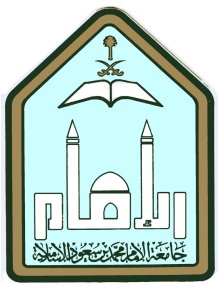
Al-Imam Muhammad Ibn Saud Islamic University
College of Computer and Information Sciences
Department of Computer Science



(2nd Semester 2022)
(1444/1444 – 2022/2023)

(CS442 – Software Security)

assignment-1

By:

Renad saud Al Hussain

440024784

Supervisor:

Dr. Basmah Alsouly

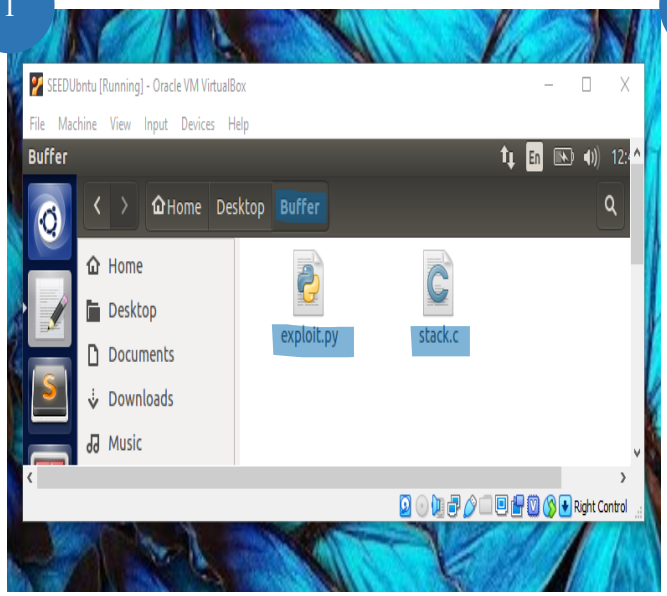
Task 1:

We provide you with a completed exploit code called “exploit.py”. You need to adjust the variables of the program accordingly and fill in any missing code to fulfill the buffer overflow attack.
After you finish adjusting the above program, run it. This will generate the contents for “badfile”. Then compile and run the vulnerable program "stack.c". If your exploit is implemented correctly, you should be able to get a shell:

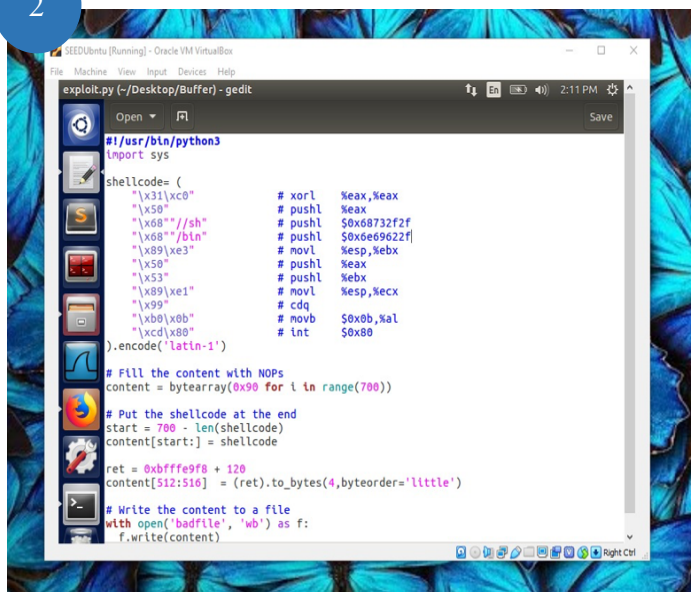
```
 $id
uid=1000(seed) gid=1000(seed) groups=1000(seed)
```

- First, I created a folder named buffer , in it exploit.py file and stack.c file.

1

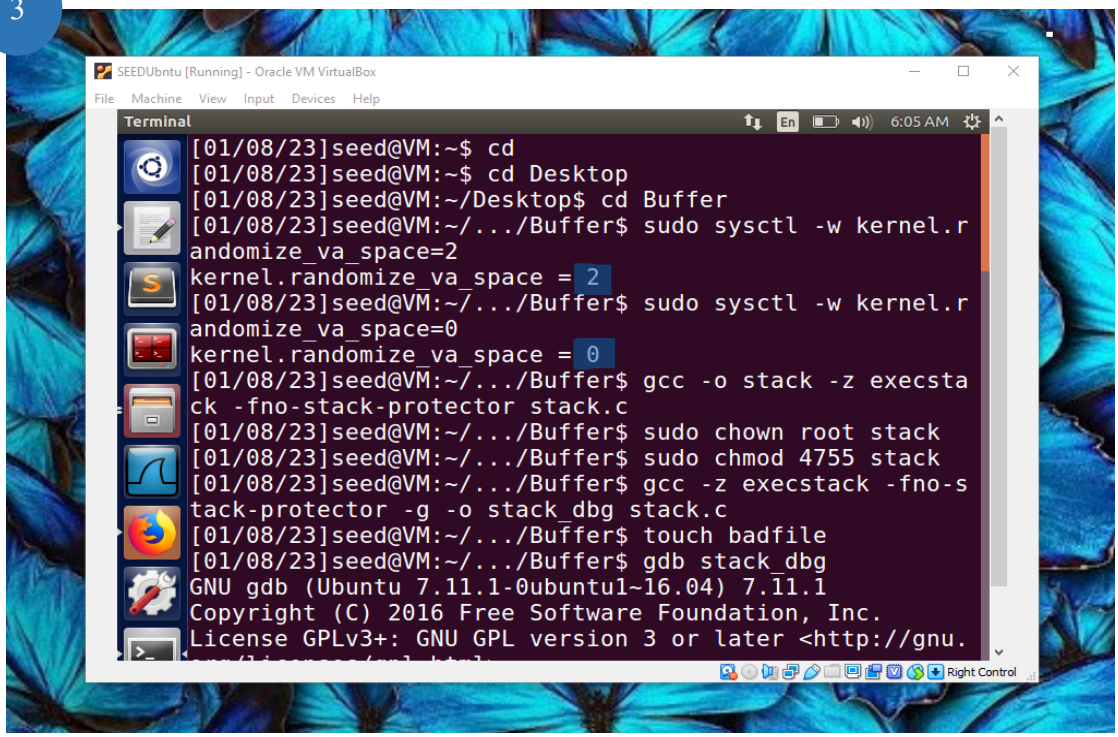


2



- I configured the variables of of the "exploit.py" file. and fill in any missing code to fulfill the buffer overflow attack .
- Also,I changed the buffer size to 700.
- And to run Python I used the method of adding "#! /usr/bin/python3" to the top of the script, making the file executable with "chmod u + x exploit.py" and then running: ./exploit.py.

3



- I open the terminal, go to the location where the file is located.

- Added at randomize space=2 . When set to 2, both stack and heap memory address is randomized.
- Turned off randomize space =0. When set to 0, the address space is not randomized.
  The randomize the starting address of the heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer- overflow attacks.
  Address randomization is turned off so that the base address of the buffer remains. It makes the task simpler to guess the return address in the stack.

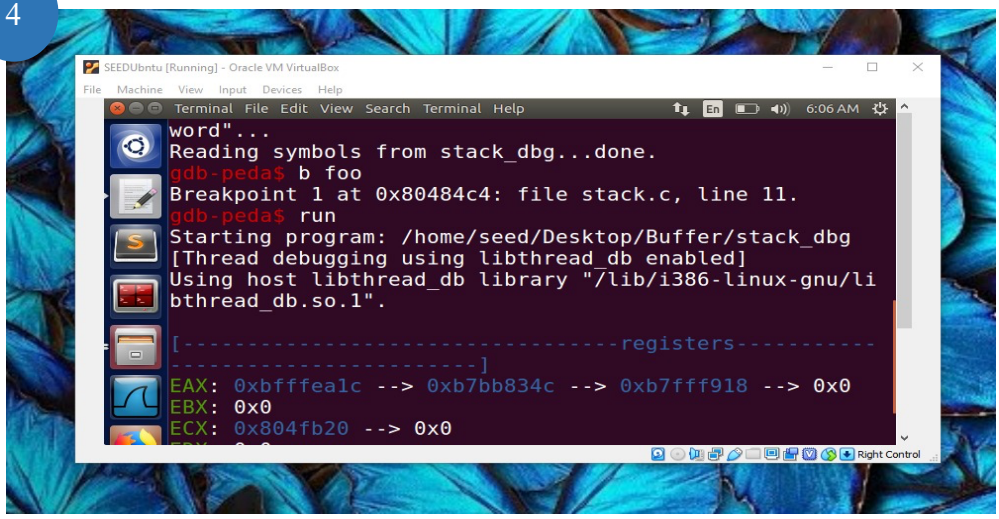
- After that, I write the gcc compiler to stack.c that we need to turn off: "-z execstack" which sets the stack as executable as we need to execute our own code on the stack. Also, we disabled stack guard protection during the compilation using the -fno-stack-protector option, which detects buffer overflows .
  The gcc compiler implements a security mechanism called Stack-Guard to prevent buffer overflows.In the presence of this protection, buffer overflow attacks will not work.
- I change owner of the executable "stack" to root.
- And change mode 4755 to make it Set-UID program (read-write executable).

To find out where the address is, use dbg Tools.

- I write the gcc compiler we discussed previously and use the –g for the debugging info used in gdb. (it is makes a Stack_dbg file").

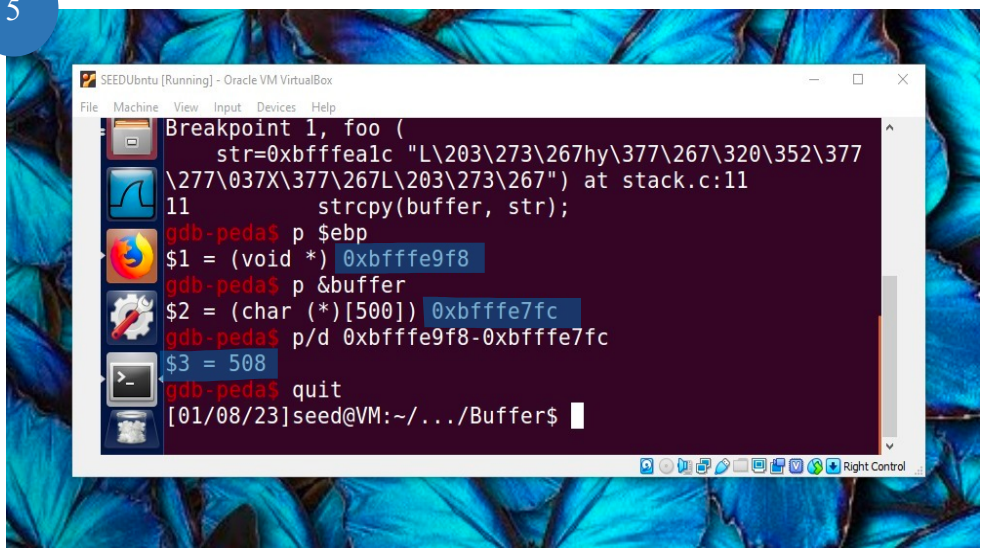
- I create a file with the name "badfile" in which there is nothing.
- Then , i run the dbg, write gdb and the name of the gdb file(stack_dbg).
- Here it started dbg.

4



- In order to placing a breakpoint, use p foo (name function), and then I do run the program.

5



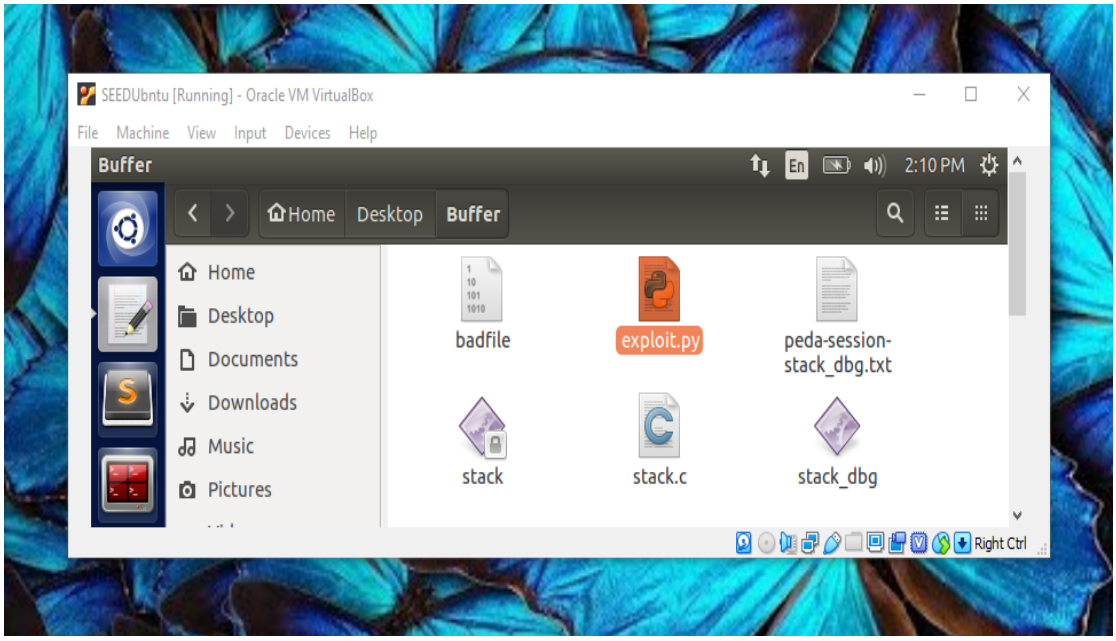
- To know the address the value of register (frame pointer), write p $ebp.
- To know the address the value of buffer (base of buffer), write p &Buffe.
- To know the distance bettwen the base of the buffer and the frame pointer, write p/d (valu frame pointer - valu base of the buffer).
- And to calculate the return address ($ebp + 4).
- "quit" to shut down gdb.

I used a gdb debugging tool to find the distance between the base of the buffer and the return address .

Given that we have access to the vulnerable code, we can use gdb to find the base address of the buffer, by first placing a breakpoint at function foo() (program will stop inside the foo function).

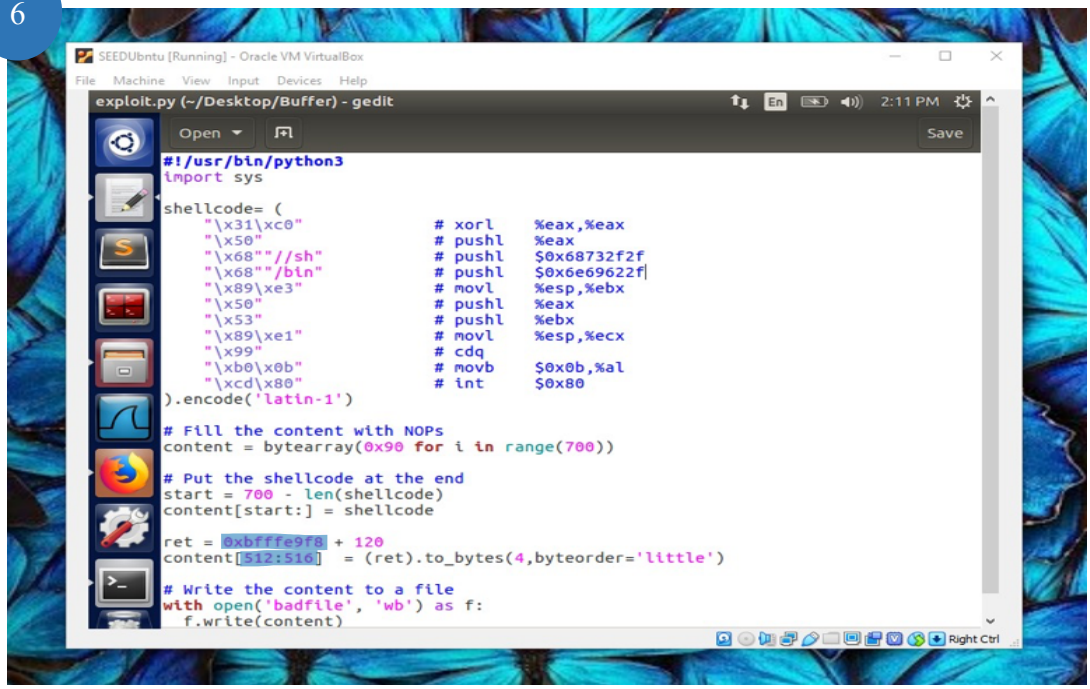
Then getting address of the frame pointer (ebp) = 0xbfffe9f8 and return address ($ebp + 4) = 0xbfffe9f8 + 4 (508+4=512)

Address of the base of the buffer ‘buffer[]’ = 0xbfffe7fc .



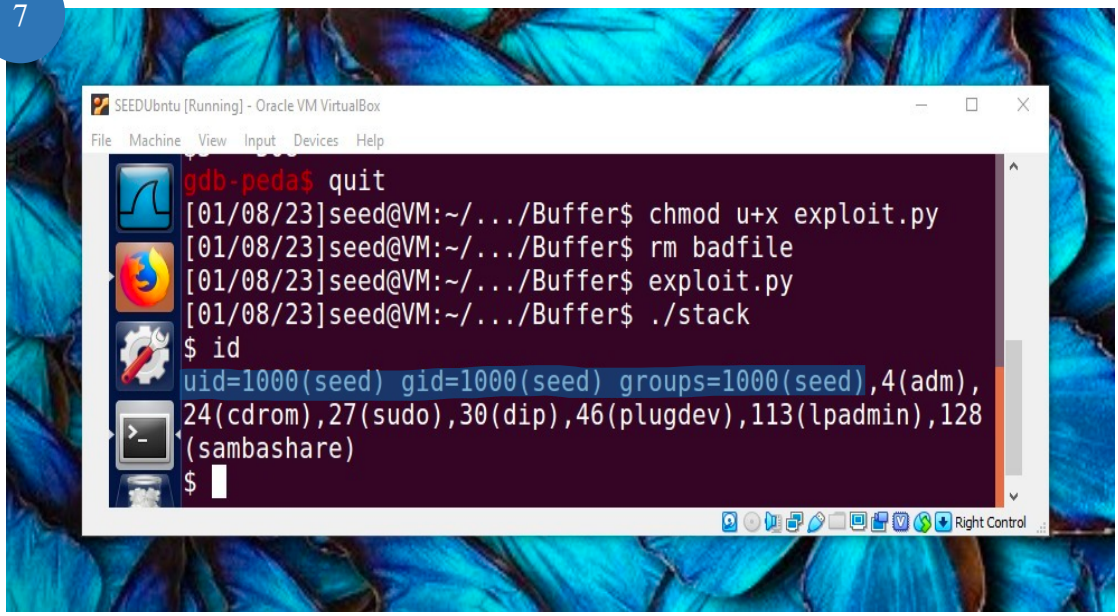
- Files after the running.

6



- I placed the return address byte by byte from content[512] to content[516], to_bytes() converts ret to bytes with length of 4 bytes.
- Also, I used first address “0xbfffe9f8 + 120”.

7



- To change the mod of the python file ،write chmod u+x exploit.py, also will grant only the owner (u) of that file execution (+x) permissions, instead of all users if we left the 'u.'
- rm badfile, It will delete a file badfile.
- exploit.py, Itwill run for the python file.
- ./stack, will run the stack program and then will everything works.

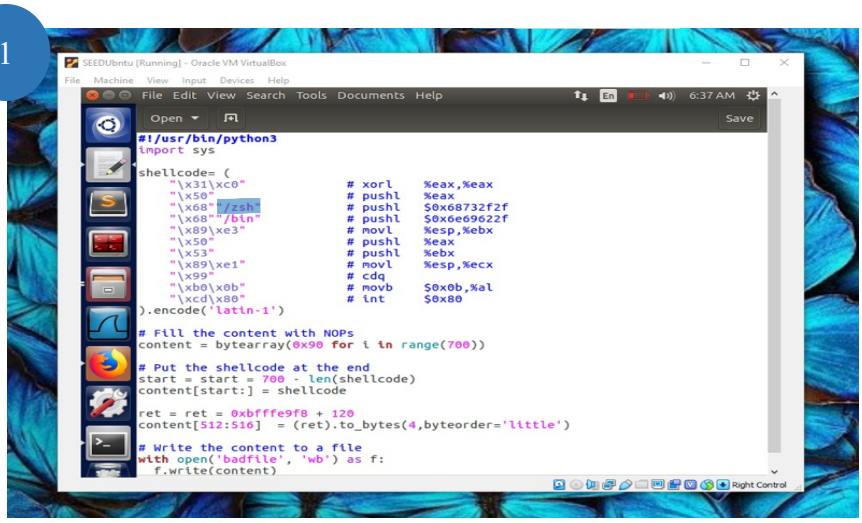
- The buffer overflow attack is successful.

Task2:

Find a root shell .
you should get the following output:
VM# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed)

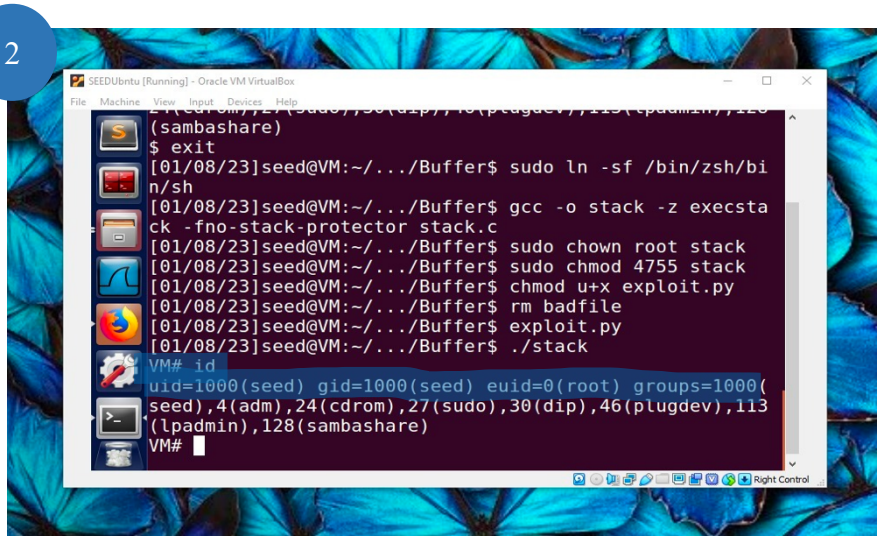
- I used the option "sudo ln -sf /bin/zsh /bin/sh."

1



- First, I modified the Python file and added /zsh" instead of "//sh.".

2



- fter that I wrote the option "sudo ln -sf /bin/zsh /bin/sh."
- And then write gcc compiler "gcc -o stack -z...", sudo chown root stack, sudo chmod 4755 stack, like the steps before.
- Shell is running as root.

Shellcode Simply it's a piece of code ("written in hex in our situation") that we use as a payload to execute something .

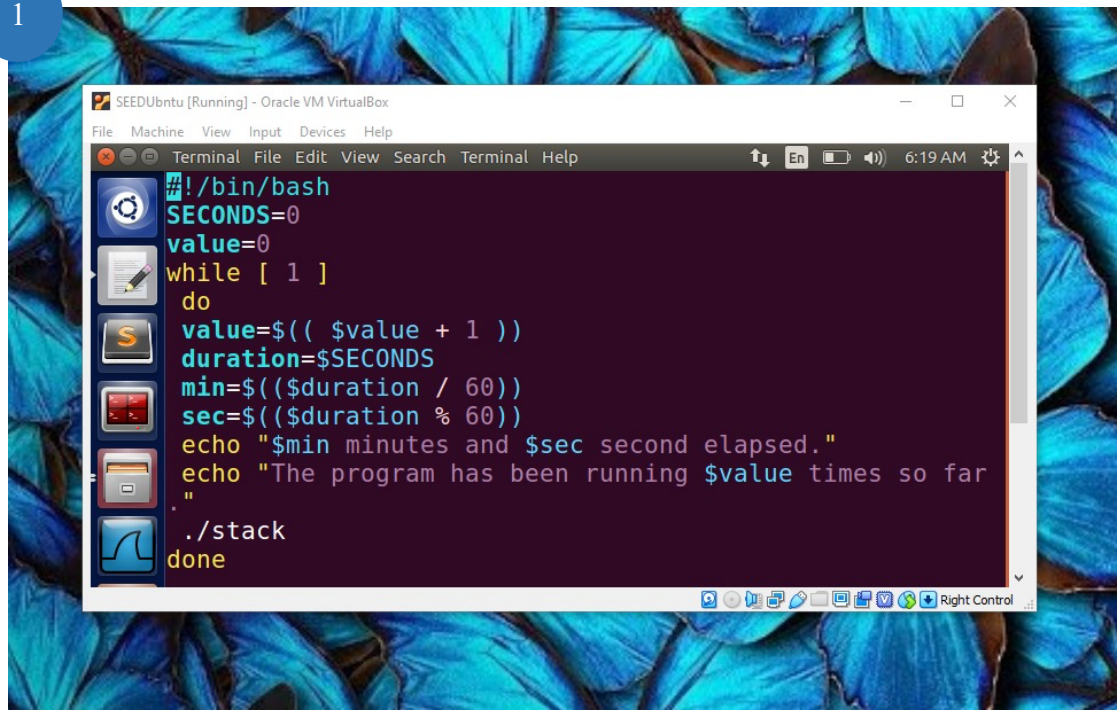
so if we execute shellcode that executes /bin/sh with the binary we will get a root shell.

Task3:

First, we turn on the Ubuntu's address randomization using the following command:
sudo /sbin/sysctl -w kernel.randomize_va_space=2

Then use the shell script figure out how to attack the vulnerable program repeatedly.

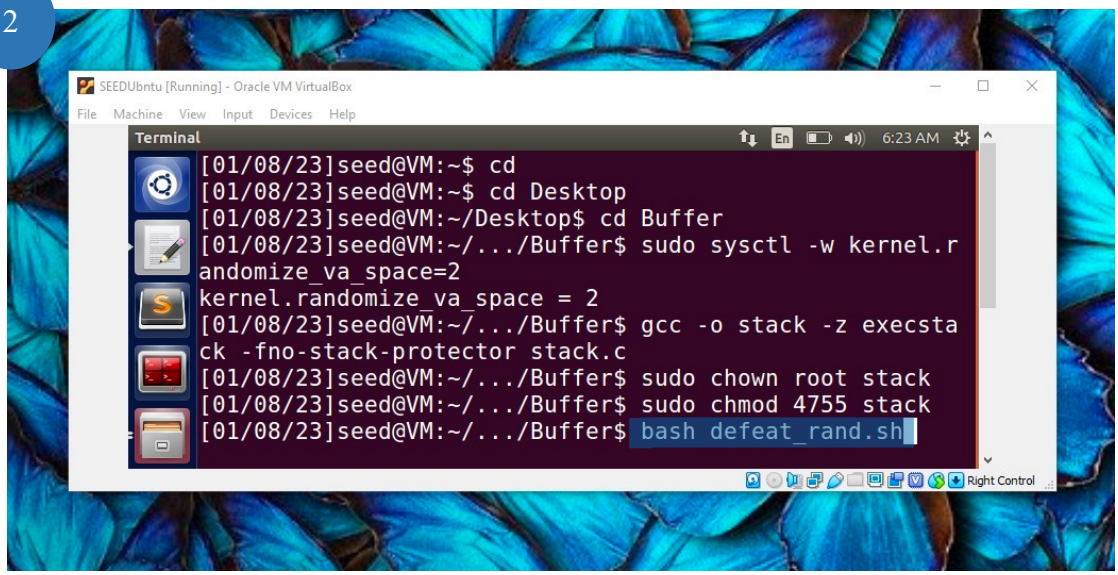
1



- First, I created a shell it running the vulnerable code in an infinite loop.
  Its aim is to run, if it does not work, it will run again in order for it to work.

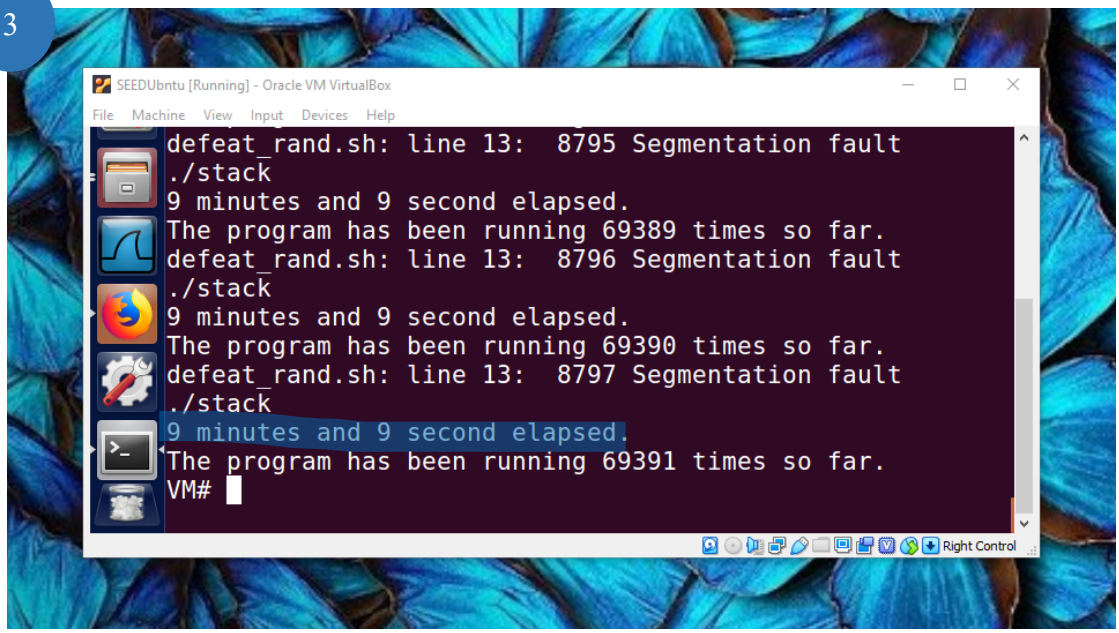
we can't use binary generated from directly from a C program as our malicious code, we will write the program directly using assembly language. We will write shellcode using assembly to launch a shell.
It's not difficult setting the registers, but in preparing data and finding the addresses of the data and making sure there are no zero in binary code.

2



- I open the terminal, go to the location where the file is located.
- Added at randomize space=2 . When set to 2, both stack and heap memory address is randomized.
- After that, I write the gcc compiler to stack.c.
- I change owner of the executable “stack” to root.
- And change mode 4755 to make it Set-UID program (read-write executable).
- Finally, I run the defeating randomization. (many times against randomly assigned stack address).

3



- On running the script for about 9 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).
- Been attack the vulnerable program repeatedly successfully.