**Name:** Henry Salgado

**ID**: 80509684

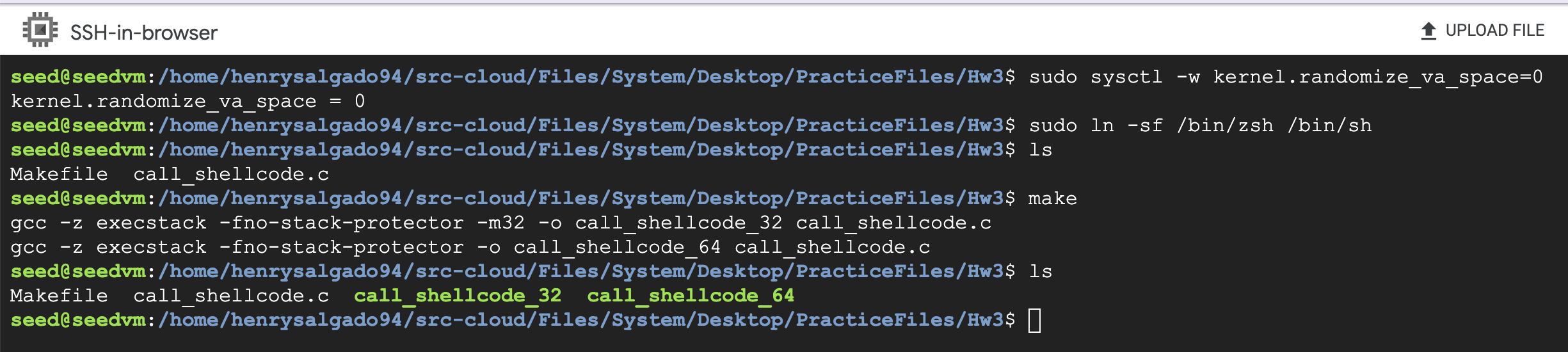
**CS 4351/5352: Computer Security**

**Assignment 3**

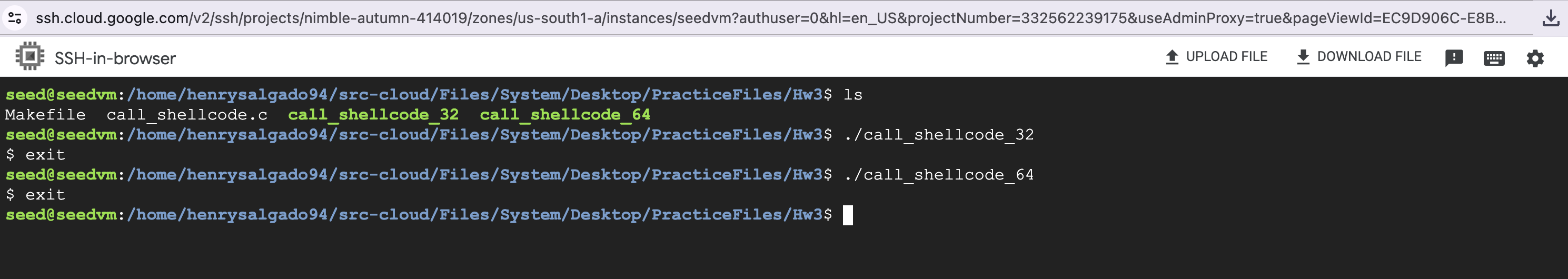
**Question 1:**

***Task 1: Invoking the shellcode***

First, I turn off the address space randomization and configure the bin. I also add the call\_shellcode.c and Makefile to my folder titled “Hw3”.

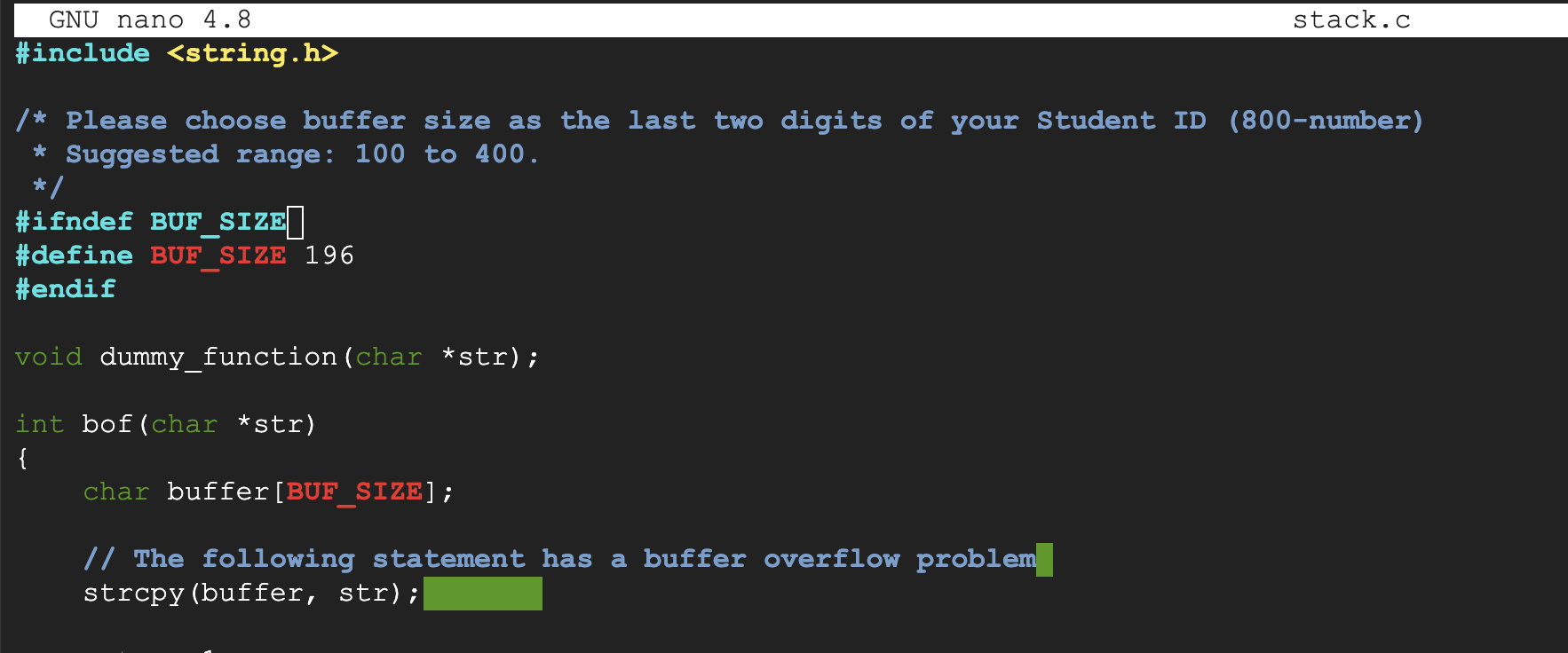


Now, I will compile with make file. After running the executables, both versions successfully spawn a shell, indicated by “ $ ”.

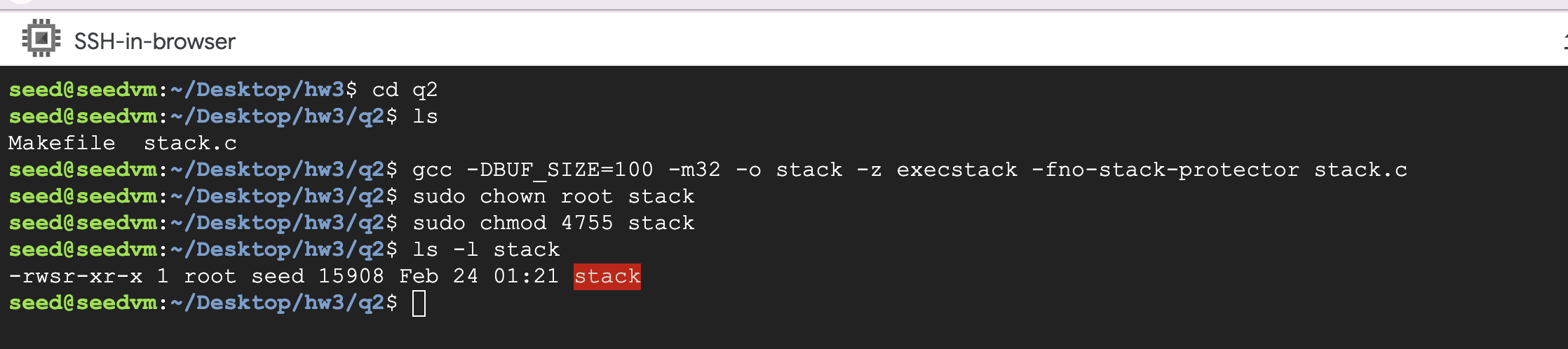


***Task 2: Understanding the Vulnerable Program and Compiling***

Changed BUF\_SIZE = 196, since my ID ends in 96

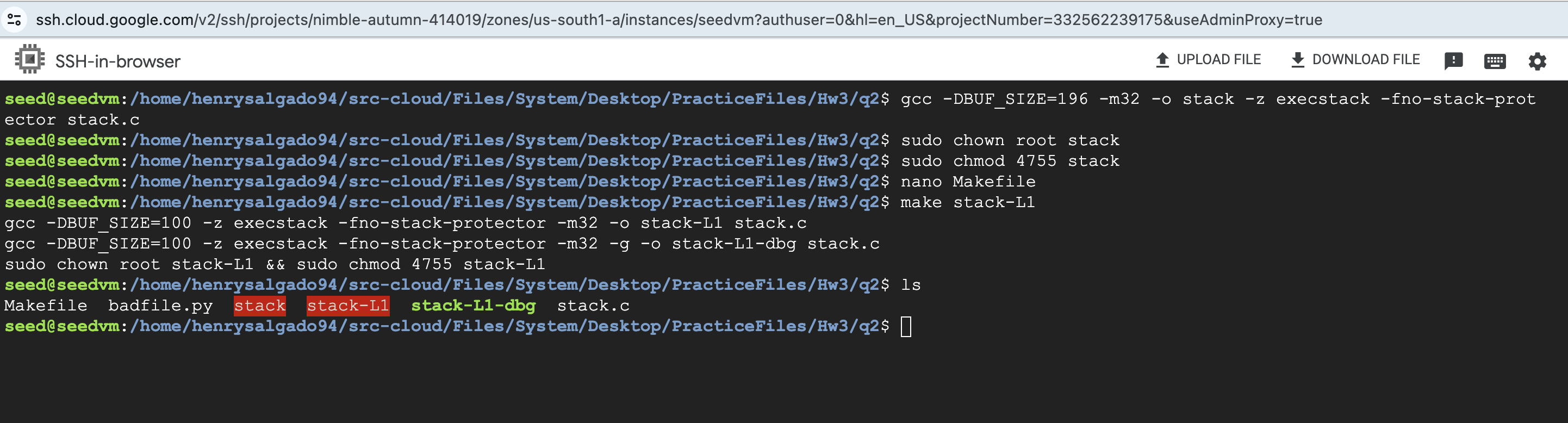


As mentioned in the assignment, I will first change ownership of the program to root and then change permission to 4755 to enable the Set-UID but.

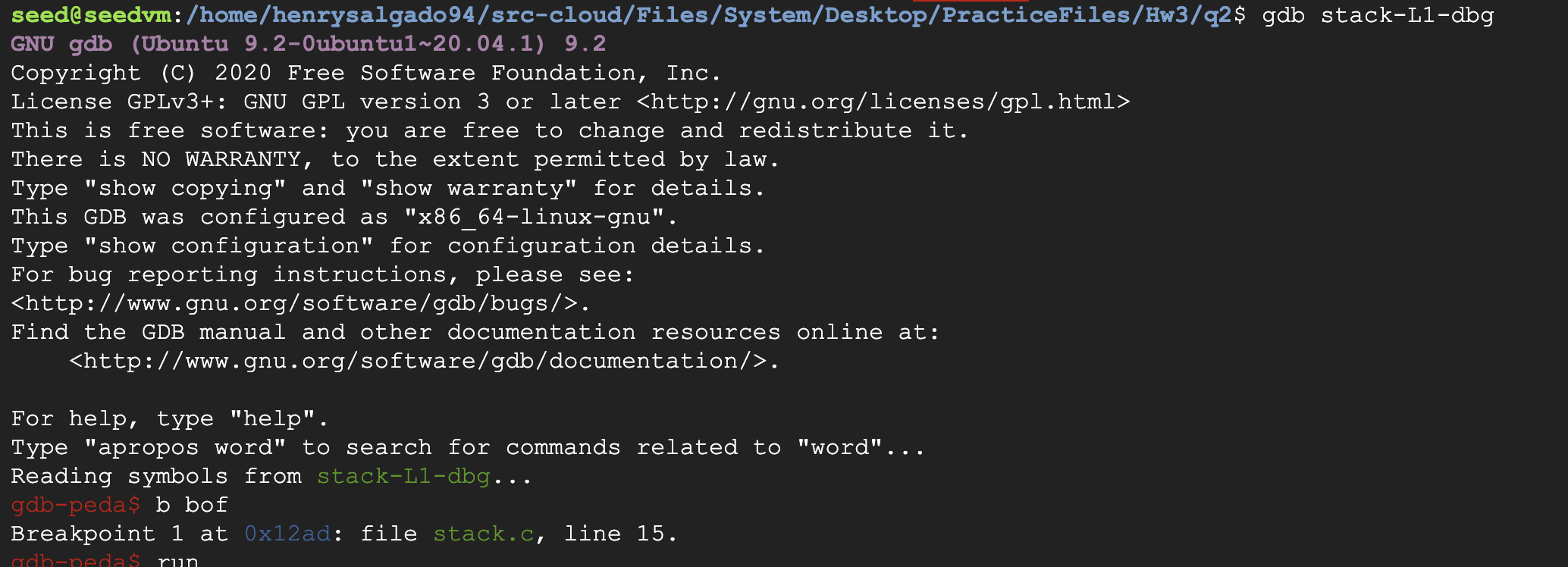


***Task 3: Launching Attack on 32-bit Program***

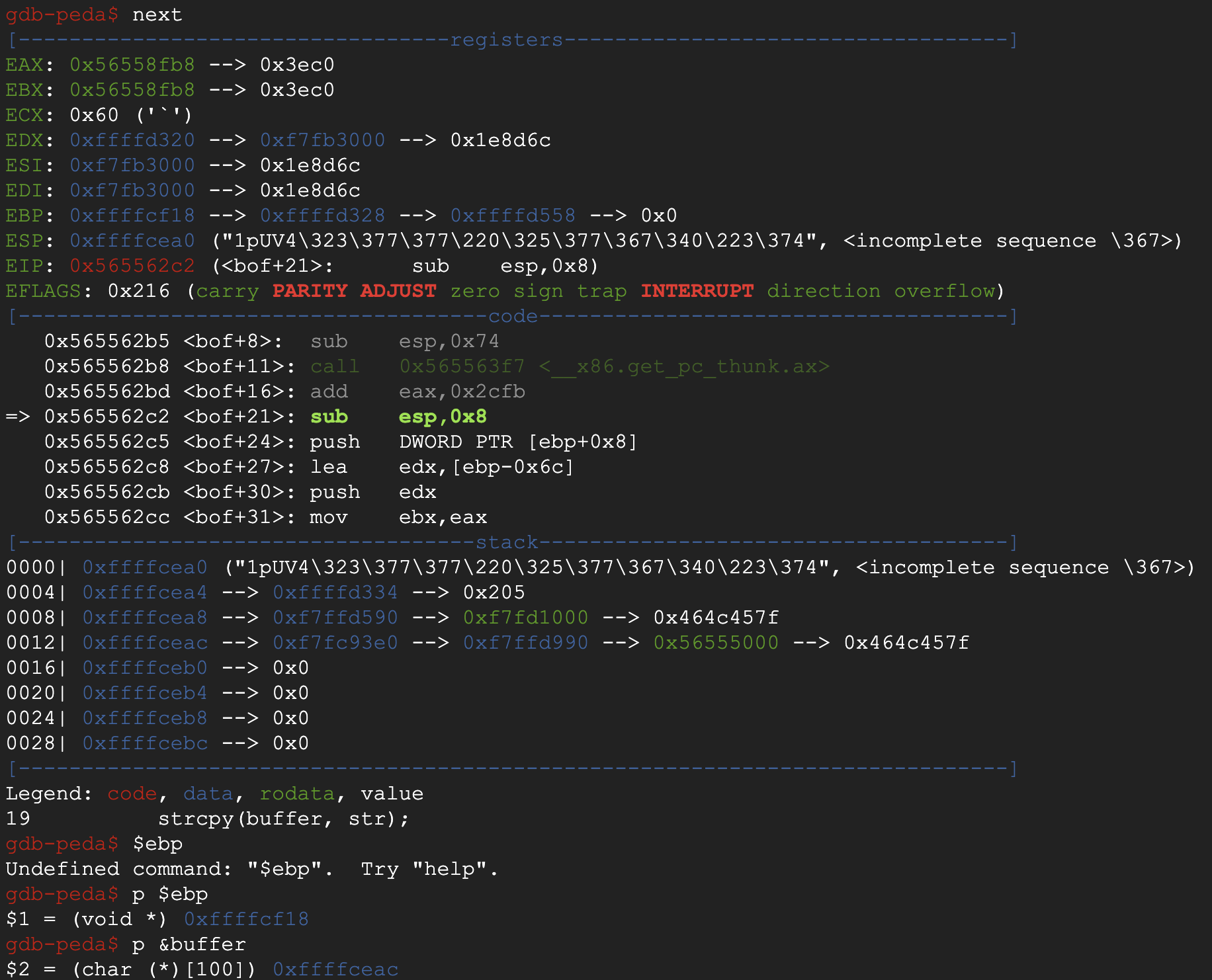
Now, I will compile my make file



I will run the debugger, set the break point at function bof() and run



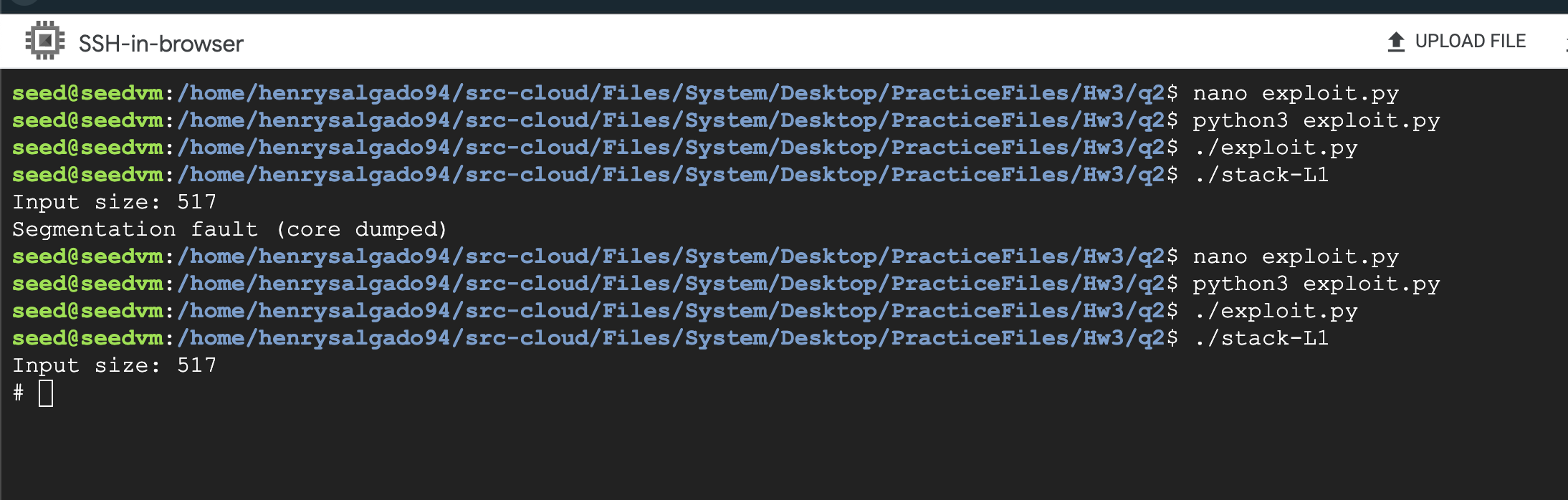
*Type next and then get the ebp value and address of the buffer*



*We can see that :*

|  |
| --- |
| *gdb-peda$ p $ebp*  *$2 = (char (\*)[100]) 0xfffcf18* |
| *gdb-peda$ p &buffer*  *$2 = (char (\*)[100]) 0xffffceac* |
| *Difference: 0xfffcf18 – 0xffffceac = 108* |

*Testing the program with different return address values. As seen, the first attack dumps the core. The last attempt was successful as observed by the root*



*These were the values I used for my exploit.py.*



**Details below:**

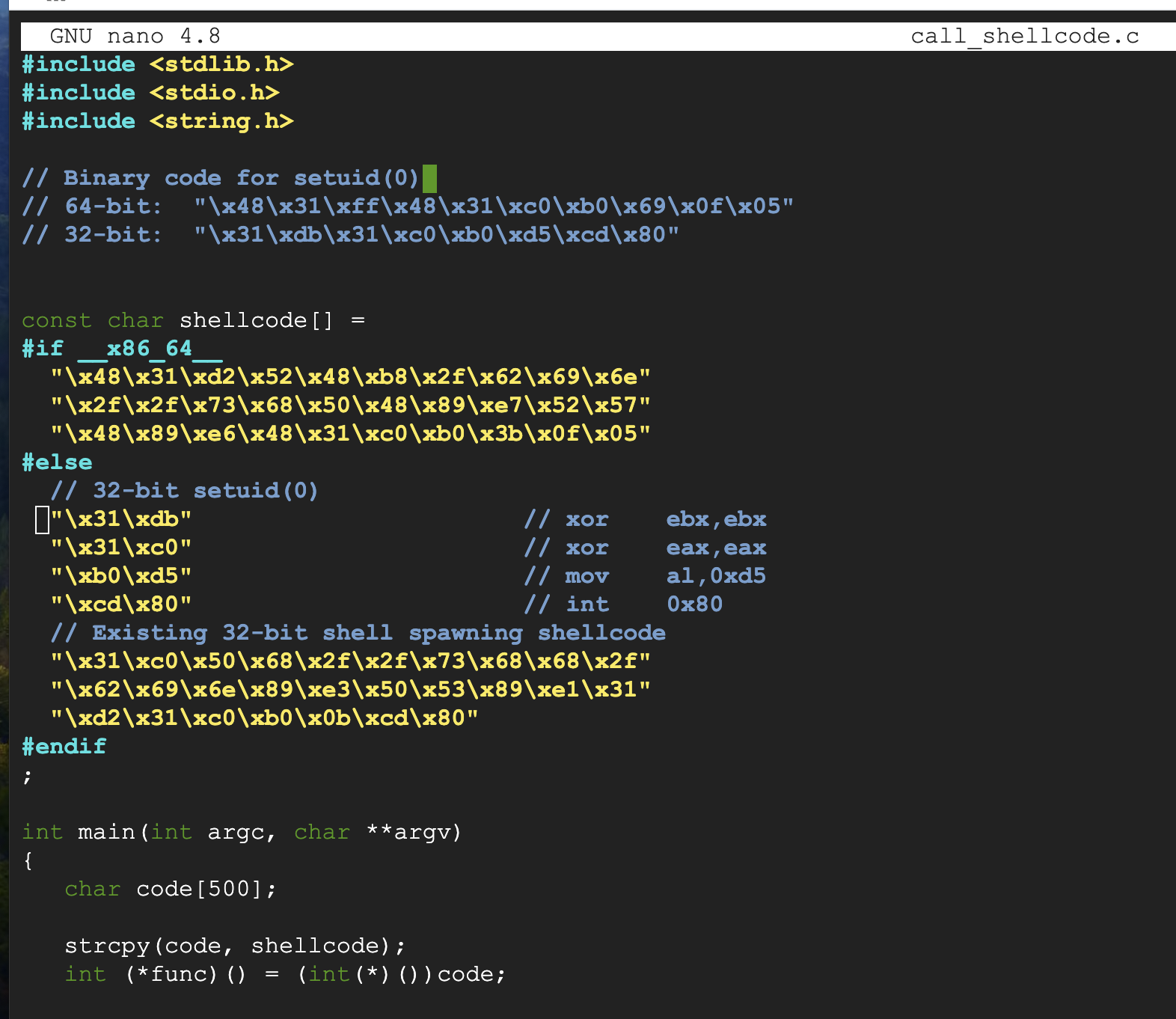
***Selection of Shellcode:*** The choice of 32-bit shellcode was dictated by the target program's architecture.

***Payload Placement Strategy:*** By setting the start value to 517 - len(shellcode), I ensured that the shellcode is positioned at the very end of the buffer, immediately before the return address. This arrangement is crucial for the exploit's success, as it allows the overwritten return address to directly point to the shellcode's location within the buffer

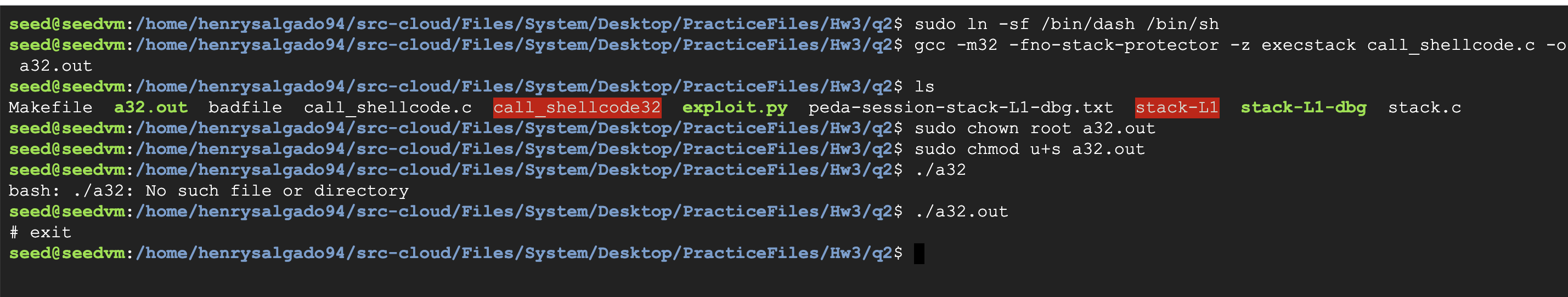
***Determining the Correct Return Address:*** By incrementally adjusting the return address beyond the base frame pointer (EBP) value and fine-tuning the offset—representing the distance between the buffer start and the return address location. This approach was necessary due to the observed discrepancies when running the target program outside the debugger (gdb), which alters the stack layout. The lab's notes clarified that without gdb, additional data pushed onto the stack would shift the frame pointer, necessitating adjustments to both the offset and the calculated return address to accurately target the shellcode's location.

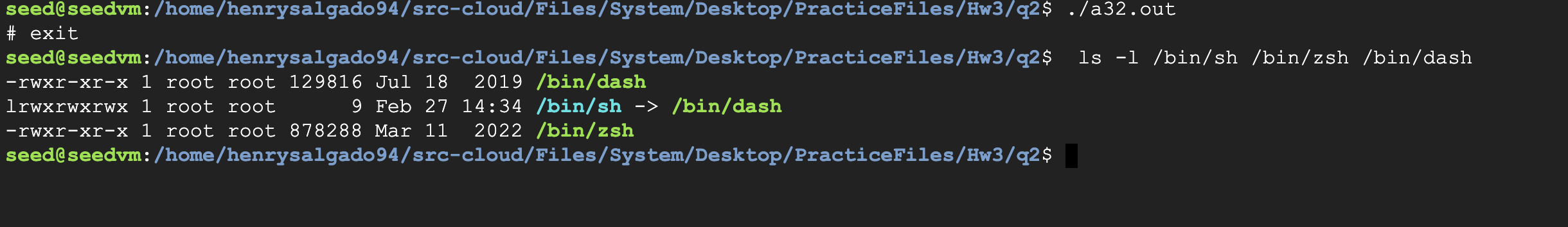
***Task 4: Defeating dash’s Countermeasure***

First, I will add the binary code for setuid(0) at the beginning of the file. Please note that I only updated the setuid for the 32 bit, I will test for 64 bit later.



First, I will compile call\_shellcode.c, make it a root owned binary with setuid and run the shell code

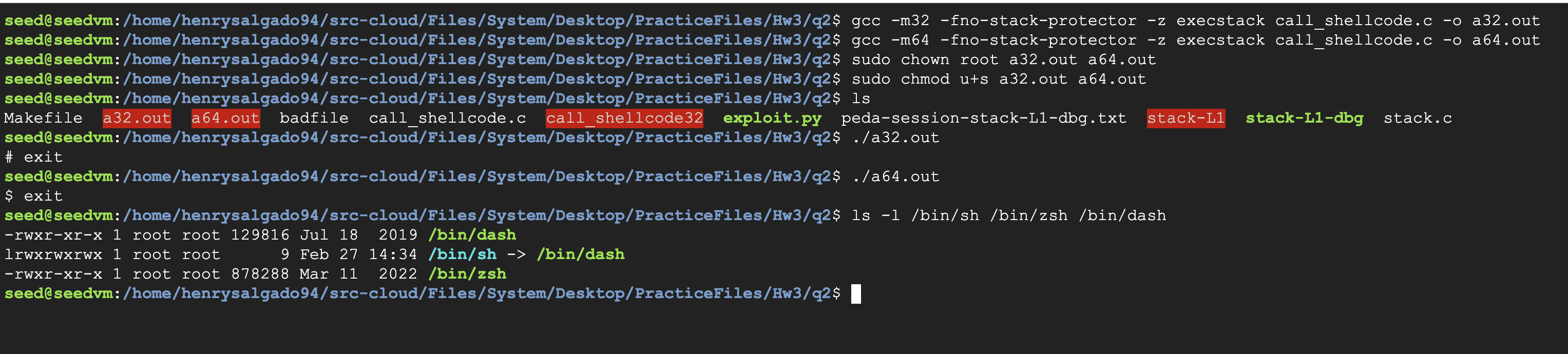




**Observations:**

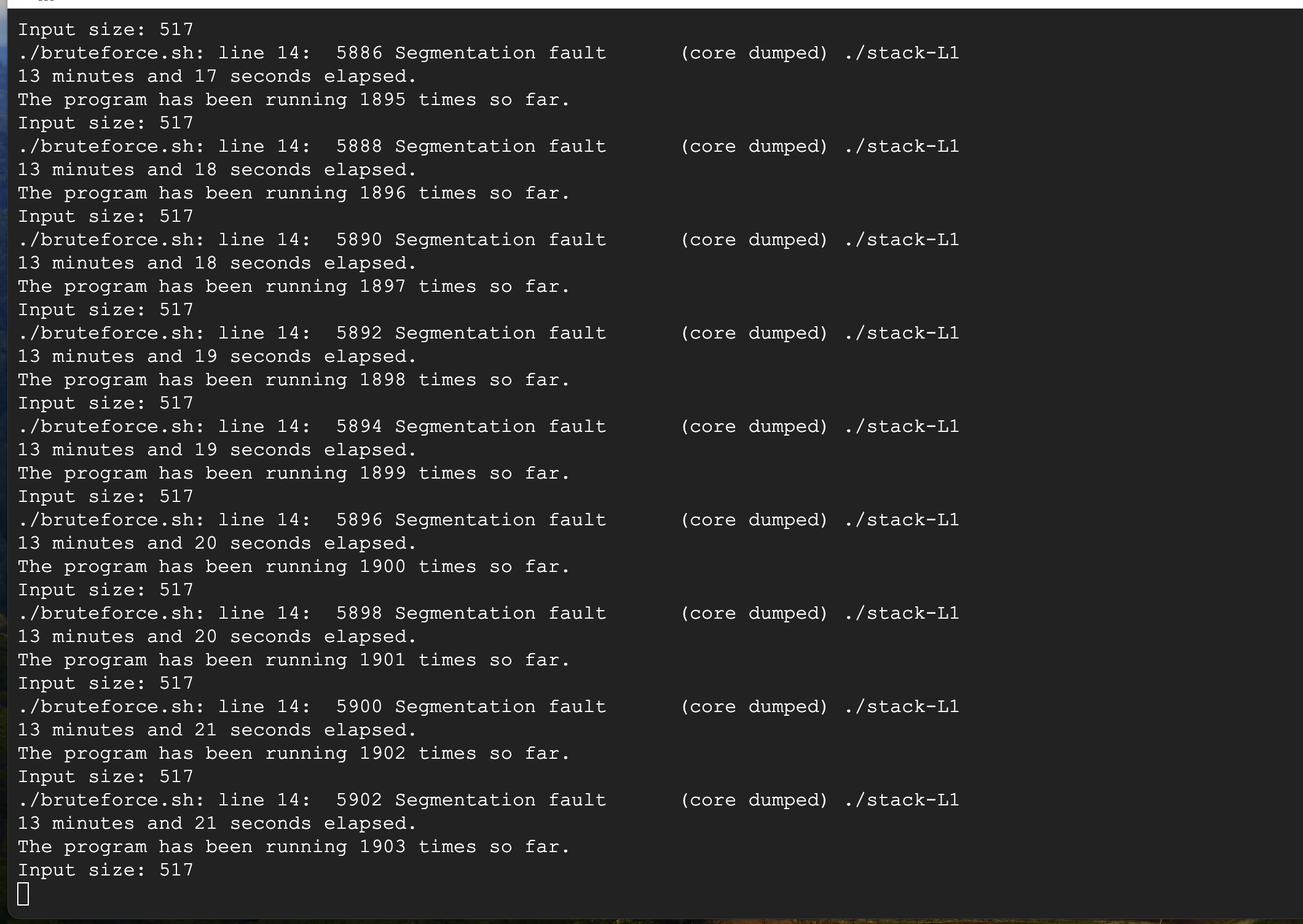
* The process indicates that with the setuid(0) system call included in the shellcode and the binary being owned by root with the setuid bit set. Executing the binary indeed provided elevated privileges, as intended by the experiment.
* Received a root shell “#” upon execution of a32.out confirms the shellcode's effectiveness in escalating privileges.

I now test what happens when I try executing the 64 bit without any setuid(0) code being added in the shellcode.c. Surprisingly, the .a64.out spawns a shell “$”. However, notice that the 32 bit spawns a root own shell “#” and 64 only a non-root “s” shell.

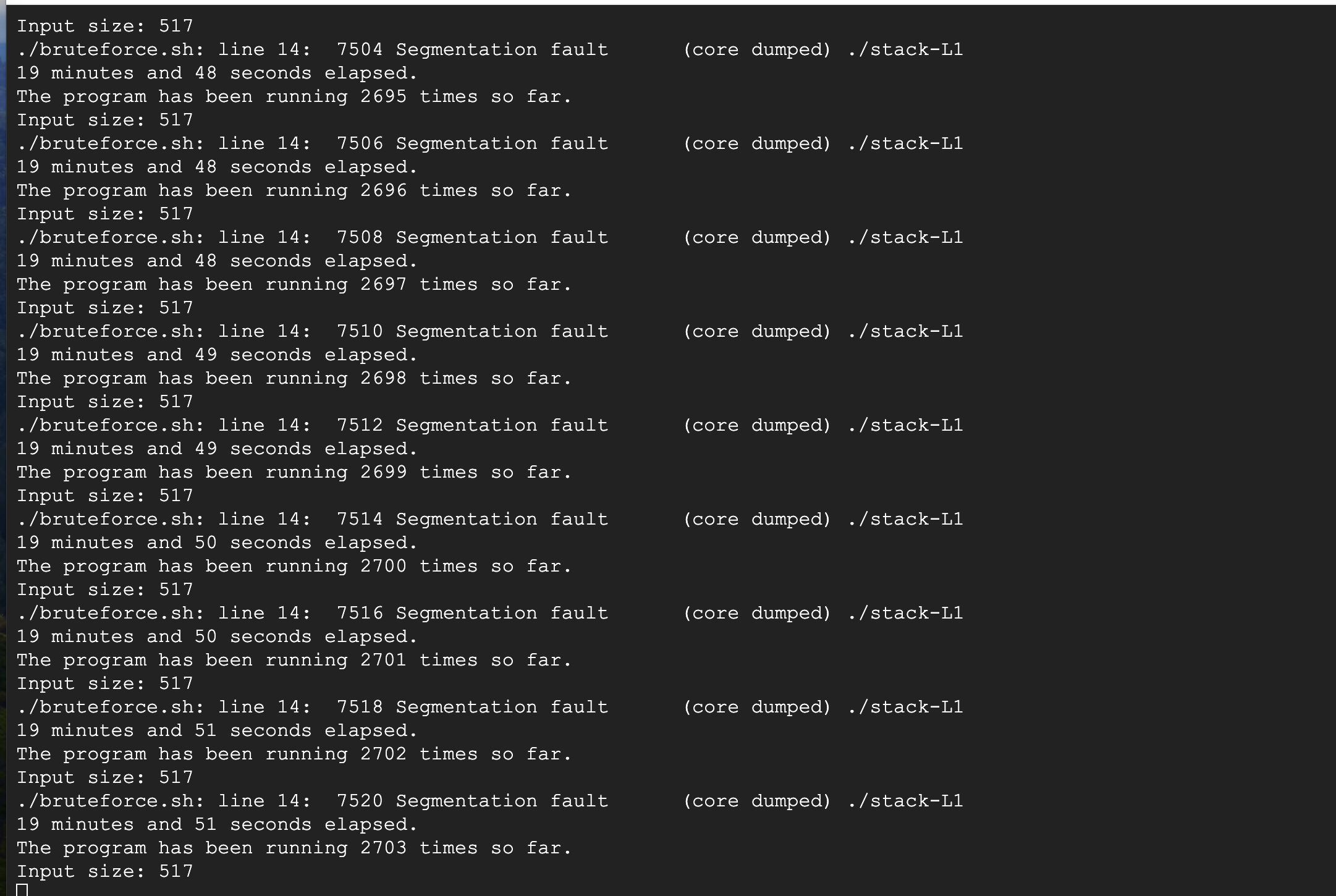


**Task 5: Defeating Address Randomization**

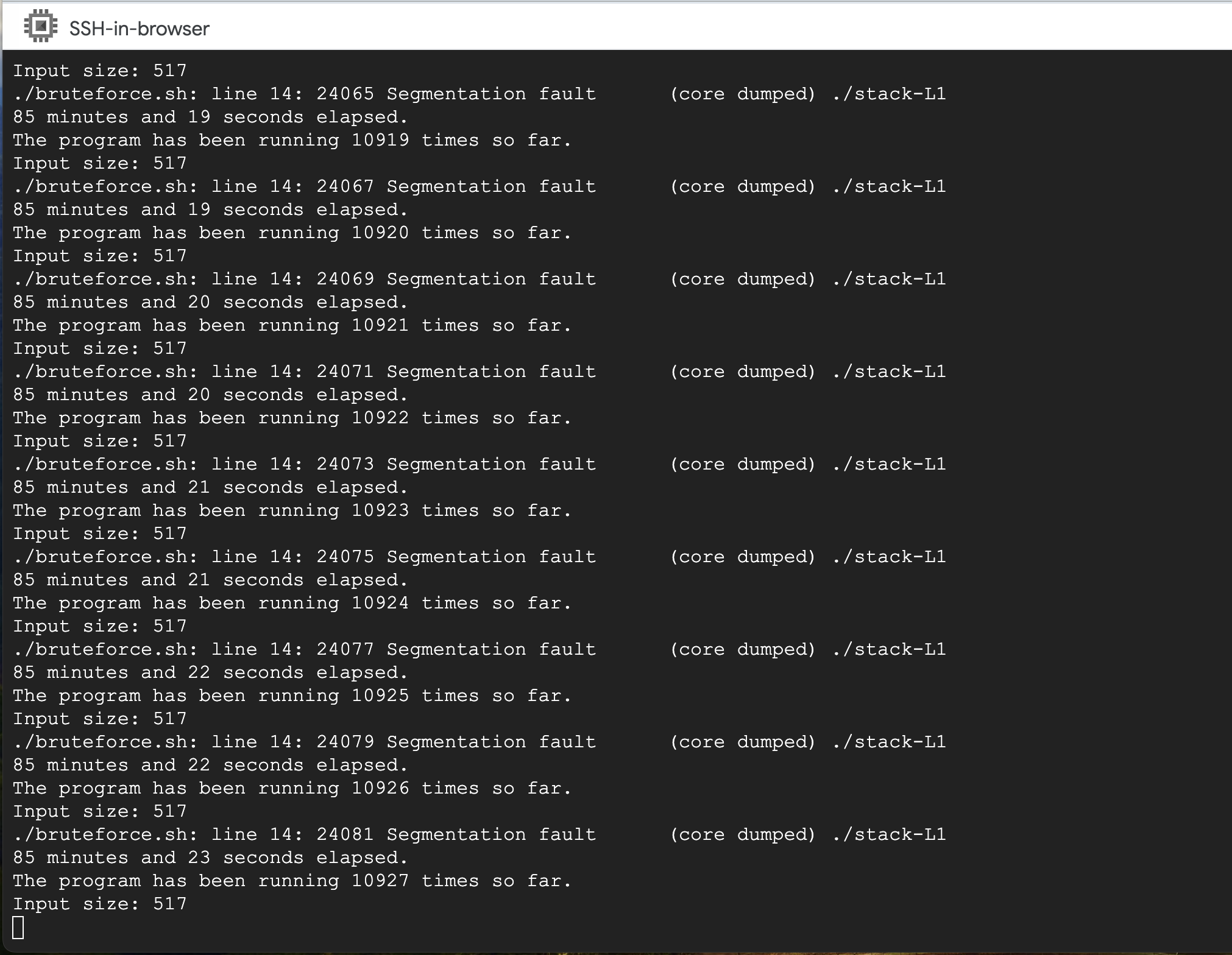
Added the shell script using nano editor, and ran the script



After 19 minutes, no success



After 86 minutes, no success



**Observations:**

* It seems that I am one of the unlucky ones. Despite running my shell program 10927 times, and for more than 85 minutes, it was unsuccessful. Since this is a brute force attack, it might is expected to be way less efficient as it is trying one by one.