**Computer Security**

**Homework 5**

**Saiteja Enimidigandla**

**saitejae**

**50441788**

**a.**

This paper mainly focuses on SSL vulnerability MiTM (Man in the middle attack) on mobile platforms. The authors examined and analyzed experimentally over 100 popular android applications static and dynamically to expose SSL certificates and their vulnerabilities.

A major alarming issue notified by the authors is that the sensitive data is being exposed such as credit card information, passwords, and other details like user credentials, location, IMEI, and IMSI information. This described the analysis and the confidential data leakage from the mobile apps. This SSL vulnerability is evaluated and exposed by conducting experiments on the android apps using several criteria like hostname verification, SSL certificate authority, permissions, etc. via both static and dynamic analysis to meet the current day scenario risk assessments.

**b.**

The authors conducted analysis on selected top 100 apps to test vulnerability using static and dynamic analysis. Static analysis revealed that half of the apps accept all certificates and fail verification of hostname. While dynamic analysis reveals that there is mainly three possible Man in the middle attacks, named S1, S2, and S3. S1 corresponds to e adversary having his certificate installed on the user’s device, S2 corresponds to the adversary presenting an invalid, self-signed certificate, and S3 corresponds to the adversary presenting a certificate with a wrong Common Name and signed by the root certificate authority. 90% of the apps establish HTTPS connections under attack, and 25% of the apps are prone to S2, and S3 attack scenarios where permissions are requested beyond their usage requirements. It also stated that many of the apps that are equipped/installed with an SSL certificate are still vulnerable to these man-in-the-middle attacks due to wrong hostname verification. And also, about 50% of the apps are vulnerable to trust managers accepting invalid or self-signed certificates. It also stated the implementation of SSL pinning guards against Man in the middle attack. In order to prevent the attacks analyzed in this paper, a suggestion is that developers should be given enough access to the apps such that they can fix the vulnerability before its actual use by the users.

**c).**

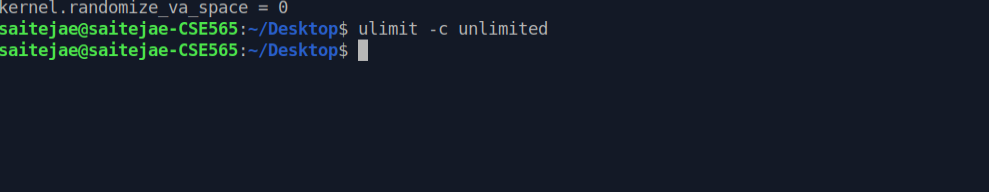
Behind the scenes of mobile apps used by us, we are unaware of what security mechanism is installed in them to ensure security like SSL digital certificates. We observe some security mechanisms like SSL digital certificate to ensure security which is HTTPS. At the time of the release of this paper, from the above analysis and observations, we can say that SSL is prone to attacks that affect sensitive data. Currently, most applications are using customized SSL code to protect or ensure strong security against man-in-the-middle attacks. Even though SSLs are installed, vulnerability still exists but decreases compared to the current and earlier security mechanisms. SSL pinning is one of the mechanisms that help in preventing these attacks.

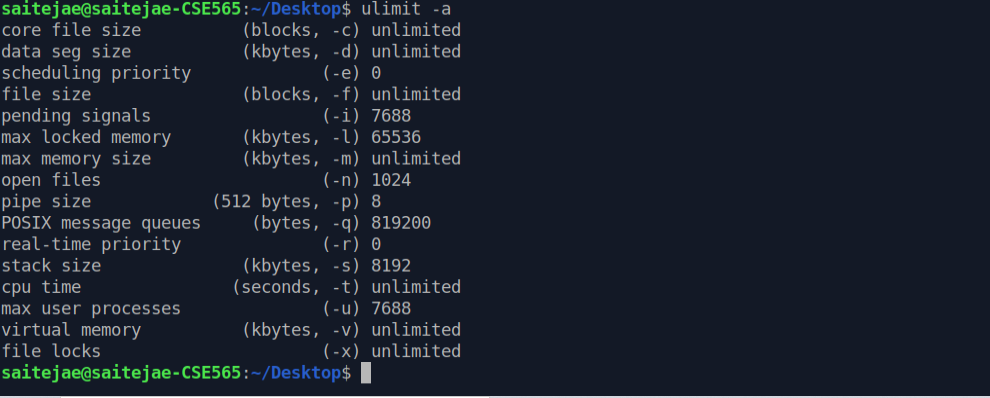
**02.**

**Part-1:**

By default, core files might not be created when a program crash. Enabling memory dumps:

Using ulimit:

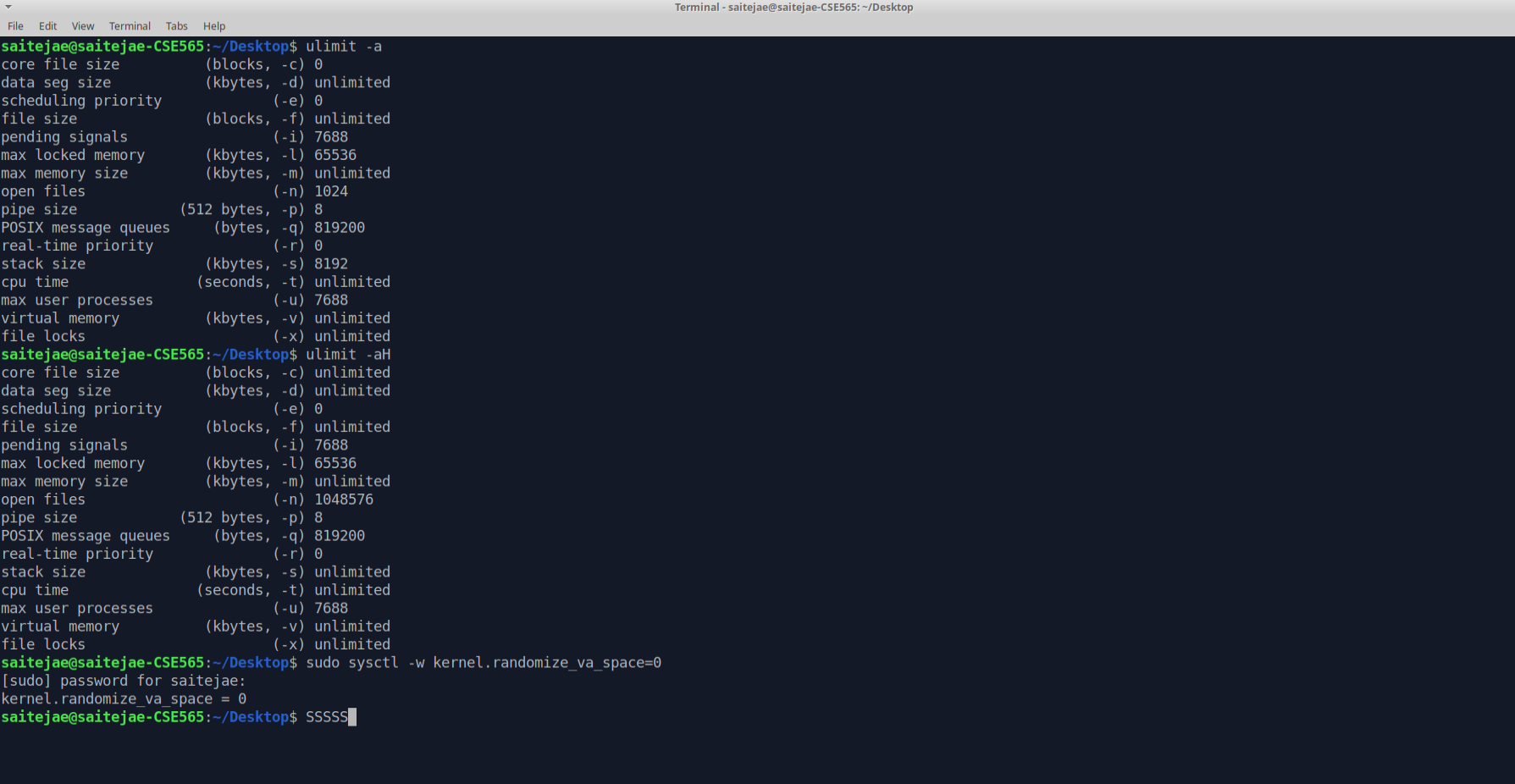




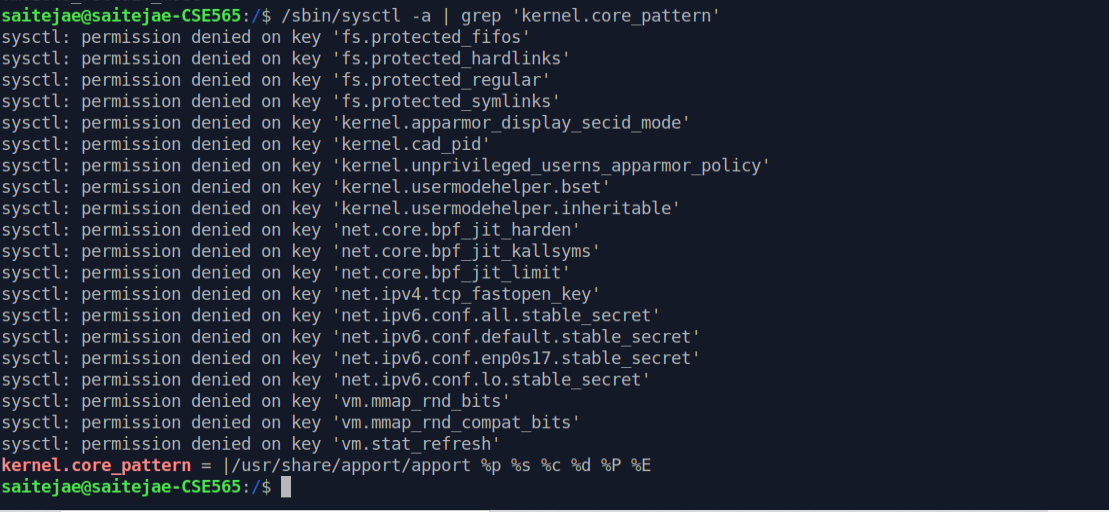
Disabling address randomization space by the below command:

sudo sysctl -w kernel.randomize\_va\_space=0

if this is turned off, the address space is static.



Core files dump location:



1. **Determined value for XX in vulnerable.c is 43**

The proof for the value obtained is mentioned and described below.

For simplicity and execution, I created a shell script named Makefile. Commands can be run using make command.

Below are the execution commands and their explanation:



**Clear** – clears the screen

**sudo sysctl -w kernel.randomize\_va\_space=0** - sets the address randomization off such that addresses won’t change for each execution.

**gcc -m32 -o vulnerable -z noexecstack -fno-stack-protector -g -ggdb vulnerable.c** – compilation of vulnerable program using 32-bit gcc compilation.

Noexecstack – no executable stack

-fno-stack-protector - guard variable onto the stack frame for each vulnerable function

**sudo chown root vulnerable**

**sudo chmod 4755 vulnerable –** giving execute access to vulnerable program.

**export MYSHELL=/bin/sh -**  setting myshell string.

**gcc -o exploit exploit.c – a** compilation of exploit program and storing the executable program in exploit

**./exploit –** exploit program execution

The below program using C creates the badfile for us, which will overwrite the return address with the system() address to access the shell with a buffer size of 19. The program is show below:

**vulnerable.c**



Starting from printing ‘A’ 19 times until we get a segmentation fault.

After following the above steps, we can find that the segmentation fault occurs exactly after 12 bytes i.e., 19 + 12 at the 31th byte. Here the address is overwriting the return address with our address of system(). Below diagram is shown address offset to better understanding.

So now we create a file named ‘exploit.c’ which will create the badfile which in turn is used by the vulnerable.c program to overwrite the return address with the system() and argument /bin/sh to access the shell.

System() address is at 35 offset address, we get exit offset location as 31 + 4

i.e. exit() – 35

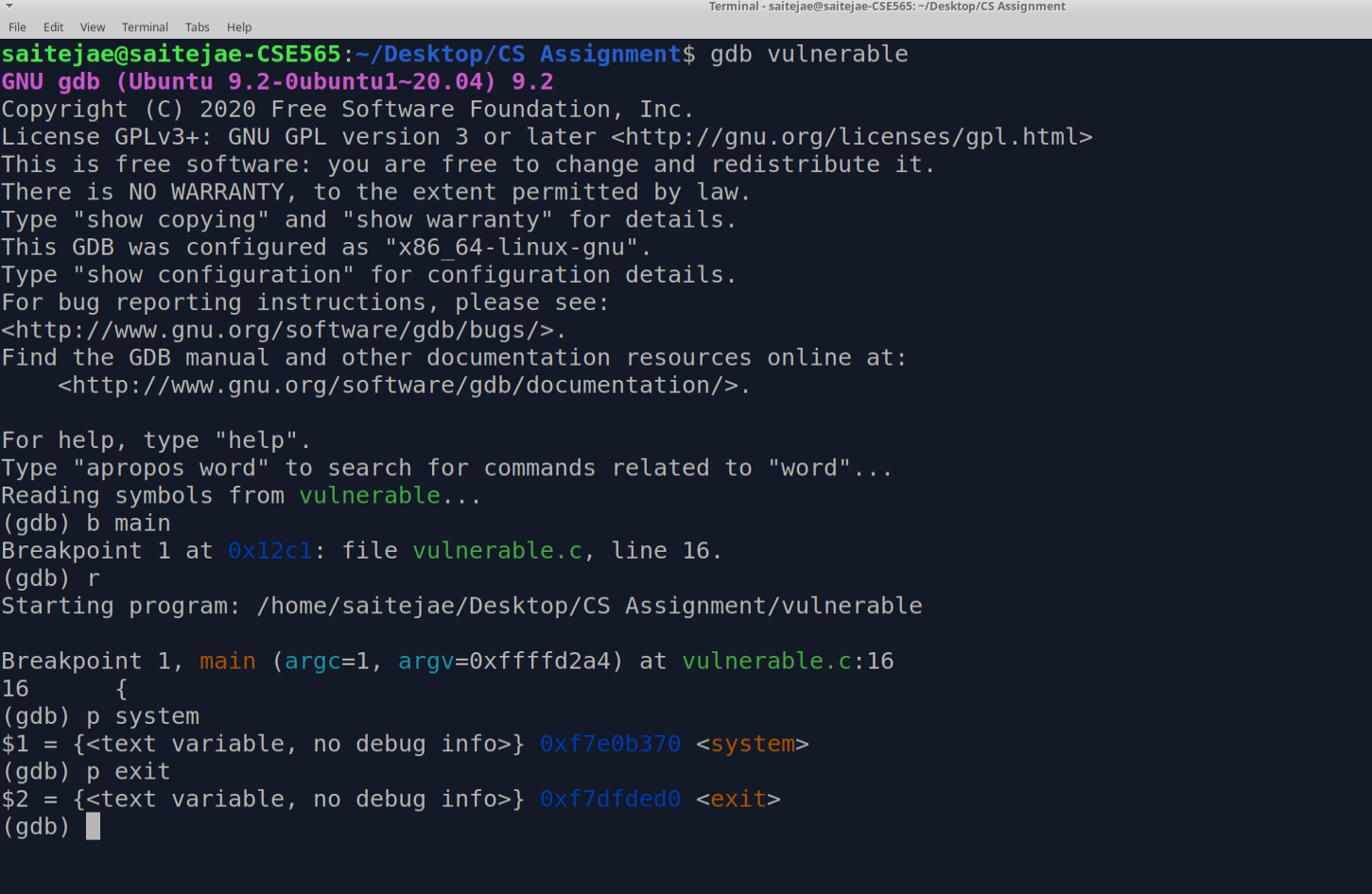
myshell – bin/sh at 35 + 4 = 39

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Buffer | Padding | System() | Exit() | MYSHELL |

0 19 31 35 39 43

N ow, it can be understood that XX value is 43

We will get the address of the system and exit as shown in the below diagram:



System (): 0xf7e0b370

Exit () - 0xf7dfded0

I create a new environment variable MYSHELL for the shell string.

Then I get the address of system() and exit() function. These are library functions already loaded

into the memory. The necessary commands are given below :

gdb retlib

b main

r

p system

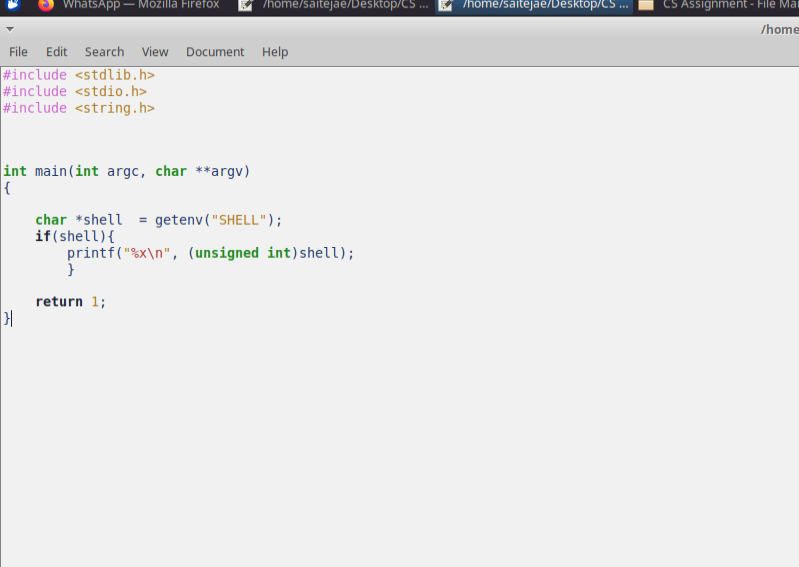
p exit

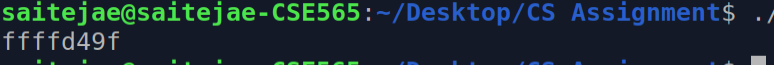
Exporting the shell string:

export MYSHELL=/bin/sh

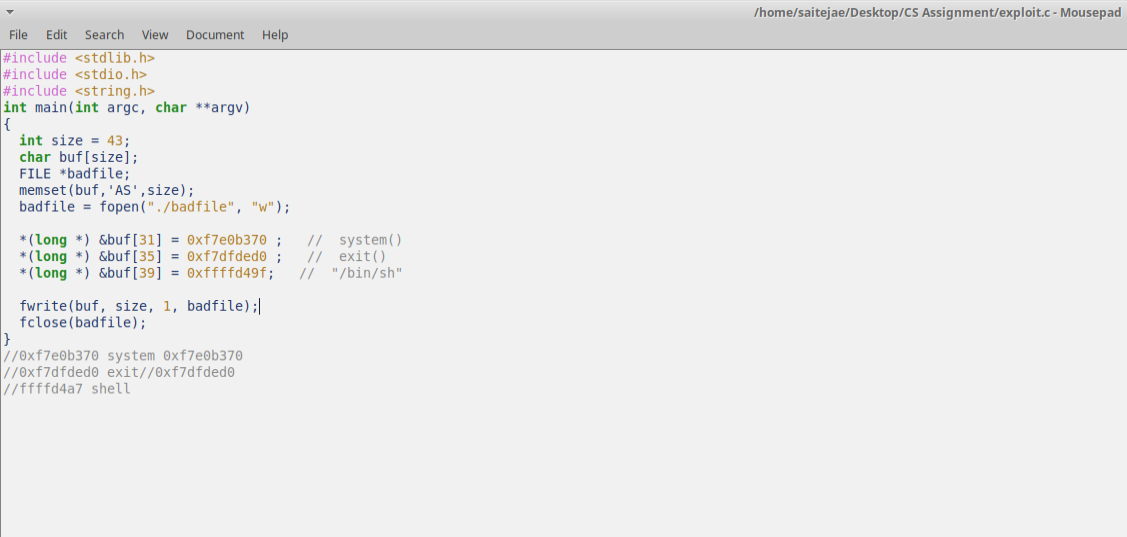
env | grep SHELL

Shell address obtained by a small program:





These addresses are stored in the buffer at their respective offset obtained above as shown in the below exploit program:



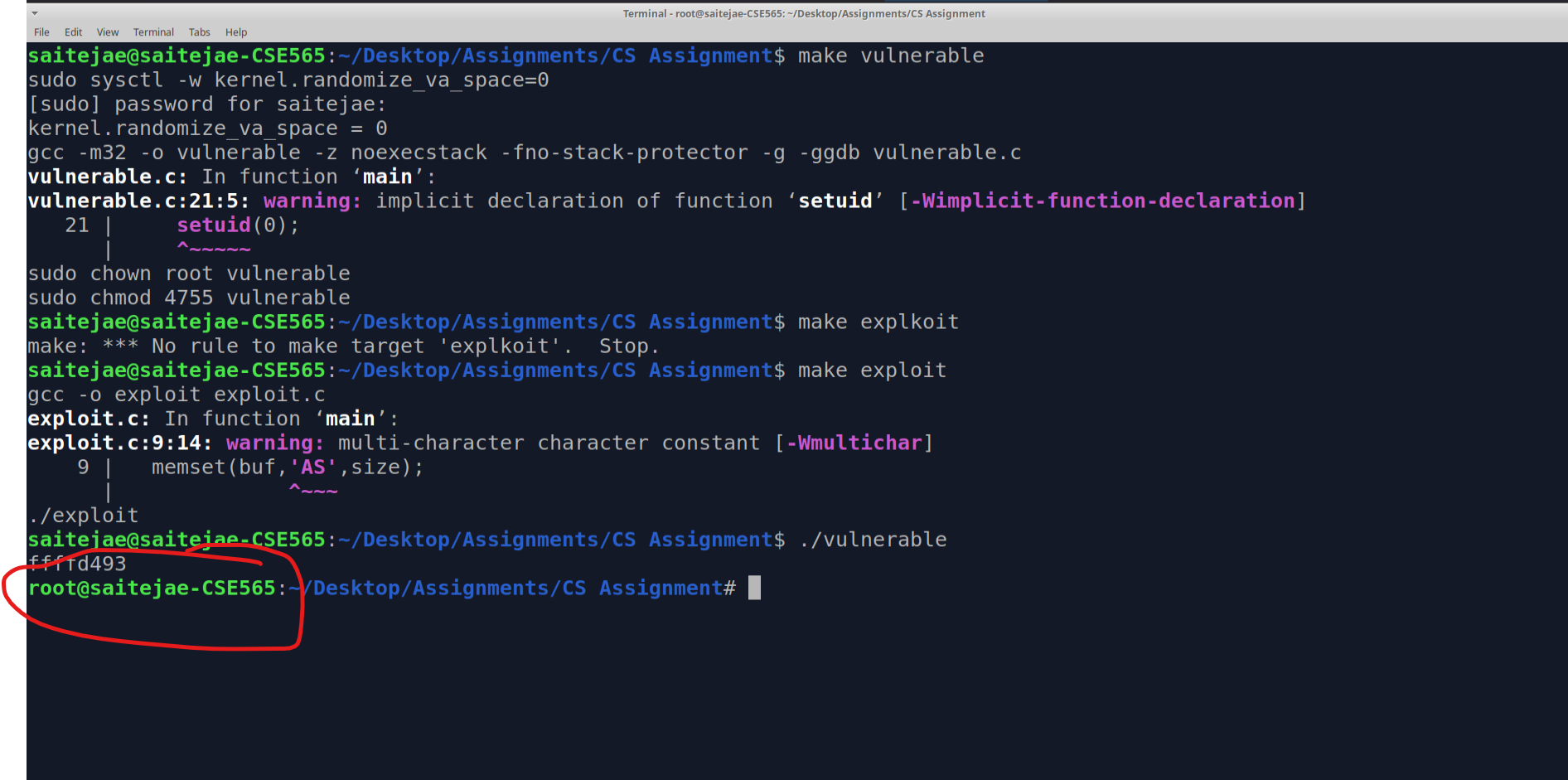
So, the final memory allocation looks like this,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Buffer | Padding | System()  0xf7e0b370 | Exit()  0xf7dfded0 | MYSHELL |

Now it can be understood why we used 43 in place of ‘XX’ in vulnerable.c file. The bytes that need to be overwritten are from 31 till 43.

With this information, we generate the badfile with the program we have created named exploit.c with the addresses.

Below is the execution process and



**Part 2:**

**Dealing with the address randomization:**

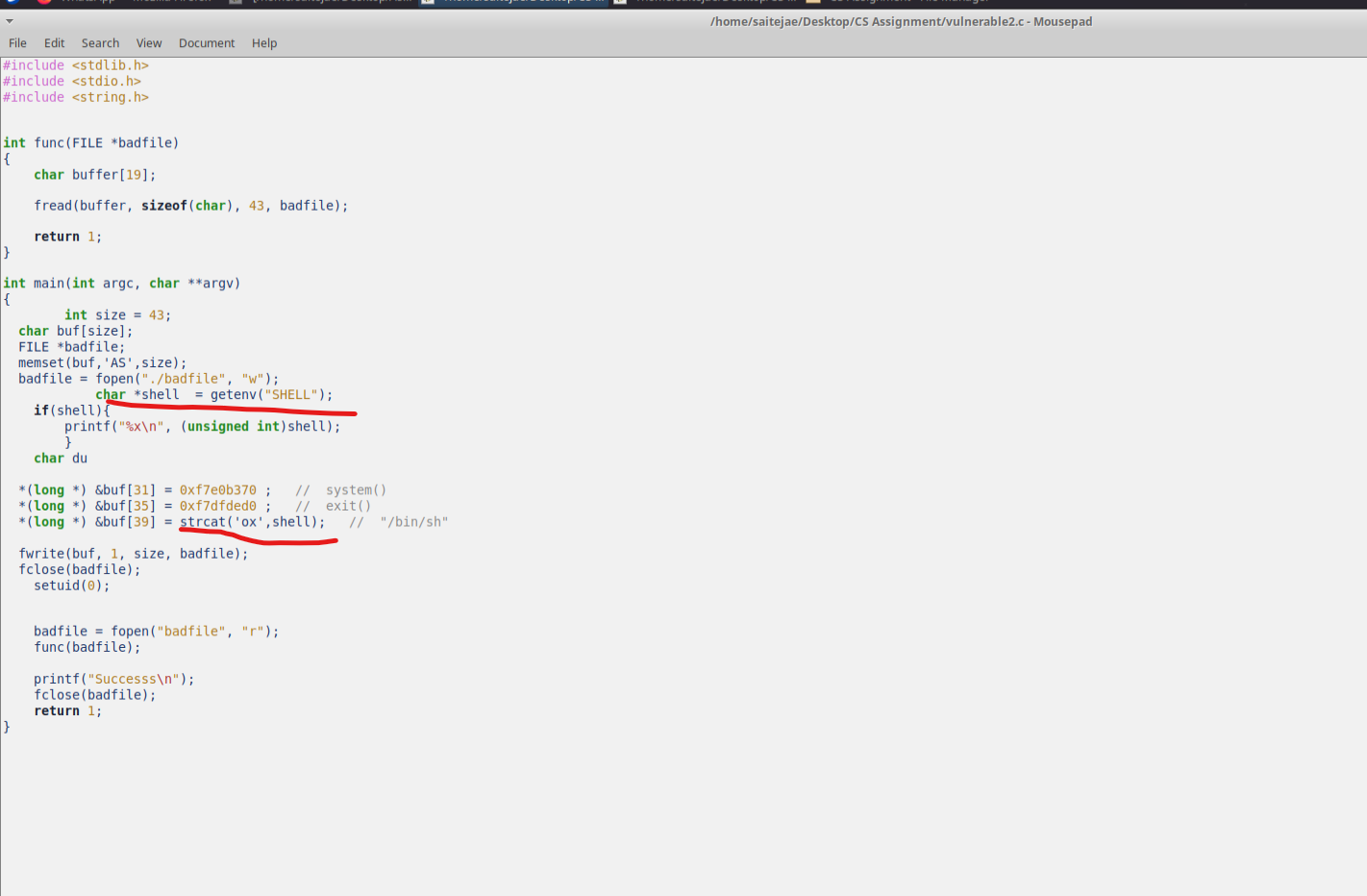
Address randomization can be turned on by the below command:

sudo sysctl -w kernel.randomize\_va\_space = 2.

The addresses change for every execution, especially the shell address.

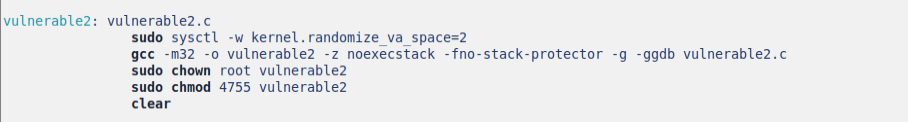
It is handled in the program for our return to lib attack as shown in the below program :

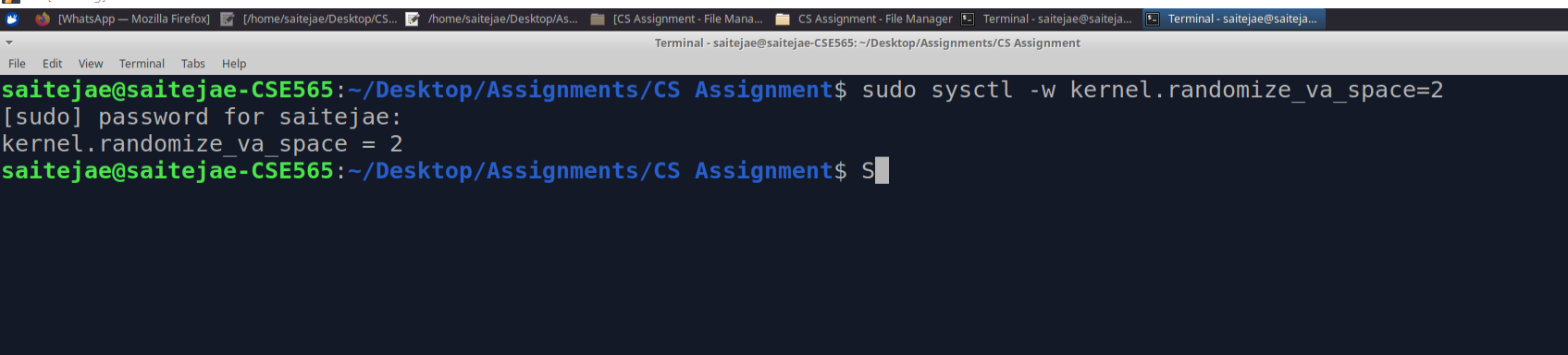
Same as the above vulnerable program, we passed shell address dynamically within the program:

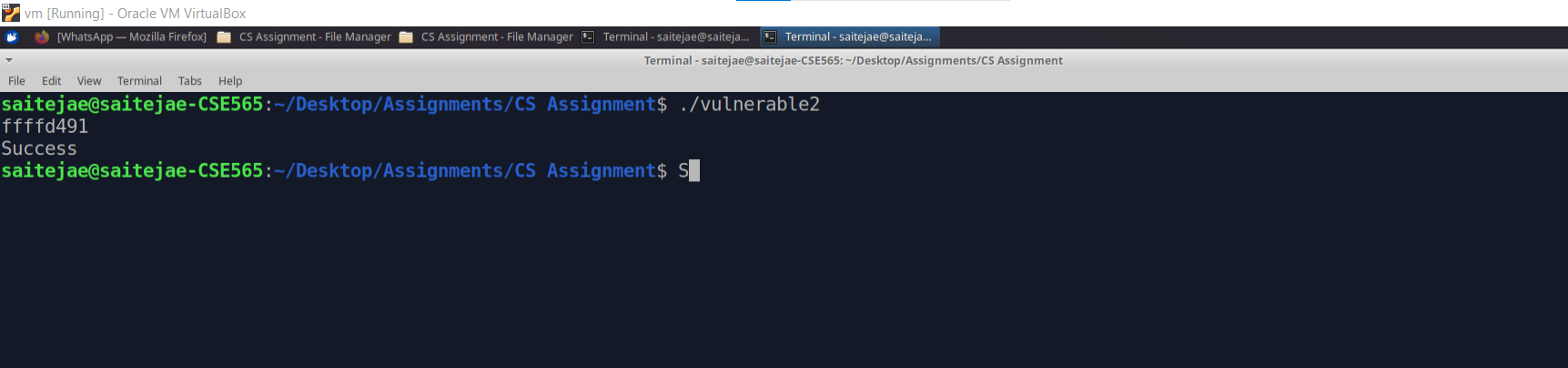


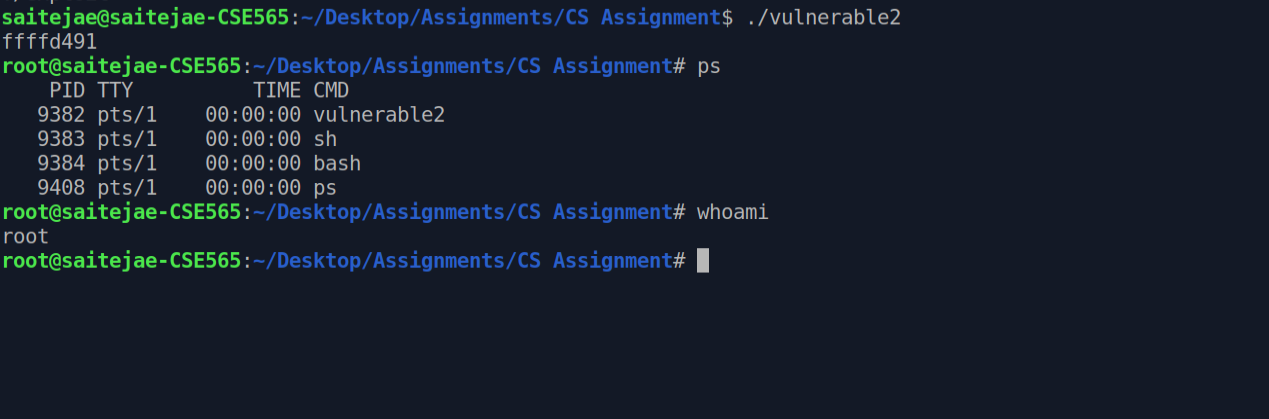
Execution is as followed:

Address randomization set to ON.









Full details are shown in the video.