CYSE 211

Mahish Mahendarkar

6/27/2021

Linux Buffer Overflow Lab 4

Lab 4 purpose:

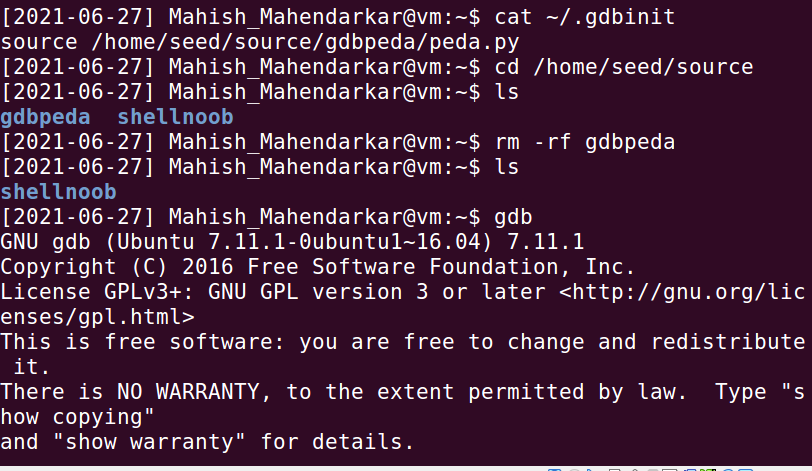
The purpose of this lab is to conduct a buffer overflow attack inside a Ubuntu system. There are provided source codes of both exploit, badfile (the attack vehicle), stack, and shellcode. The lab guides the user through the necessary settings to disable to allow the buffer overflow attack to be conducted inside of the Linux system.

---------------------------------------------------------------------------------------------------------------------

The BUF\_SIZE value for this Lab is 160

Pre-Lab configuration settings:

The first prelab configuration settings address the gdb program. The lab instructions instruct us to remove the PEDA folder from the location [~/.gdbinit]. The following screenshot shows this task being completed. The last command run was [gdb] and this validates that gdb is still able to run after the gdbpeda has been run.



Lab tasks:

Turning Off Countermeasures

This task describes the settings that need to be modified in order for the buffer overflow attack to occur. There are preset OS settings related to address space randomization, stack guard protection, non-executable stack, and the linking of a bin shell. These preset settings are meant to identify and prevent a buffer overflow attack from occurring.

Observations: This task discusses the commands to be run to address all the pre set settings but no commands are run in this task except for the command relating to the shell linking [$ sudo ln -sf /bin/zsh /bin/sh],

and the command relating to address space randomization

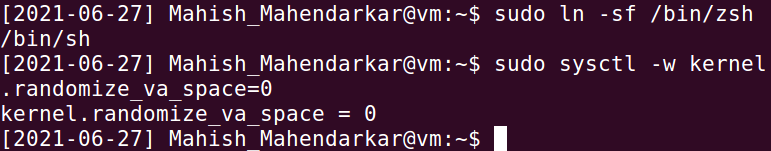
[sudo sysctl -w kernel.randomize\_va\_space=0].

The other commands discuss a way of modifying compiler settings during the compilation of a file[$ gcc -fno-stack-protector example.c]

and a non-executable stack compilation

[For executable stack: $ gcc -z execstack -o test test.c//For non-executable stack:$ gcc -z noexecstack -o test test.c].

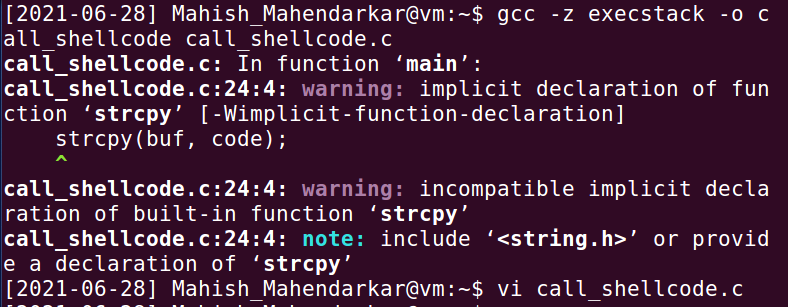
This will be done in later tasks.

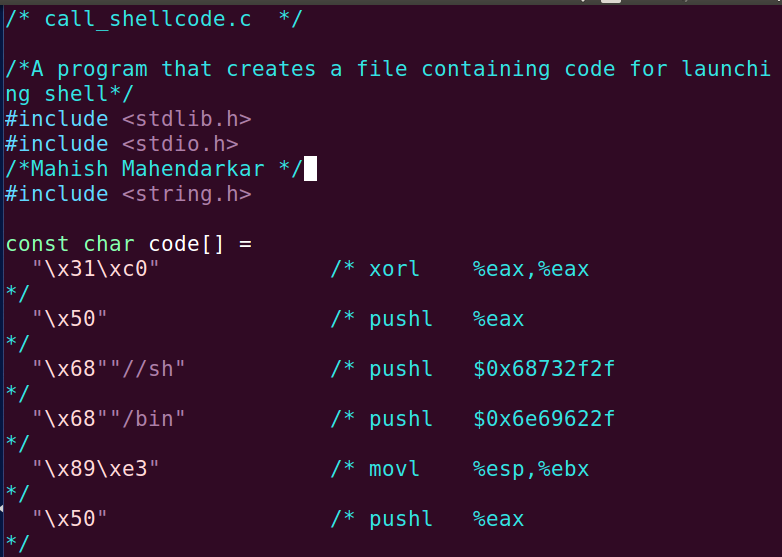


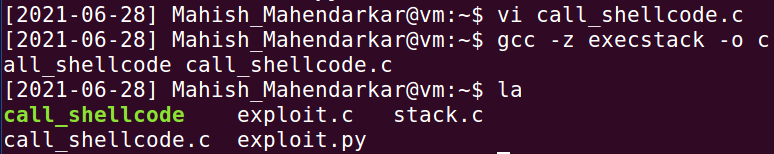
**Task 1 Running Shellcode**

This task asks us to note the observations and output after compiling the given shellcode file. The expectation prior to running this command is that the c program will be compiled into an executable process.

Observations: The provided files from the lab instructions website [https://seedsecuritylabs.org/Labs\_16.04/Software/Buffer\_Overflow/] were incorrectly constructed. The lab instructions show the string.h library included in [call\_shellcode.c] however upon running the file I received an error and I remedied this by correcting the [call\_shellcode.c] code. This is shown in the first two screenshots. The third screenshot shows the results of a successful compilation of the file [call\_shellcode.c], the key part is that a object file has been created and can be executed. The new object file is named [call\_shellcode].



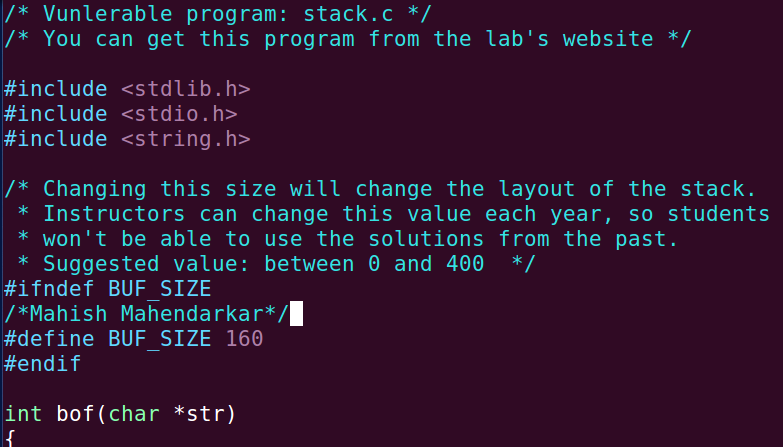


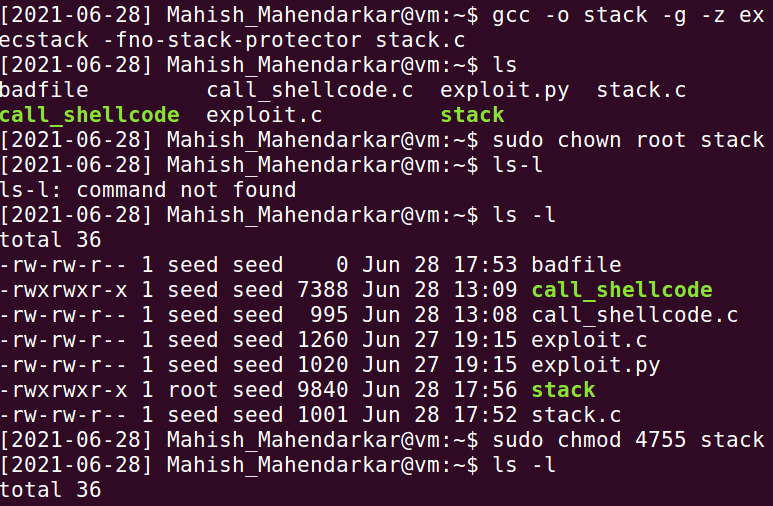


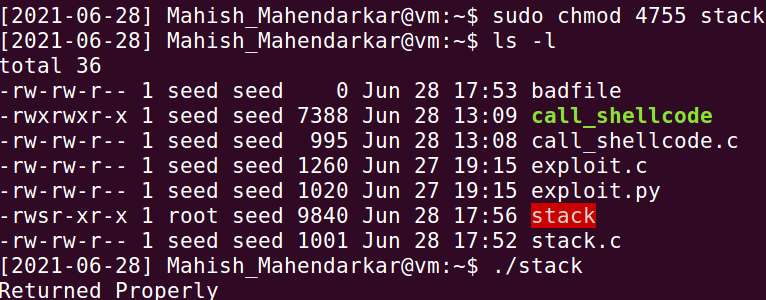
**Task 1 continued: The vulnerable program**

This task is about the [stack.c] file that has been provided. There is a buffer overflow vulnerability within this file at the [strcpy] line and this can be used to gain root privileges.

Observations: The first step is make sure the [BUF\_SIZE] global variable is 160 according to the lab instructions. Then the badfile can be created, this file will be edited in later steps. Next the stack security is turned off during compilation and the [stack.c] file can be converted to an object file with elevated permissions. The last screenshot shows the root ownership and permissions of all files specifically [stack]. The running of the stack object file returns [Returned properly] because that is one of the final lines of stack.c and as our badfile is currently empty the successful compilation of stack.c should have this statement printed by the terminal. Stack.c relies on badfile to exploit the buffer overflow vulnerability.



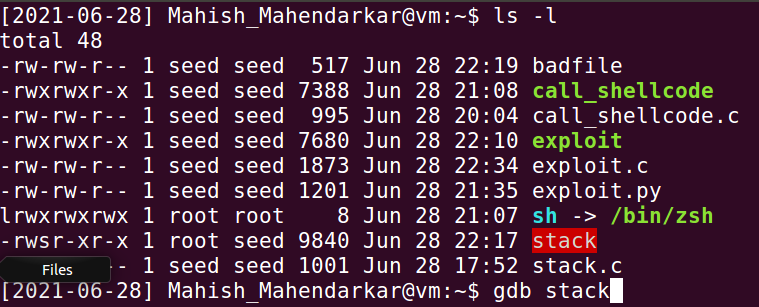


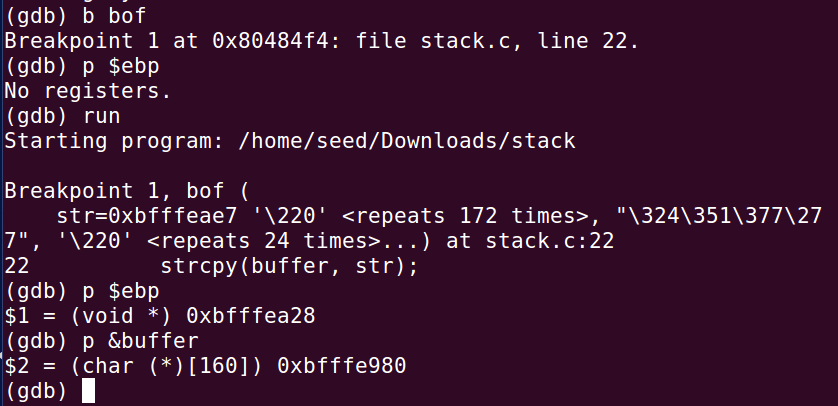


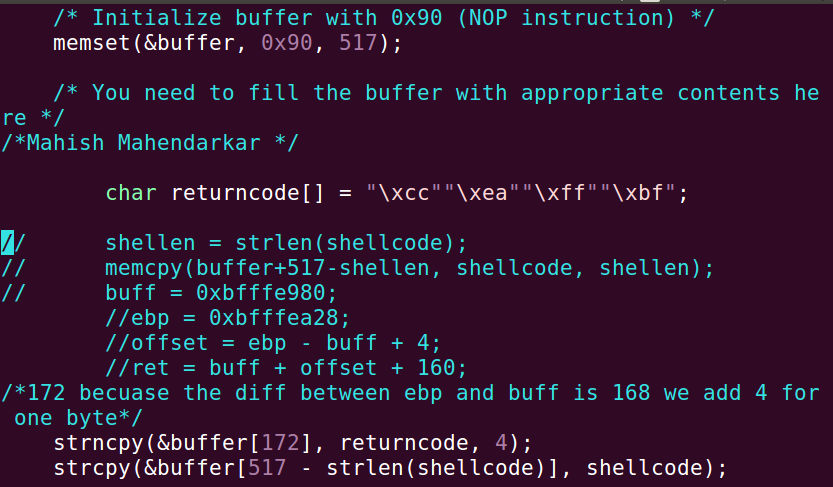
**Task 2 Exploiting the Vulnerability**

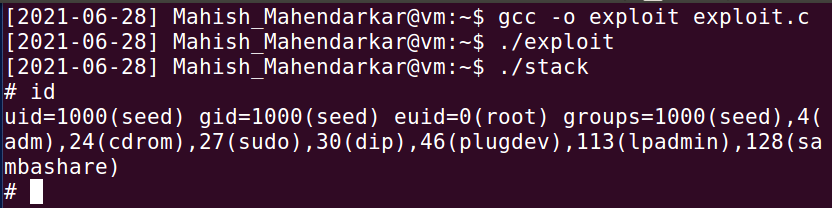
This task is the actual exploitation of the buffer overflow vulnerability. The return address should be overwritten by the provided assembly language code provided. The work required involves finishing this [exploit.c] source code file. The [exploit.c] file writes the contents of calculation into badfile and this sets us up for future steps. Upon successful compilation we expect to have the /bin/sh shell opened.

Observations: The first part of this task is to run [gdb stack] which opens gdb with the executable. Then set a breakpoint at [bof] within gdb and find the addresses of buffer and ebp which are shown in screenshot 2. These hex addresses are brought into [exploit.c]. The [exploit.c] is written to copy the current buffer + overwrite spaces with the malicious code to conduct the attack(screenshot 3). Lastly the new [exploit.c] is compiled and then the [stack.c] is run after. This is successful as shown in the last screenshot and the command id.





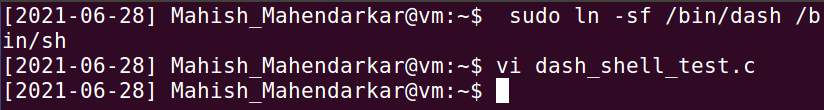


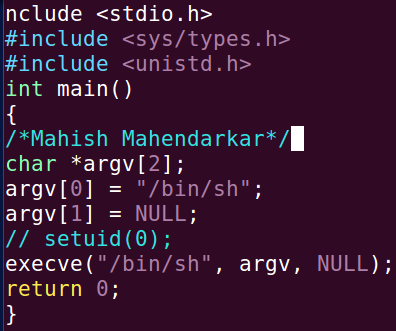


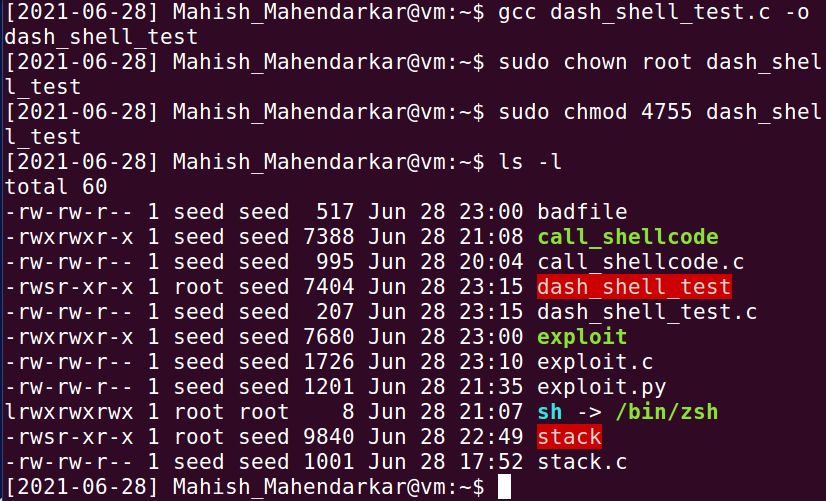
**Task 3 Defeating dash countermeasures**

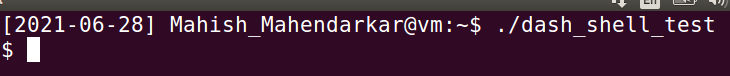
This task details how to bypass the countermeasures set by the dash shell. There is a command that can be run and a program which changes the system call setuid. This gives the highest privileges for the file.

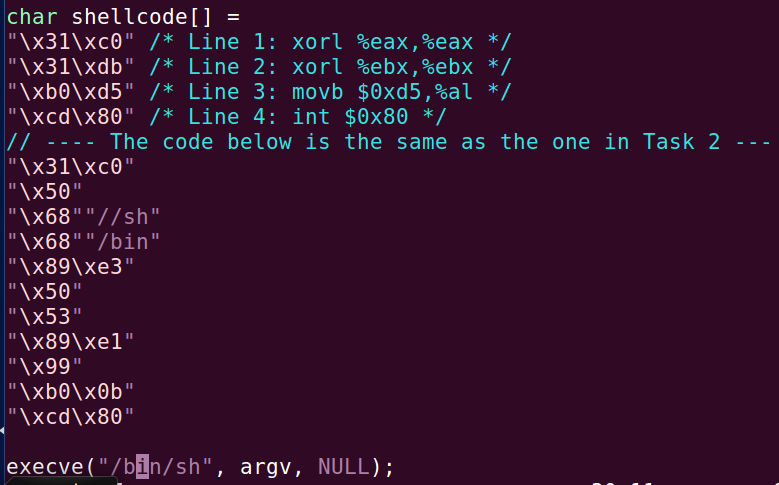
Observations: The first step is to create and modify the [dash\_shell\_test] file according to the instructions. This yielded no issues screenshot 4 shows that we are not in root. This is to be expected as the dash shell has prevention built in. The permissions of this file are shown clearly as well. The next step is to modify the [exploit.c] file and see if task 2 is replicable, it is! We are back in root.

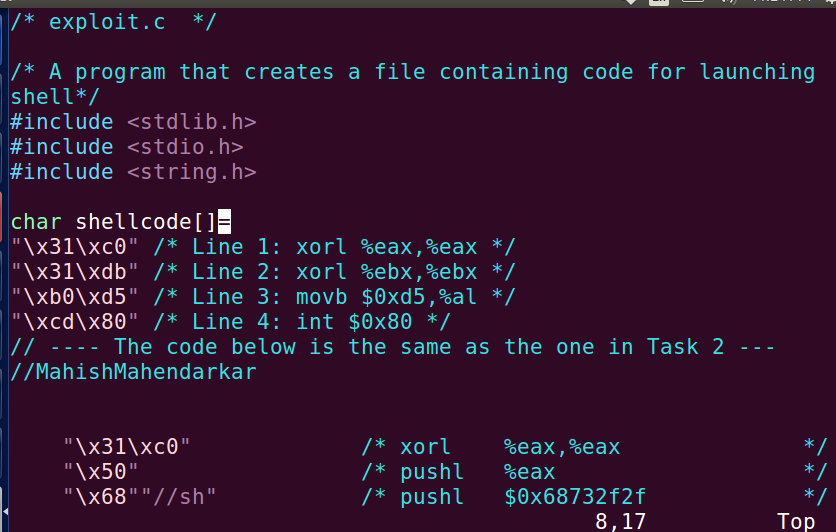


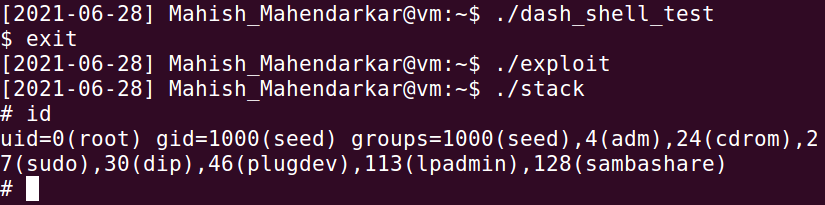








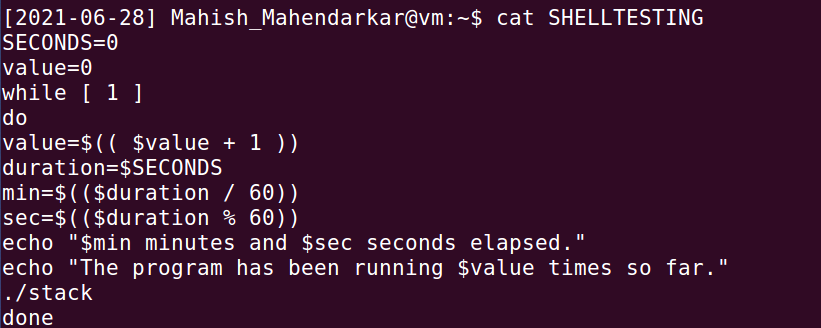


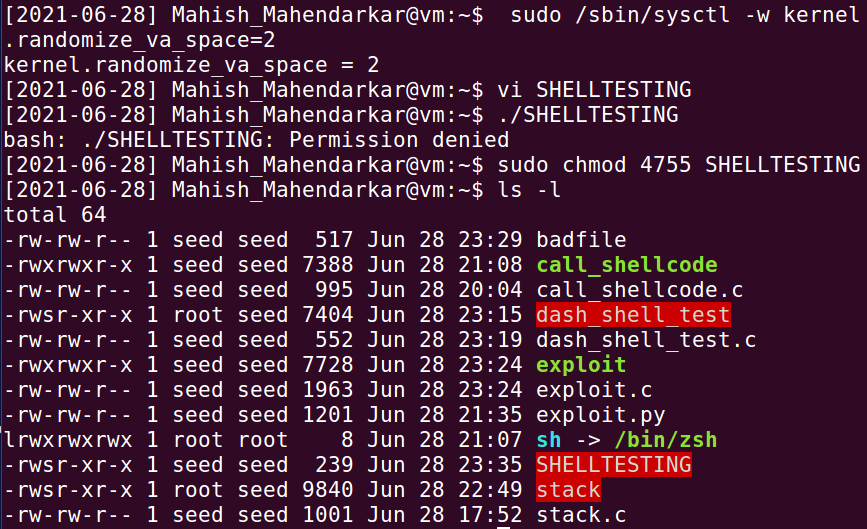


**Task 4 Defeating Address Randomization**

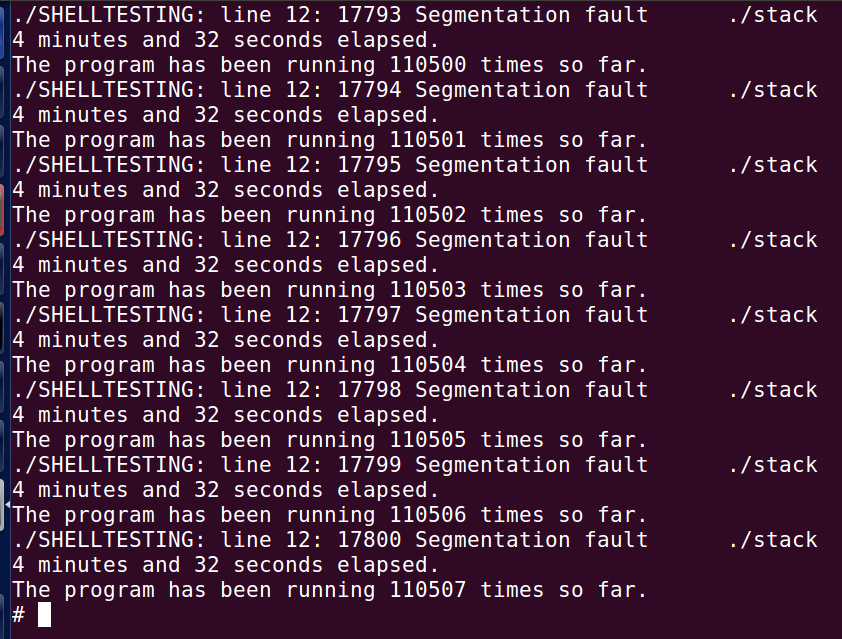
This task deals with the built in address randomization which limits the capabilities of a brute force attack to determine addresses.

Observations: The first step is to turn on the address randomization. Then the script is run to brute force the address guessing with [.stack]. The first screenshot shows the setup undertaken and the last screenshot shows the successful brute force into shell with 110507 attempts.





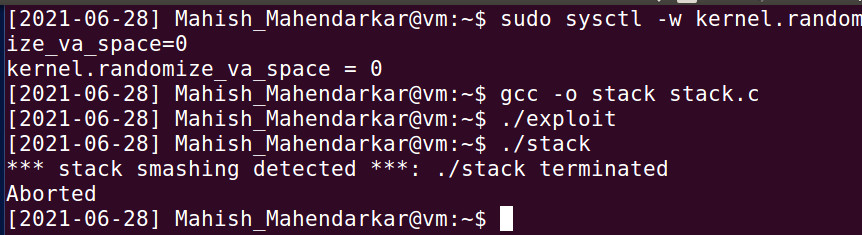




**Task 5: Turn on the StackGuard Protection**

This task repeats task 2 but with the stack guard protection enabled.

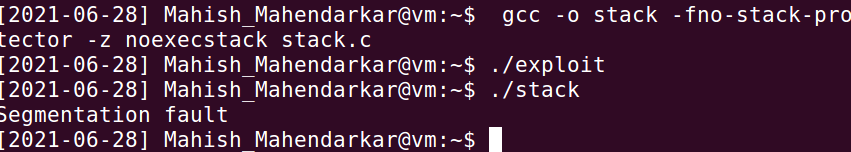
Observations: The first step is to turn off address randomization. The next step is to repeat task 2. This fails as expected.



**Task 6: Turn on the Non-executable Stack Protection**

This task repeats task 2 but with the non executable stack protection enabled.

Observations: The first step is to turn off address randomization. The next step is to repeat task 2. This fails as expected with a segmentation fault. The compilation with the overflow messes up the way the segments are ordered because the stack is preventing execution.



1. **What happens when you compile without “-z execstack”?**

**There are stack smashing errors that abort the running.**

1. **What happens if you enable ASLR? Does the return address change?**

**Yes, this causes the vulnerability to be nullified.**

**c. Does the address of the buffer[] in memory change when you run stack**

**using GDB, /home/root /stack (stack.c location), and ./stack?**

**No after compilation it has remained the same address.**