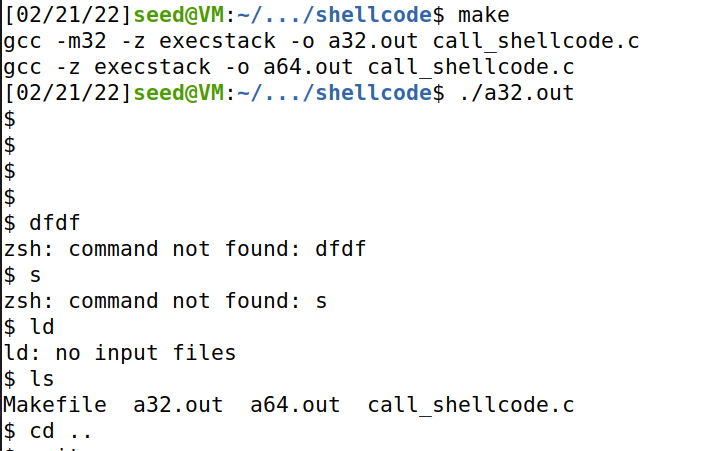
Val Robichaux

Homework 3

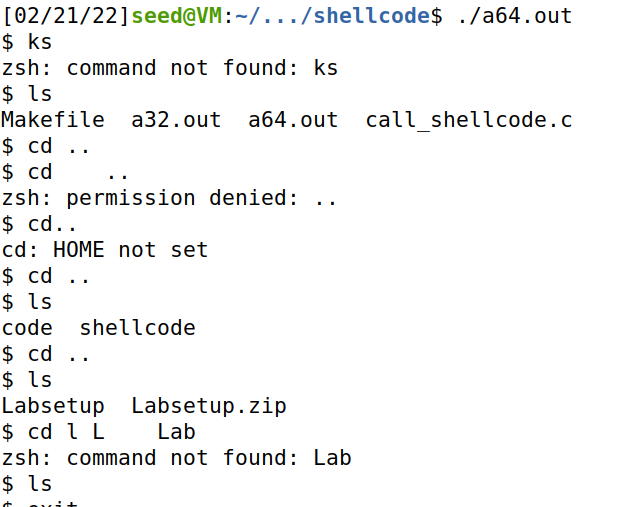
CSCE 465

**Task 1:**

**32-bit**

****

**64-bit**

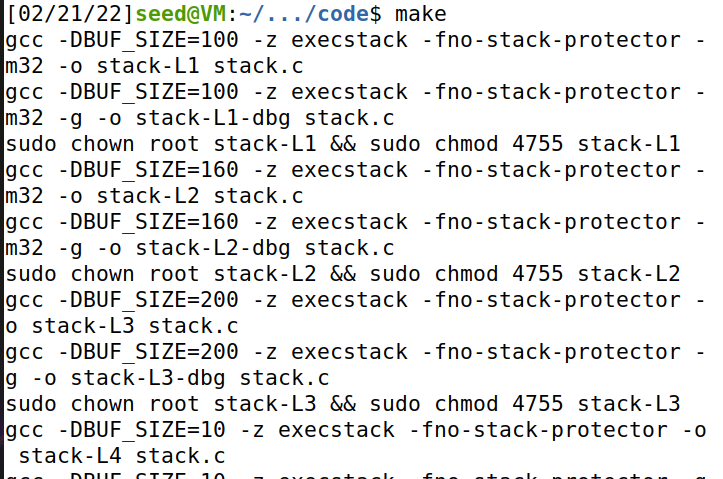
****

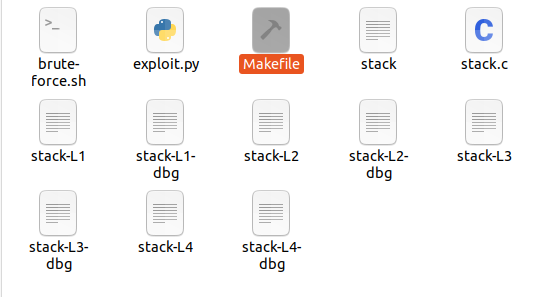
**Observation**

* The paths are not listed out on the command line
* Most general linux commands work
* The delete key is registered as a tab?
* A couple of specific inputs produce unexpected results on the command line

**Understanding the Vulnerable program**

**Buf\_sizes**

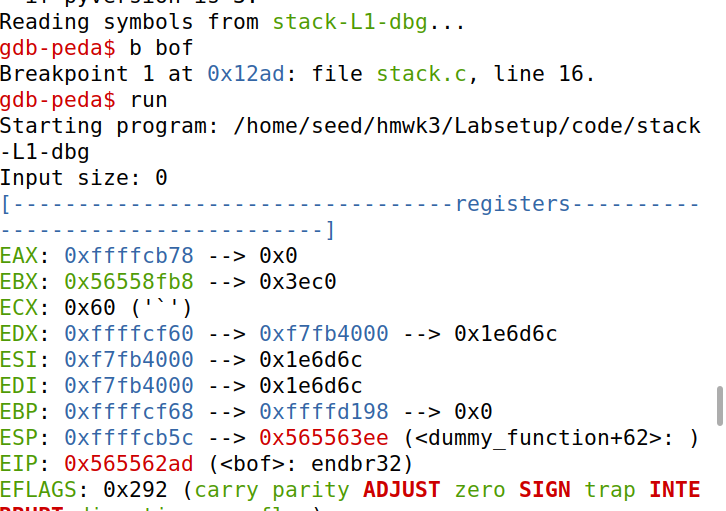
****

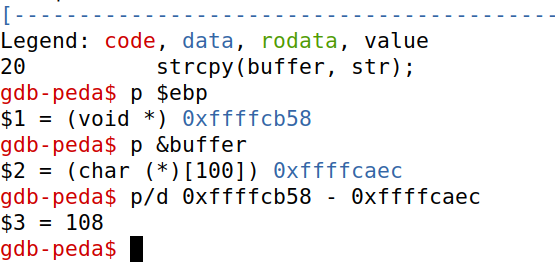
****

* Followed the directions as outlined in the deliverable report and the lab report given

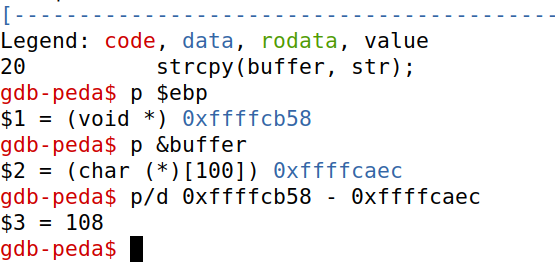
**Task 3:**

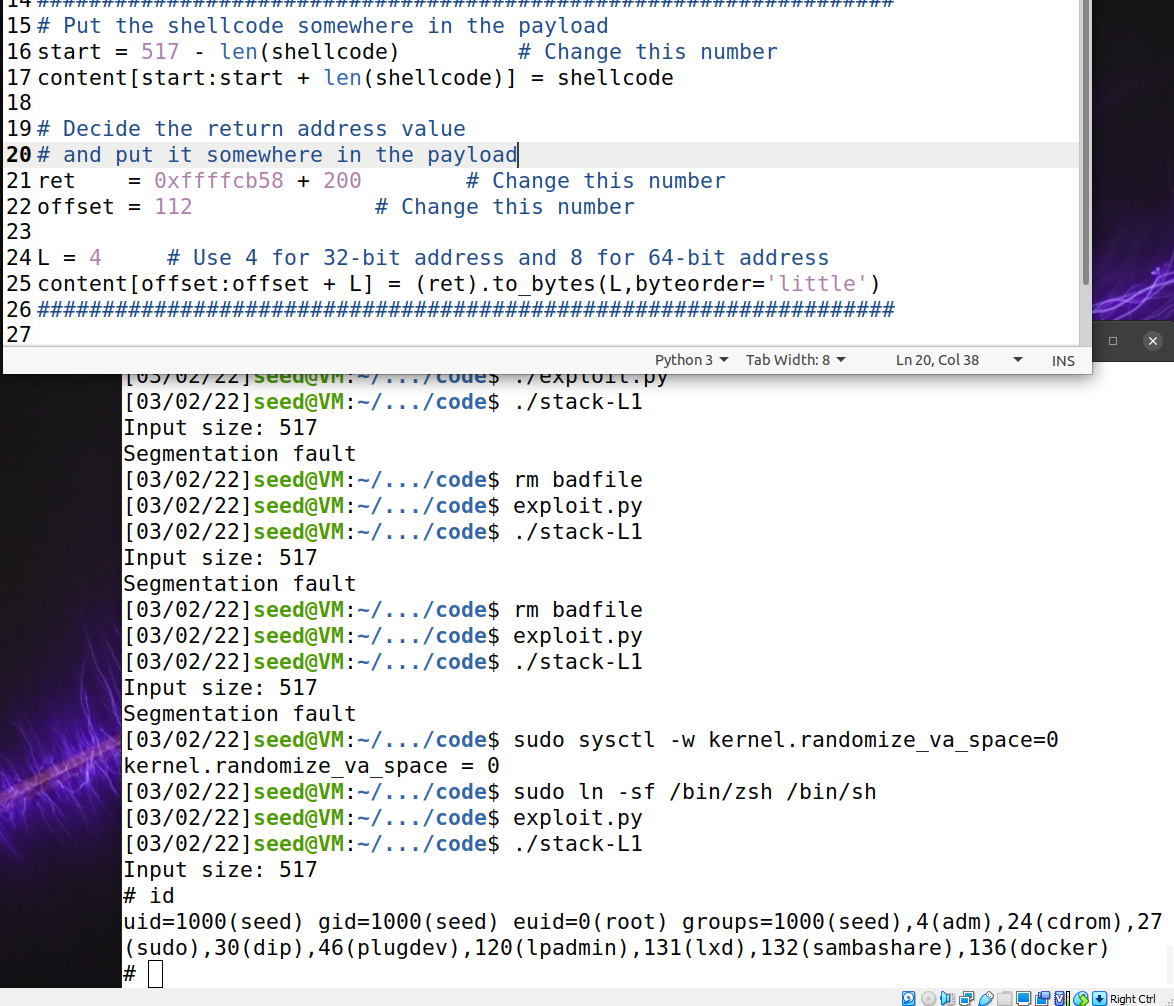
**Investigation**





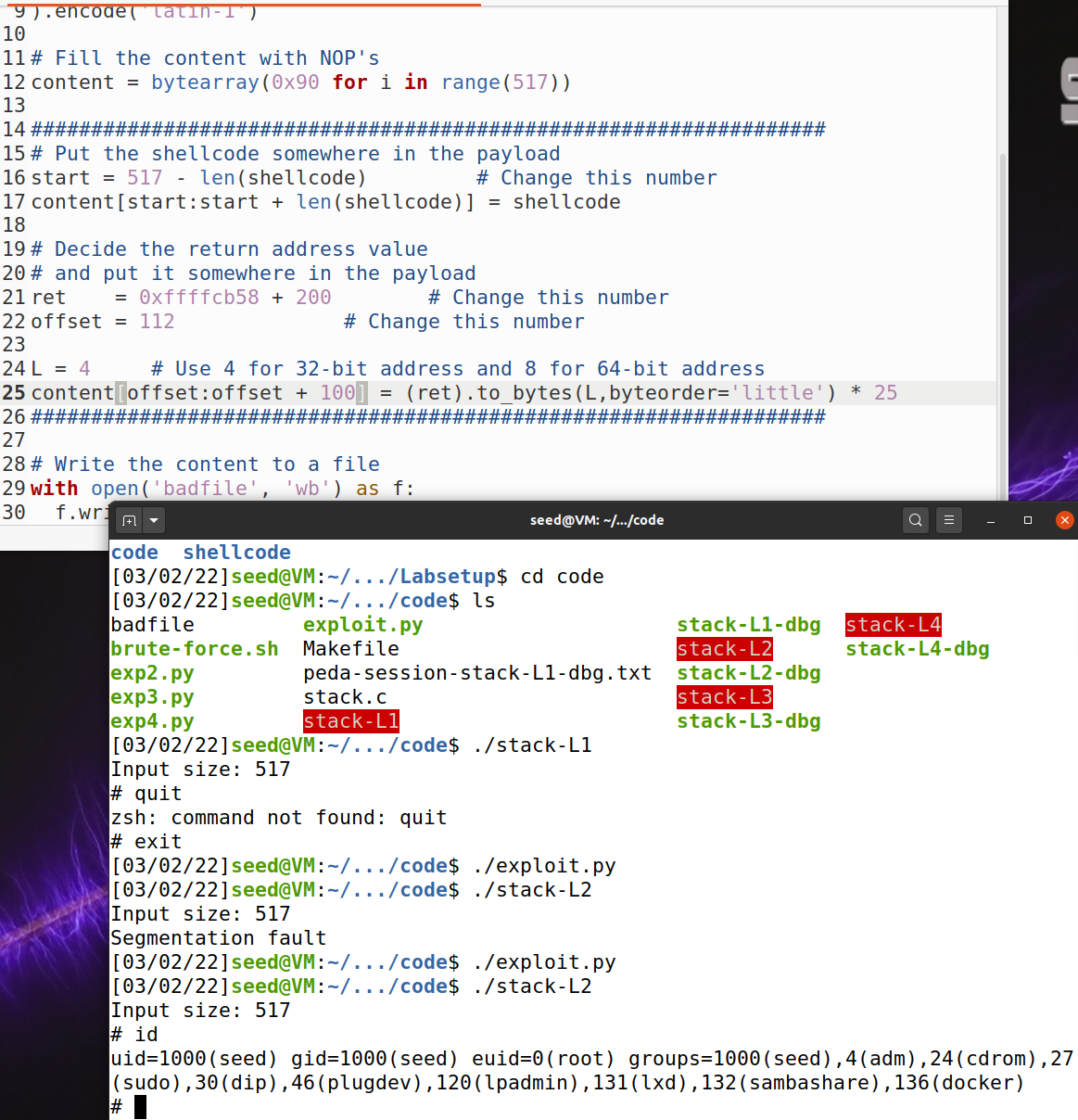
**Level 1**





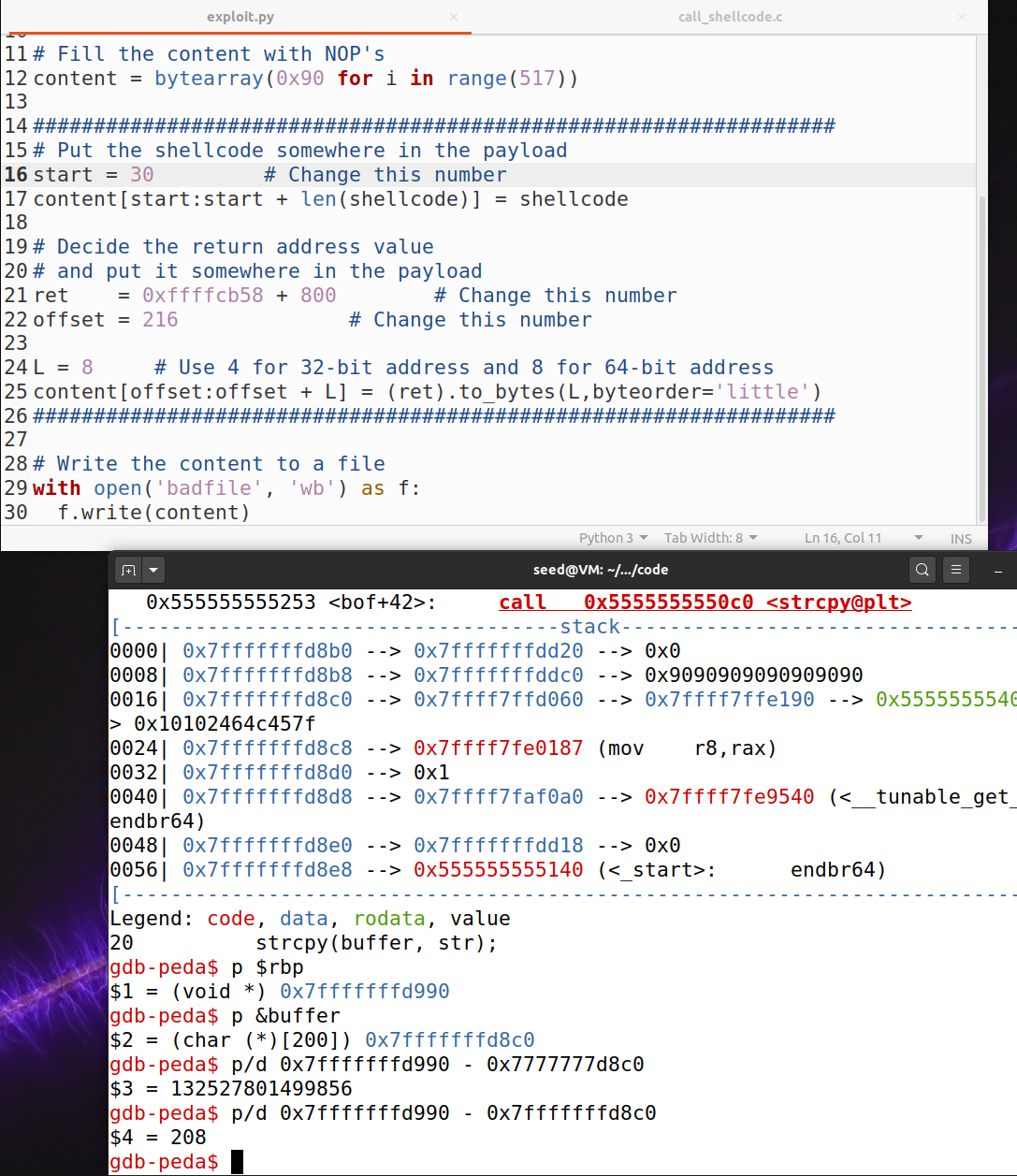
* We find the current stack from when gdb stops inside the bof
* We use next to execute the next few instructions and stop after the ebp registered is modified so that we have the stack frame of the bof() function.
* We can now see the ebp value and the buffer’s address which will both be very useful for us when we start launching our attacks
  + **Start value is set at 517 - len(shellcode)** so that we can place our payload at the end of our stack
  + **Ret** address is the frame pointer plus an arbitrary amount to compensate for the extra data that is pushed during gdb
  + **Offset** is simply the distance between the frame pointer and the buffer + 4 to account for the character size

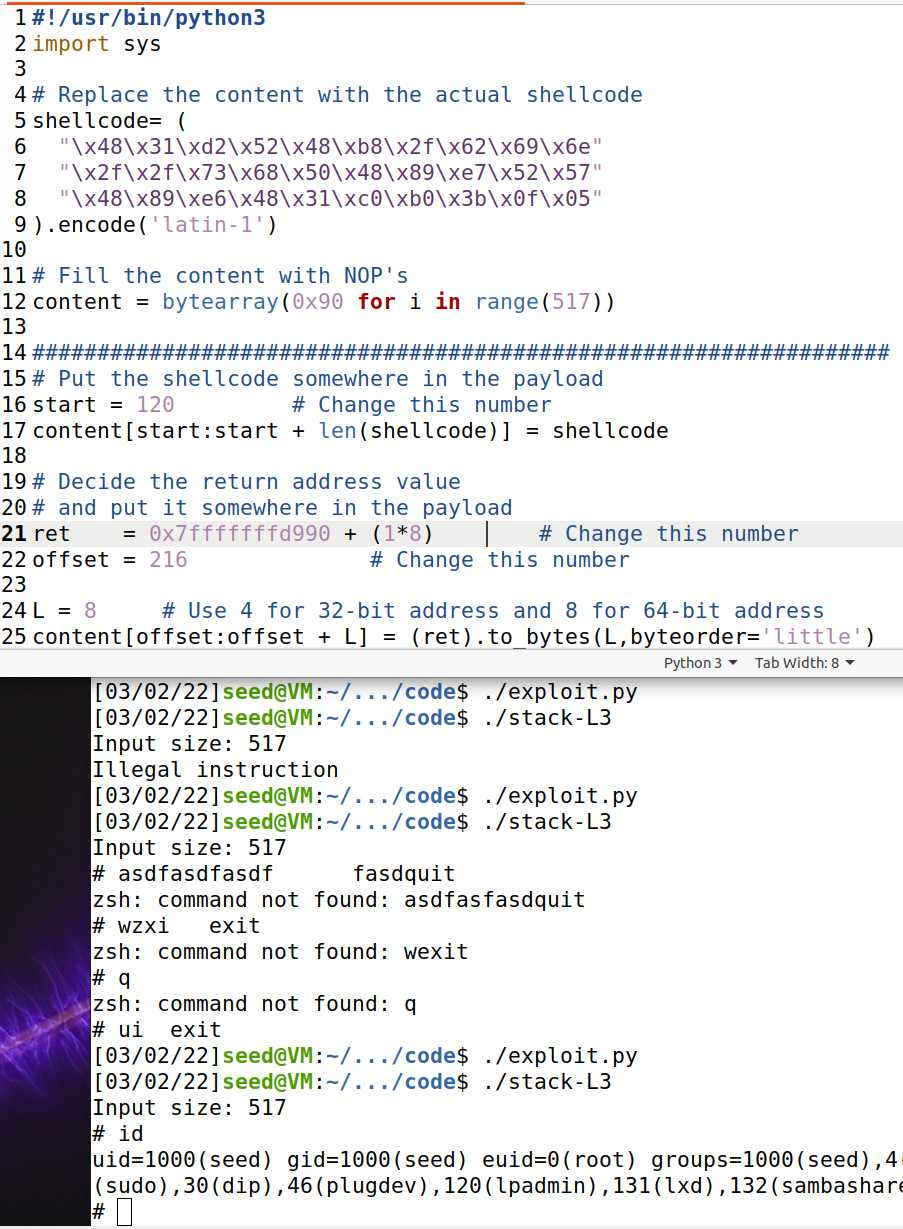
**Level 2**



* We can now see the ebp value and the buffer’s address which will both be very useful for us when we start launching our attacks
  + **Start value is set at 517 - len(shellcode)** so that we can place our payload at the end of our stack
  + **Ret** address is the frame pointer plus an arbitrary amount to compensate for the extra data that is pushed during gdb
  + **Offset** is simply the distance between the frame pointer and the buffer + 4 to account for the character size
  + The **content** line on line 25 is modified because we know that the buffer size is between 100 and 200 so I simply add the lower end of the bound to achieve the expected results and place the content in the correct part of the program

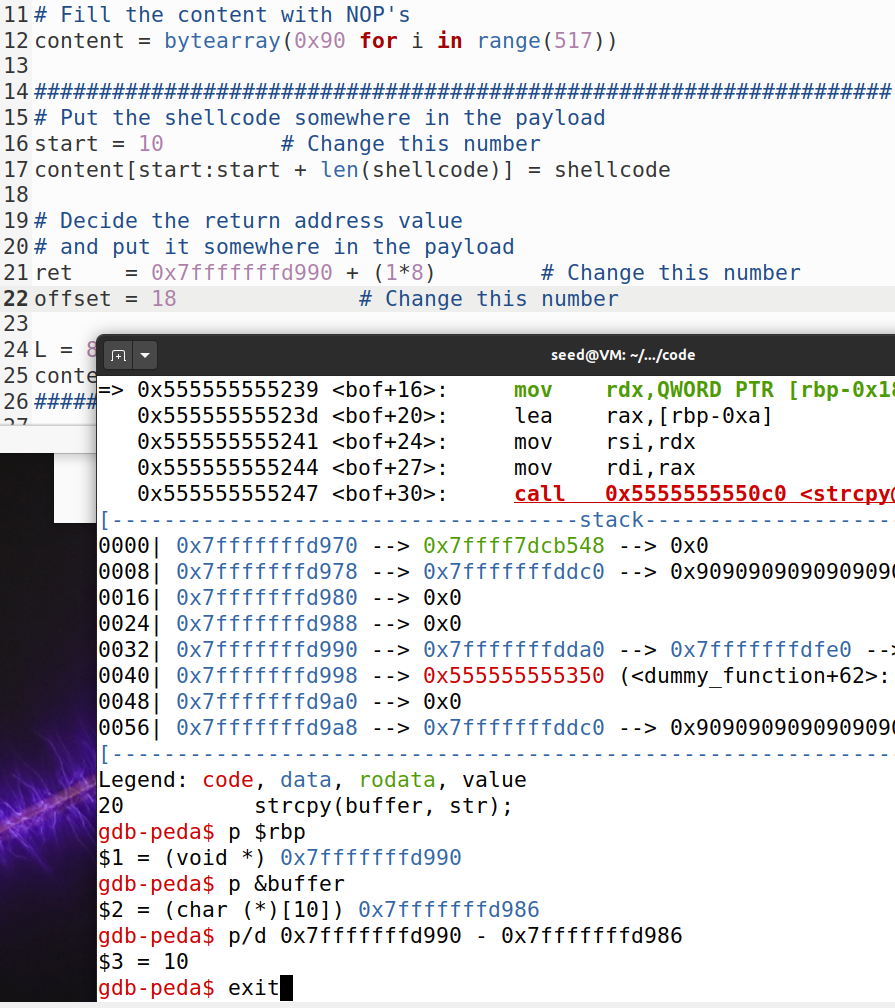
**Level 3**

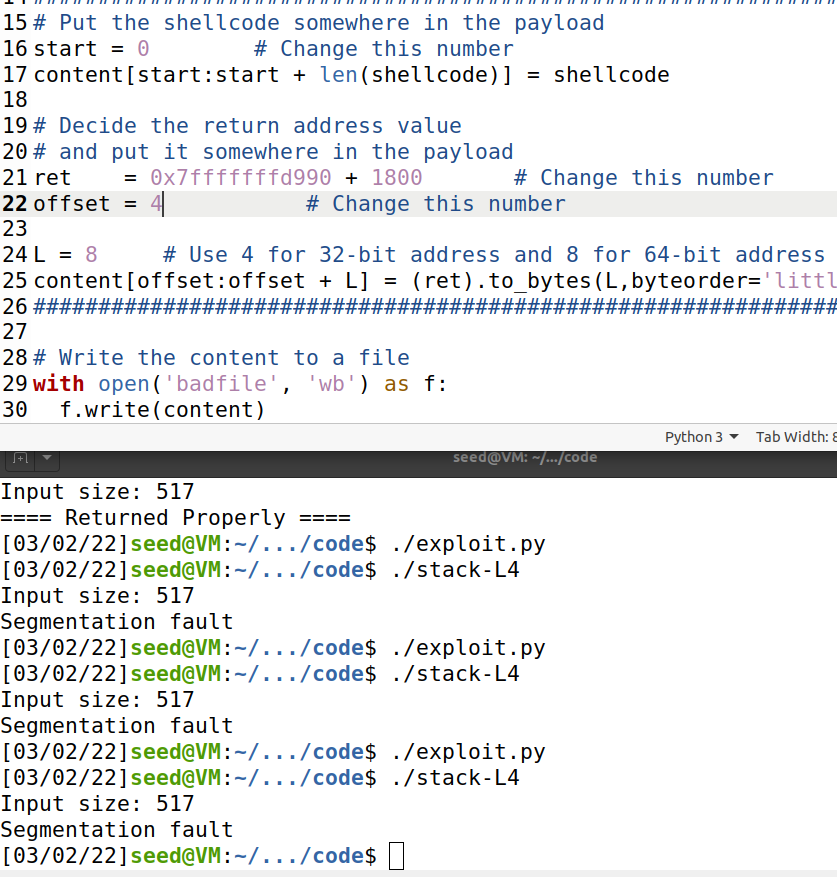




* We can now see the rbp value and the buffer’s address which will both be very useful for us when we start launching our attacks
  + **Start value is set at 120** because the buffer size is known to be 160 as shown in the makefile so anything between 0 and 130 should work for us.
  + **Ret** address is the frame pointer plus an arbitrary amount that is divisible by 8 to compensate for the extra data that is pushed during gdb
    - Side note: the frame pointer in this case is rbp in x64 architecture.
    - 8 is used because now the word size is larger for a x64 system.
  + **Offset** is simply the distance between the frame pointer and the buffer + 4 to account for the character size so in my case it is **216**

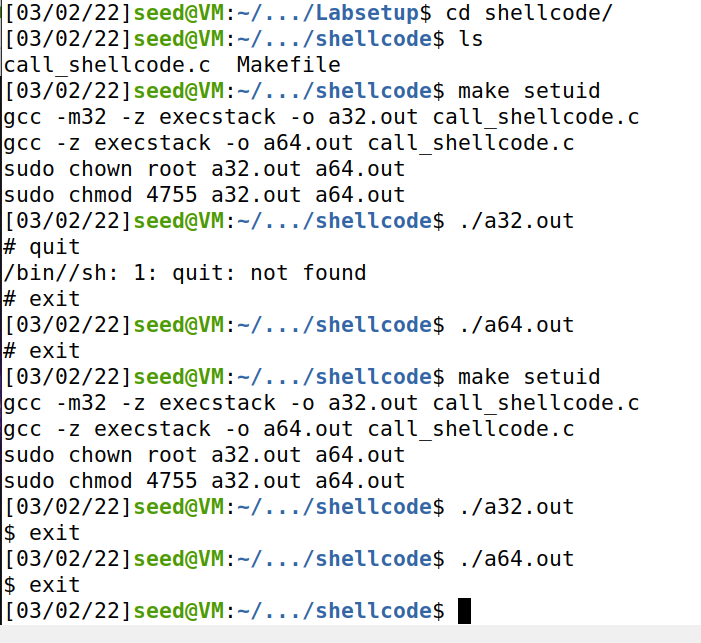
**Level 4**

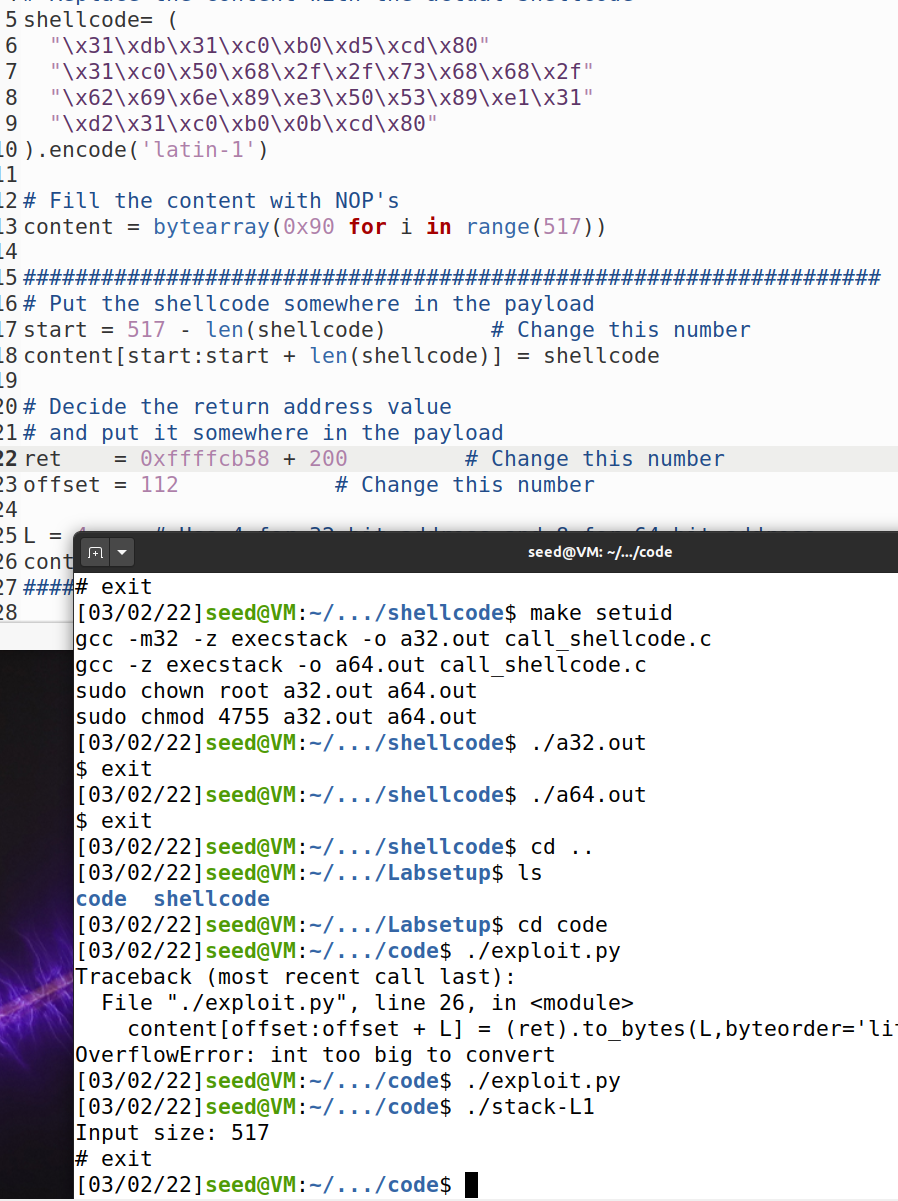


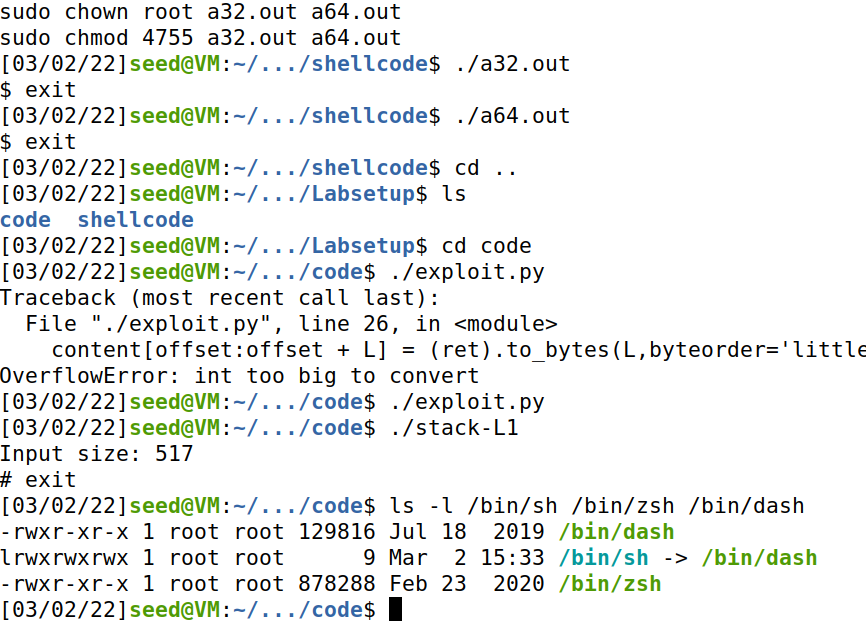


* **Start value is set at 0** because the buffer is so small it’s arbitrary where to start, so I am starting it at zero
* **Ret** address should be the rbp value + 1000 or more because there is a dummy function that has the size of 1000
* **Offset** is the number that really gave me a hard time because I found that every time I increased the number past 4 it would immediately cause my system to seg fault and I could not figure out why this is the case.

**Defeating Dash’s countermeasures**

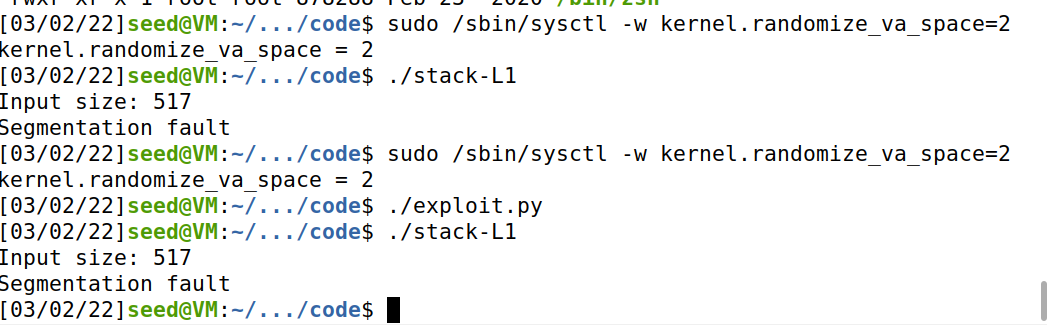
****

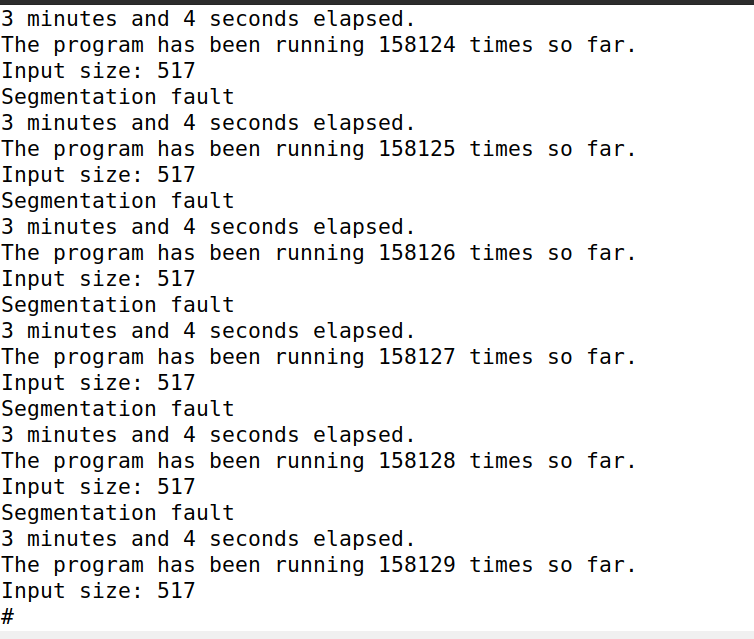
****

****

* Observations for adding UID to the shellcode and removing the UID from the shellcode
* As you can see with the UID set it gives us the **root shell**
* With the UID not set it gives us the normal **user shell**
* **Yes** the attack still works because the setuid() is still called to get root privileges.

**Task 8**

****

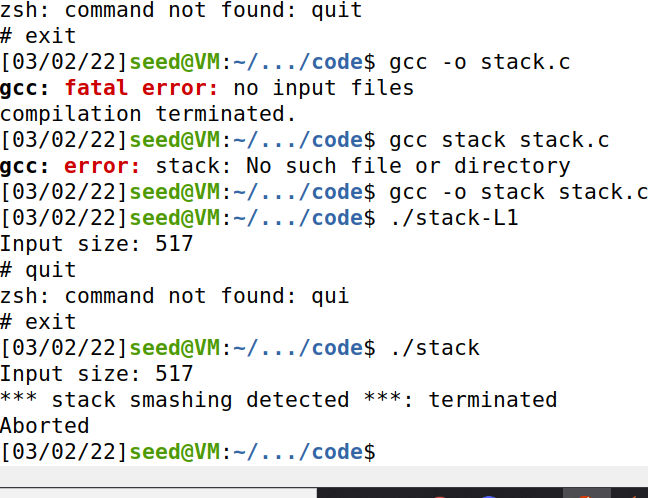
****

* By setting address randomization, we can see that our attacks are no longer working for the addresses that we provide
* Now we run a brute-force approach to hopefully hit our desired return address and in this instance it took me a little bit over 3 minutes to hit the desired address.
* When the address space layout randomization countermeasure is on, the stack frame’s pointer is always randomized and different so it’s not nearly as easy to find the starting point or the offsets that we need. So the only approach here now is entering millions of different combinations until we reach our desired output.

**Task 9**

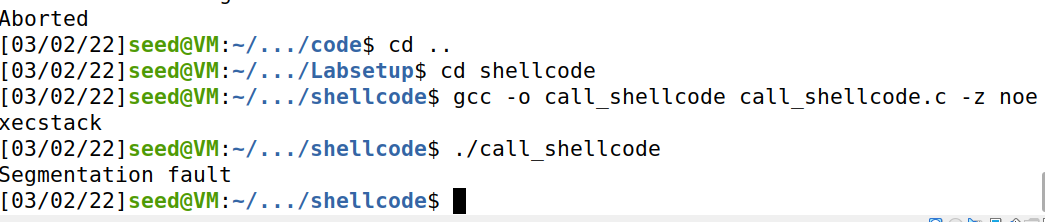


* Working without the StackGuard protection on

****

* Here we can see that the program recognized the buffer overflow attack and it can be detected and prevented.

**Task 9.b**



* Here we can see we can no longer run this because the stack is no longer executable. The program that we are inserting into our code now is only read-only data so it does not provide much help to us when we want to launch our malicious attack.