specific value `0x999`.

Question5b Which command did you use ? Explain your methodology.

4.5.3 [Bonus] Task 5.C : Change the value to `0xdeadbabe`.

Then, you need to modify the value with a bigger number `0xdeadbabe`.

Question5c Which command did you use ? Explain your methodology.

Note : In a format string attack, this value is the total number of characters that are printed out by the `printf` function ; printing out this large number of characters may take hours. You need to use a faster approach. The basic idea is to use `%hn`, instead of `%n`, so we can modify a two-byte memory space, instead of four bytes. Printing out 2^{16} characters does not take much time. We can break the memory space of the target variable into two blocks of memory, each having two bytes. We just need to set one block to `0xdead` and set the other one to `0xbabe`. This means that in your attack, you need to provide two addresses in the format string.

4.5.4 [Bonus] Task 6 : Inject Malicious Code into the Server Program

Now we are ready to go after the crown jewel of this attack, i.e., to inject a piece of malicious code to the server program, so we can view the content of a protected file. To do this task, we need to inject a piece of malicious code, in its binary format, into the server's memory, and then use the format string vulnerability to modify the return address field of a function, so when the function returns, it jumps to our injected code.

This time, let's inject a shell code which performs `"cat /etc/passwd"`. The code is provided in `catpasswd.c`.
Hint1 : For this exercise, you need to change the pointer of the return address to the location of shellcode. The difficulty is to find the exact address of the shellcode. Fortunately, we can use NOP operations ($\backslash x90$) to make the process easier. Indeed, if we fall in the middle of the NOPs, then the program will go to the next instruction, which is a NOP, then the next one, ..., until it arrives to the beginning of the shellcode. Then, it will execute the shellcode.

Hint2 : Check section 3.4.1 of the following document [3].

Hint3 : You can use `xxd` on the `input` file to verify the offset.

Hint4 : We are here to help.

Question6 Which command(s) did you use ? Explain your methodology.

5 Submission

The submission of your lab is subject to the following rules :

1. an archive named **uliege_id.zip**, which contains *answers.txt* with your answers, must be uploaded on the submission platform (see the Submission Platform) ;
2. the deadline is **February, 19th, 12h30**.

3. https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf