Buffer Overflow Attack Lab (Set-UID Version)

By Sebastian Medina

Content:

2. Environment Setup

1. Overview

In this lab our goal is to be given a program with a buffer-overflow vulnerability and find a way to exploit the vulnerability and gain the root privilege.

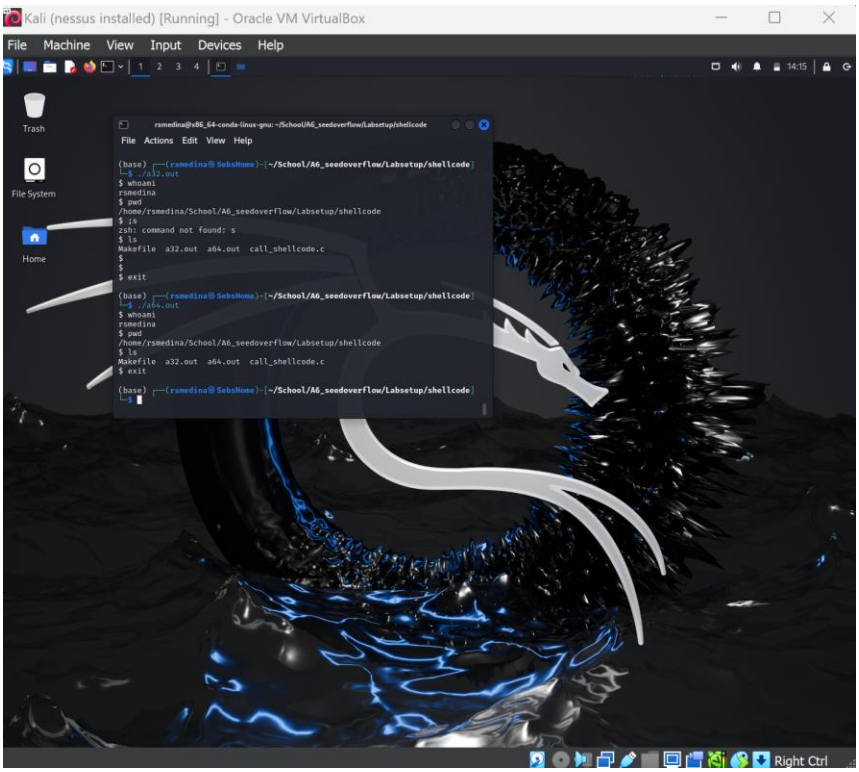
2. Environment Setup

Before we start the lab, we must turn off countermeasures that most modern OS have, that prevent and make buffer-overflow attacks difficult. When we use the commands “sudo sysctl -w kernel.randomize_va_space=0” and “sudo ln -sf /bin/zsh /bin/sh” we disable the countermeasures on our OS that allow us to launch a successful buffer-overflow attack easier for the purpose of learning.

3. Getting Familiar with Shellcode

With Buffer-overflow the goal is to inject malicious code into the target program, which most often than not is in Shellcode. Shellcode is the most used code-injection attacks. Shellcode is just code that launches a shell in the

terminal. For this assignment SEED labs gave us a .c file with written Shellcode within in. Also SEED labs gave us a make file to compile the .c file however it gives us two binary codes a a32.out for 32 bit and a 64.out for 64 bit. When ran both binary files they both launched a shell within the terminal (as shown below) which is what Shellcode needs to do.

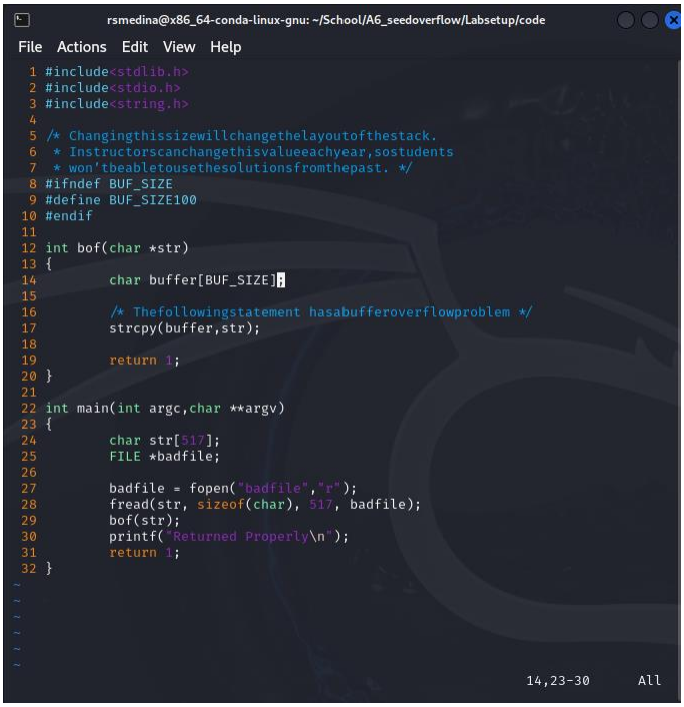


If we dive into the call_shellcode.c file which was given to me from SEEDLabs you can see that all it is doing is calling a shell in either 32-bit or 64-bit depending on how you compile it.

```
└─$ cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}


(base) ┌──(rsmedina㉿SebsHome)-[~/School/A6_seedoverflow/Labsetup/shellcode]
└─$
```

4. Understanding the Vulnerable Program

In this step we look over the code provided to us by the SEED lab named stack.c Within this code there is a buffer-overflow vulnerability within the bof() function. The first input reads a file called badfile, then passes it into the function bof(). The original input can be as long as 517 bytes however the buffer within the function is only as big as the BUF_SIZE variable witch is less then 517 bytes. Since there are no safeguards buffer-overflow will occur and since stack.c is root-owned Set-UID program we can get access to a root shell via the badfile being inputted.

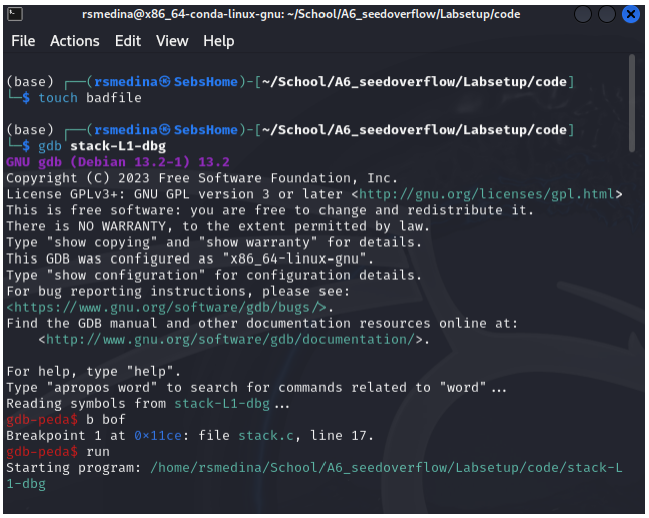


5. Launching Attack on 32-bit Program

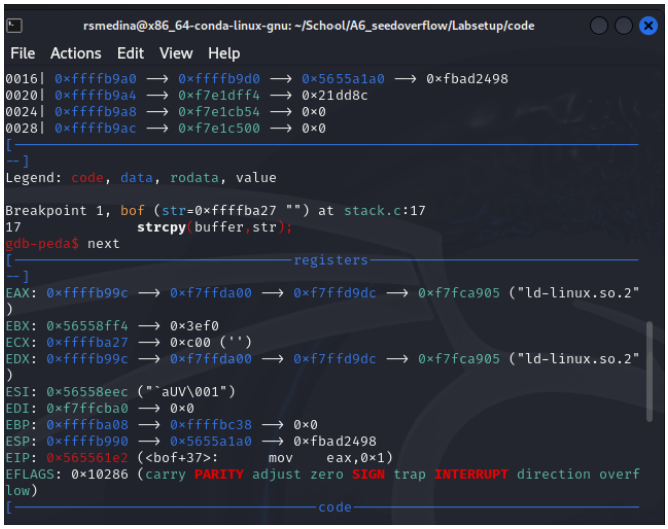
In this step our goal was to successfully launch a Bufferoverflow attack on a 32-bit program. To successfully run a Bufferoverflow attack we first need to know the distance between the buffer's starting position and where the return address is stored. To start this process, I compiled the stack.c file with

the make file given to me by SEED labs which compiled stack.c with debugger the flag -g so that the binary has debugging information (name of this out file is stack-L1-dbg).

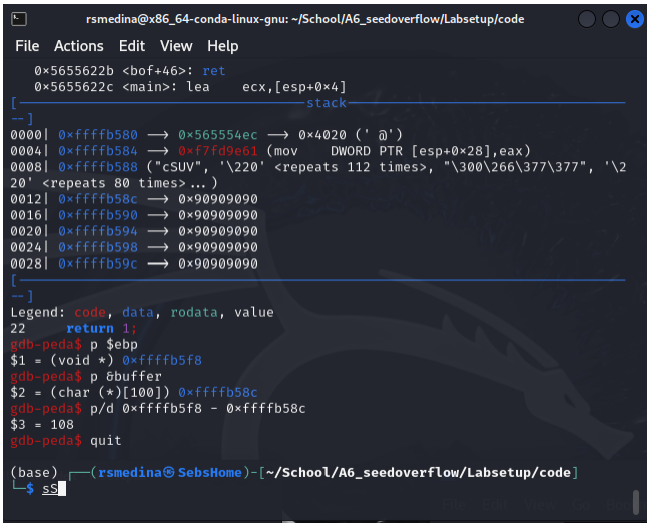
The first step after I compiled stack.c was creating a file named badfile which was called in stack.c, then I ran “gdb stack-L1-dbg” so that gdb can be opened with the file stack-L1-dbg. The next step is to put a break on the bof() function with the command “b bof”. What this does is stop gdb inside the bof () function before the ebp register is set to point to the current stack.



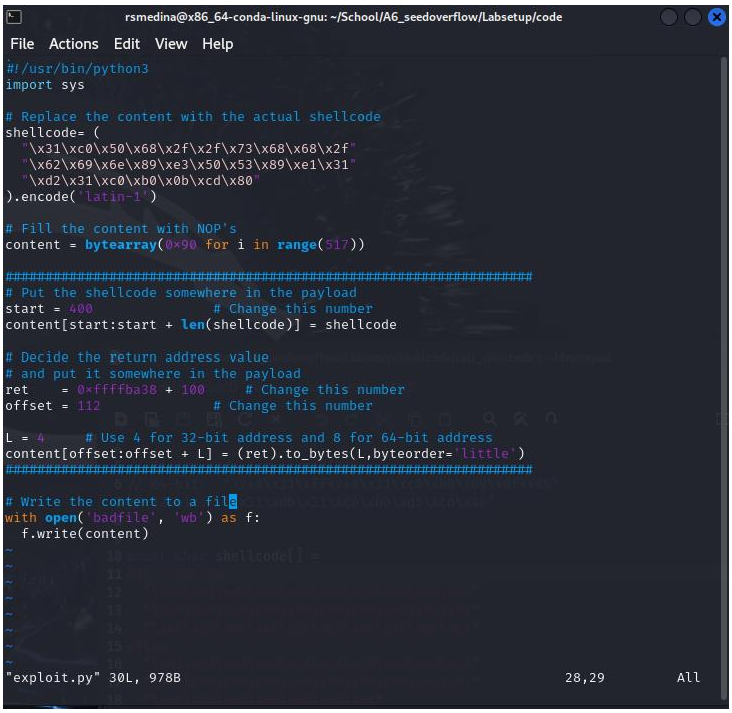
After gdb has stopped we use the “next” command in gdb to execute a few more steps in the program and then stop after the ebp register is modified so that it can point to the stack frame of bof () function.



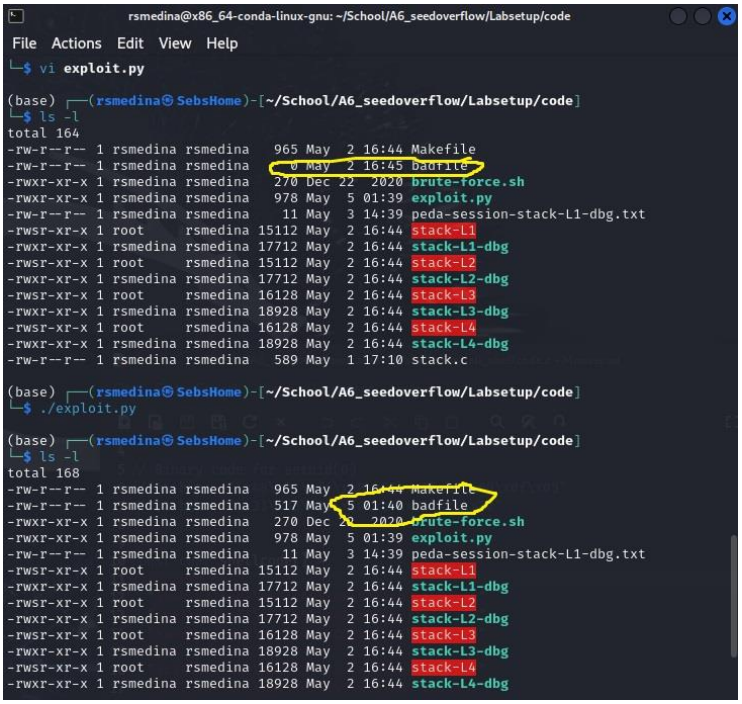
Now that the ebp value has been modified, we must get that ebp pointer value. To do this I used the command “p $ebp” and I got “$1 = (void *) 0xffffba08”. Next we have to get the buffer’s address and to do this I used the command “p &buffer” and that got me “$2 = (char(*)[100]) 0xffffb99c”. Now all I had to do was run the command “p/d 0xffffba08 -)xffffb99c” and get “$3 = 108” and this is the distance



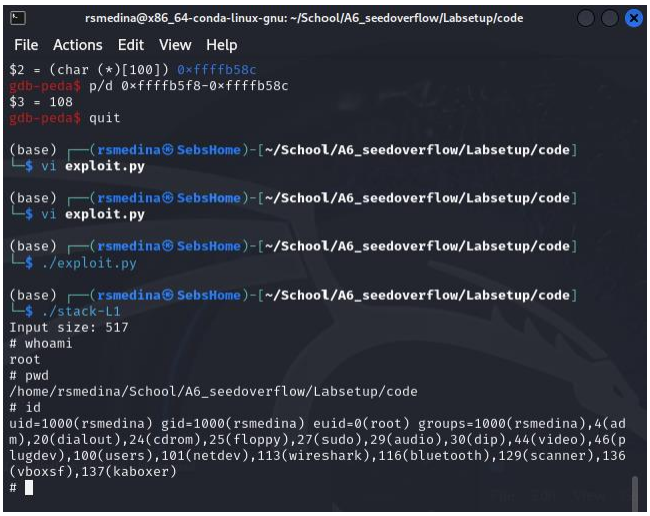
Now we have gathered all the information we need to construct a payload and launch an attack. SEED Labs provided us with a file name exploit.py which is the payload that will be put into the file badfile. However, the exploit.py file by itself is not complete so I had to make some edits first. For the shellcode part I used the 32-bit shell code from callshell.c that was provided to me by SEED labs. After a few guesses I settled on 400 for the starting point. For the return address I inputted the ebp value and added 200 for it. Lastly, I took the offset of 108 and added 4 to get 112. With this information I edited exploit.py so that it uses all the information I just explained.



After I completed all the edits for exploit.py I ran “./exploit.py” witch took badfile witch was empty and inputted the payload to complete a bufferoverflow attack into it.

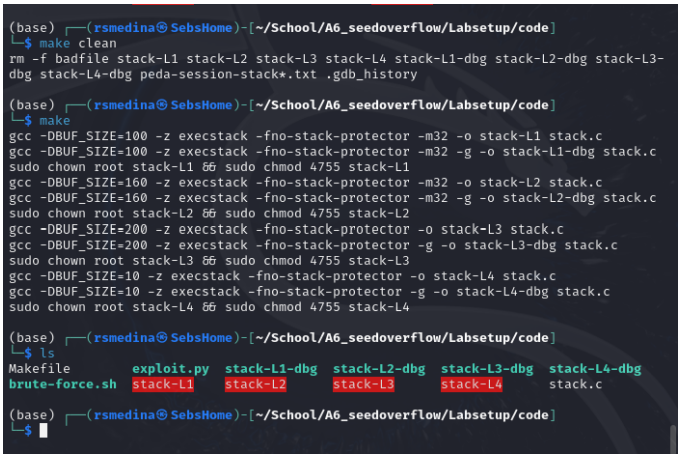


Lastly all I had to do now is run the complied stack.c named stack-L1 with the “./stack-L1” command and as seen below I can successfully exploit the program and gain root access.

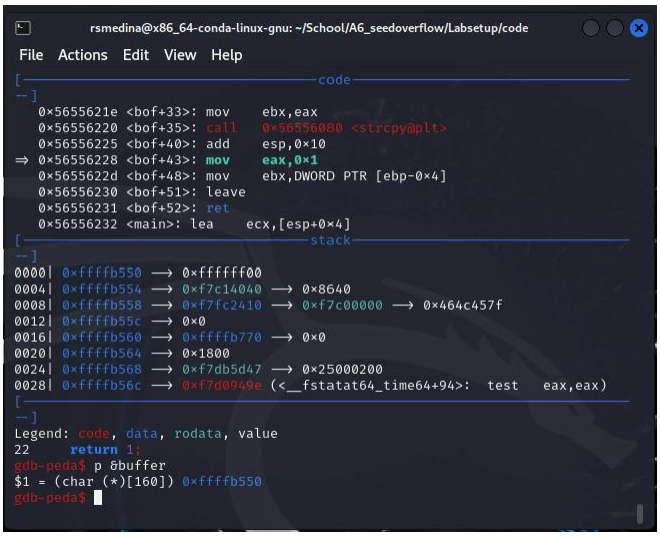


6. Task 4: Launching Attack without Knowing Buffer Size (Level 2)

In this task I had to successfully complete a bufferoverflow attack without knowing the size of the buffers, just knowing that the range is 100-200 bytes. To start this assignment, I used the command “make clean” so that the make file will delete all unwanted files and start from new. Afterwards I ran the “make” command so that stack.c can be complied into stack-L2-dbg witch is complied with the debugger flag and stack-L2 witch is just the binary for stack.c.

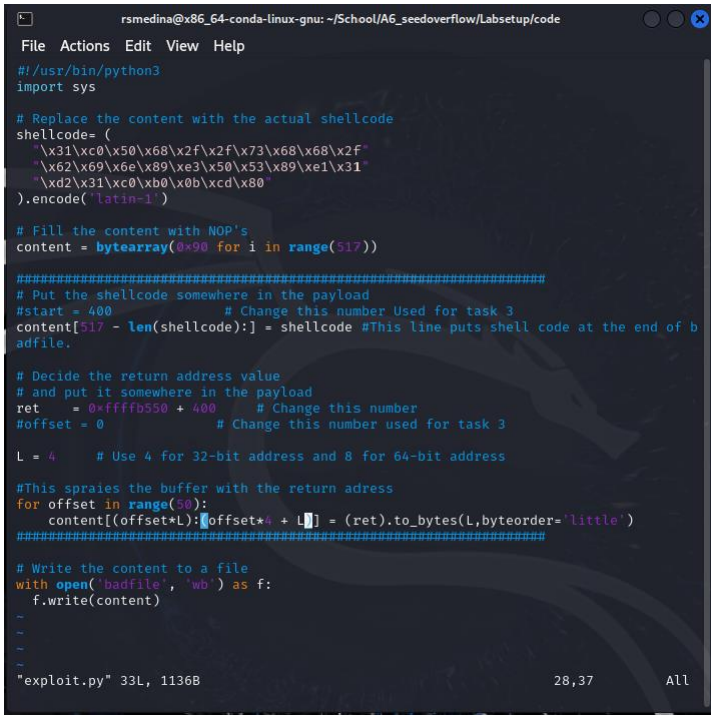


Next, I created the file badfile and used gdb to debug stack-L2-dbg just like I did in section 5. First, I launched gdb with the command “gdb stack-L2-dbg” to have gdb launch with that program. Next, I created a break at the bof () function with the command “b bof” so that the program stops before the ebp resister is set. After it breaks you need to input next so that the program can modify the edp pointer and run the rest of the program. Once is completes we need to know the buffer’s address and I used the command “p &buffer” and it outputs “$2 = (char (*)[160]) 0xffffb550”. Since we are not suppose to know the edp value I do not use the command “p $edp”.

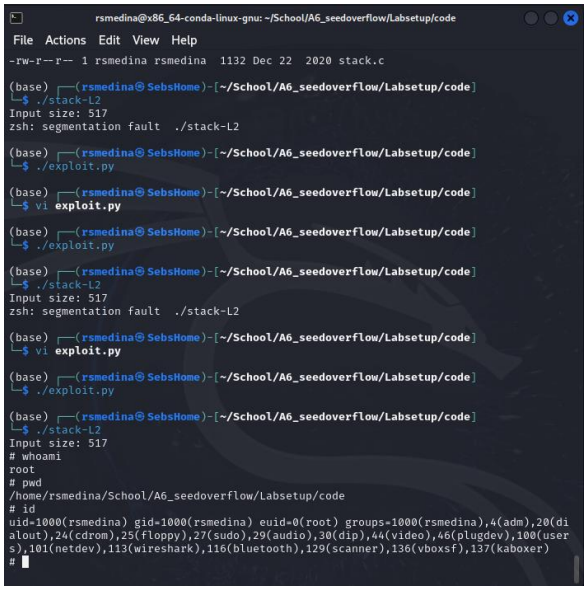


After we gathered all this information the next thing, I did was edit the exploit.py so that the attack can be successful. So, I kept the same shellcode for 32-bit from section 5 in the same spot. Since I did not know the size of the buffer I did not know where to start so I ended up just commenting out that line. The next thing I edited was the return address, this took some guessing. I entered the buffer’s address here (0xffffb550) and then a number that will exceed the buffer. First, I entered 300 but this was not big enough, so I changed it to 400 and allowed me to exceed the buffer. The next step ask for the offset but since I am not suppose to know that I just commented it

out. Lastly, I need to have the buffer address with the correct return address and to do this I added a for loop that sprays the buffer with the return address until I get the correct one.



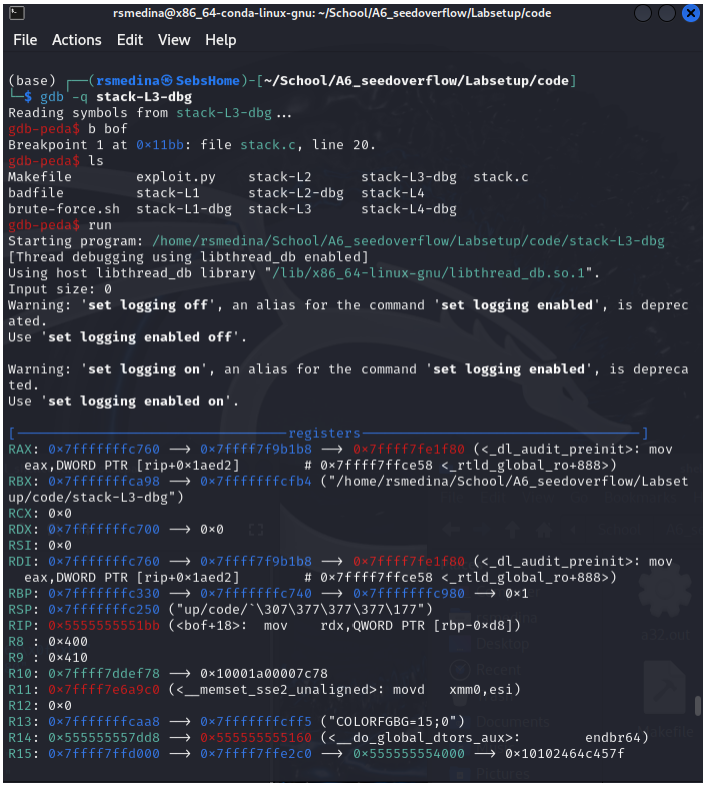
Now that exploit.py has been edited I need to do the same steps as in section 5. I used the command “./exploit.py” to run the program and insert the payload into the file badfile. Once this was done I ran the stack.c program with the command “./stack-L2” witch calls badfile and I successfully was able to exploit the bufferoverflow vulnerability in stack.c and gain root shell access.

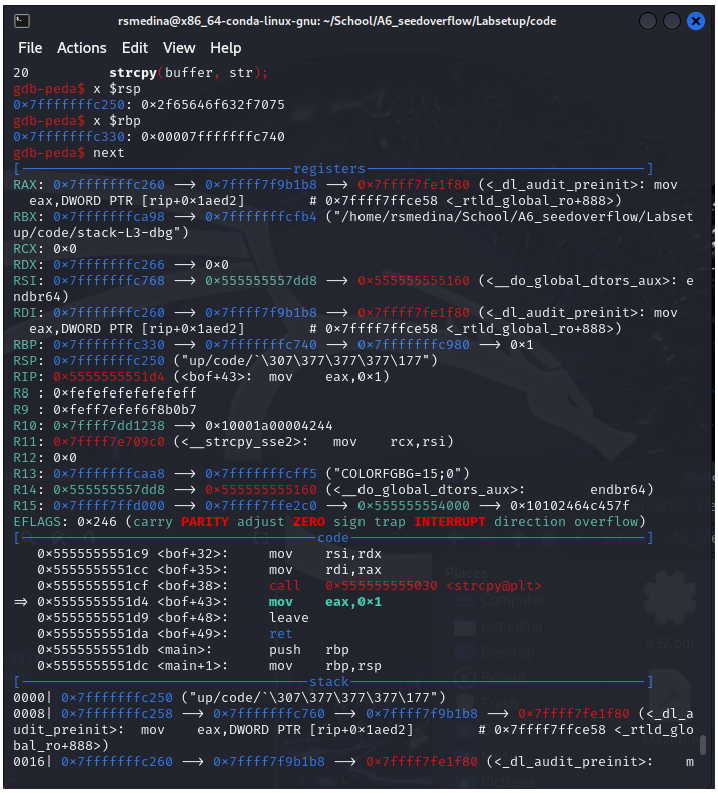


7. Launching Attack on 64-bit Program (Level 3) Extra Credit

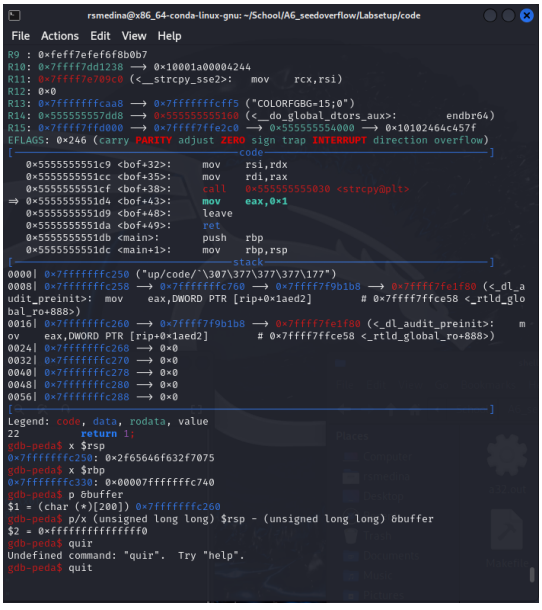
For this task it was very similar to task 3, which was launching an attack on a 32-bit program. The difference in this task is that we must launch an attack, a 64-bit program instead. Most of the steps I took were the same as in task 3 minus some minor items.

I started with the commands “make clean” and “make” so I got rid of any files I made and have a fresh start with newly compiled files the way they are supposed to be compiled. The first step was to use gdp on the file stack-L3-dbg with the command “gdb -q stack-L3-dbg”. Once I was in gdb I set a break at the bof function with the command “b bof”. This will stop the program before the rbp register is set witch is the same as the ebp register for a 32-bit.





Once the program is stopped, I used the “next” command to continue the program and then stops after the rbp register is modified to point to the stack frame of bof () function. I then found the hexadecimal of rpd, rsp and buffer so I can use that information to construct a payload.



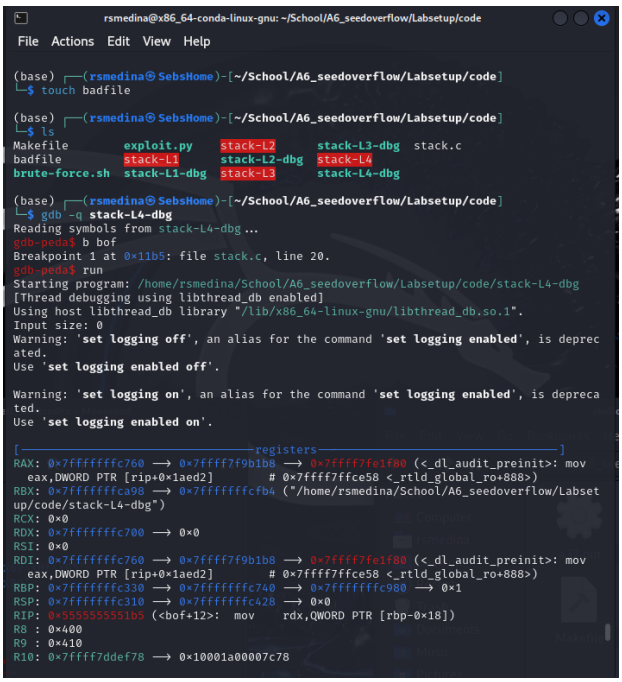
Once I have gathered all the information, I modified the exploit.py that SEED labs gave us as seen below. Some major changes were that I entered the 64-bit shellcode instead of the 32-bit shellcode from call_shellcode.c. Next, I modified the start because to always starts at the beginning of the buffer. Then, I used the rsp address for the return address and lastly changed L to be 8 because it needs to be an 8 instead of a 4 because 4 is only used in 32-bit.

After all of this was done I complied exploit.py with the command "./exploit.py" witch put the payload into the file named badfile. Once this was done then I ran the complied stack.c with the command "./stack_L3" to run the code and successfully launch the attack.

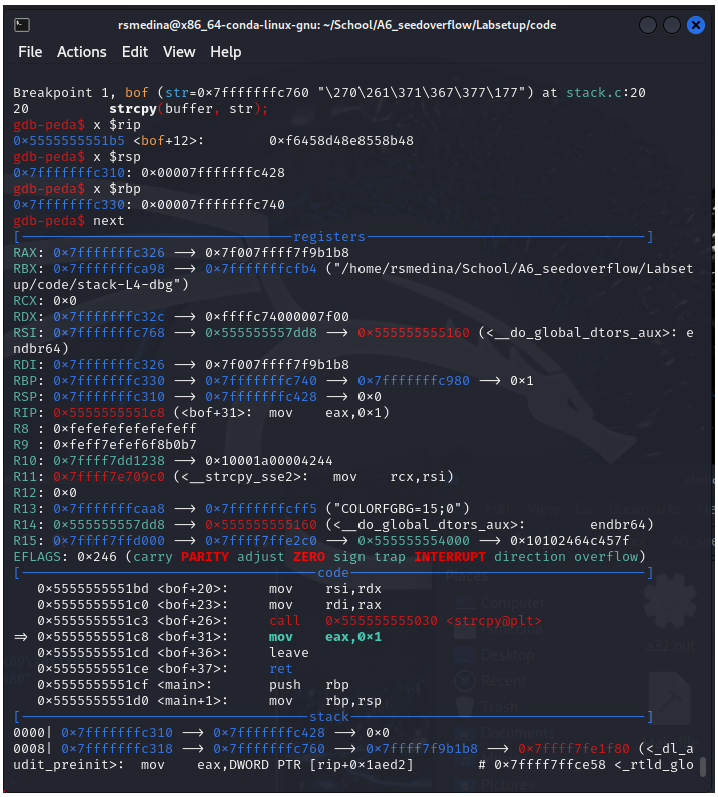
8. Launching Attack on 64-bit Program (level4) Extra Credit

In this level it is the same as level 2, where you must execute an attack without knowing the size of the buffer except this time the buffer size is very small. SEED labs set the buffer size to 10 for this level.

Just like every level before I started with the “make clean” and “make” commands to refresh all needed files and get rid of the ones I do not need. Then after this I created a file named badfile then ran gdb with the command “gdb stack-L4-dbg”. This allowed me to run gdb with the stack-L4-dbg file. Once I was in gdb I created a break at the bof () function with the command “b bof” This well stop the program right before the rbp register is set.

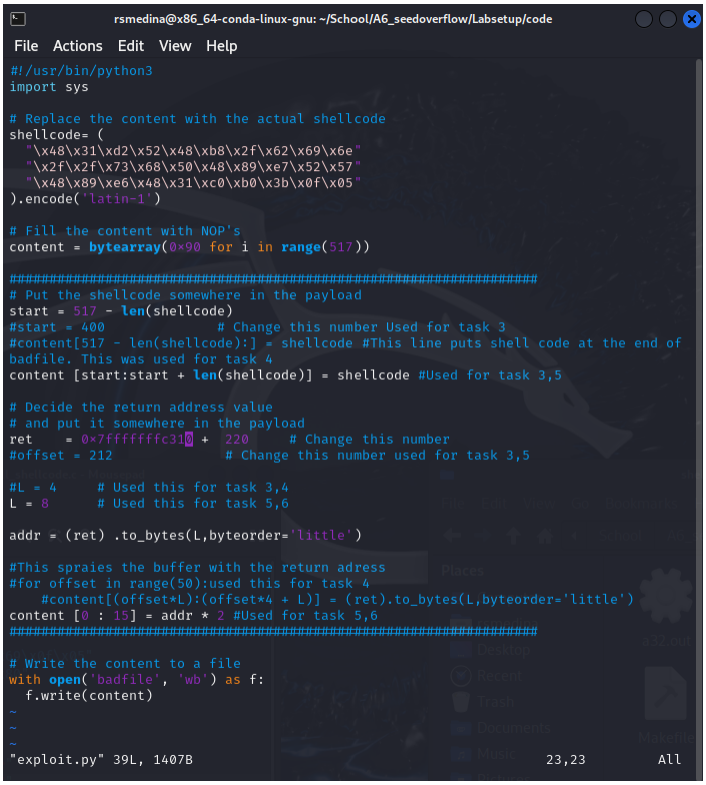


Then after it stops the program, I use the next command to allow the program to continue until rbp register is changed to point to the function. Once I have done that, I look up what the rbp and rsp register are as well as &buffer so I can edit the exploit.py.



Once I was done using gdb and finished gathering all the information the next step is to edit the exploit.py file with the correct information. So I kept

the 64-bit shellcode from callshellcode.c in the same spot as in level 3. I changed the return address to match rbp then added 220 to it. Then since the buffer is so small I made the content and size only 15 because it does not need to be big.

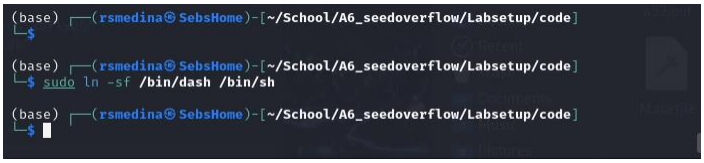


After the exploit.py was complete I ran exploit.py which inserted the payload into the file named badfile. Once badfile had the payload I ran “./stack-L4” witch took in the contents of badfile and allowed me to successfully achieve a bashoverflow attack.

9. Defeating dash's Countermeasure

In this section I had to figure out a way on how to get by the dash's countermeasure. The dash counter measure is when the OS drops privileges when it detects that the UID program does not equal to the real UID.

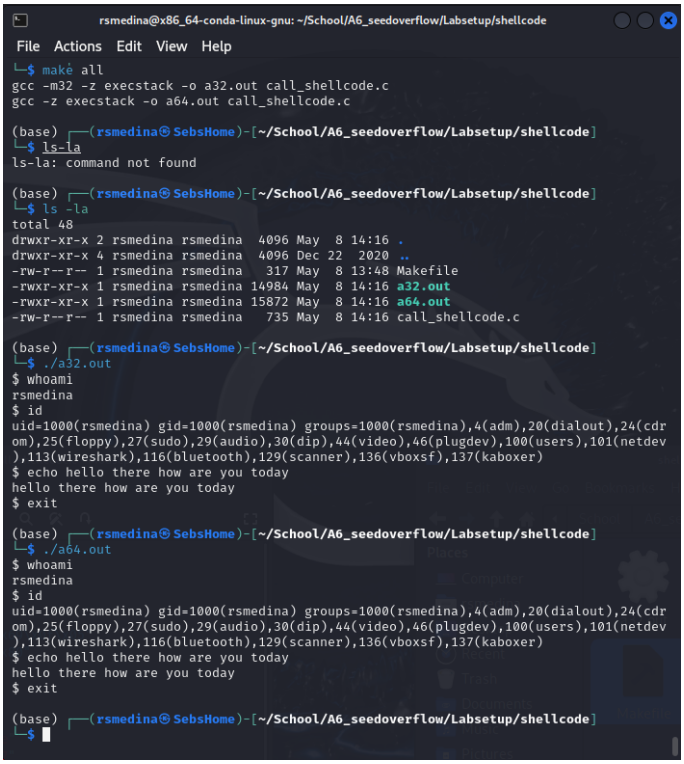
To try and get around the countermeasure I had to first turn on the countermeasure with the command shown below.



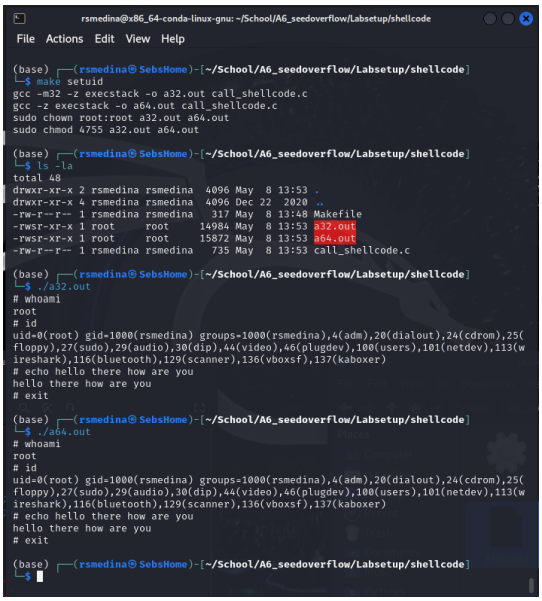
To make sure that I can successfully use a buffer-overflow attack all I had to do was change the real UID to that it equals the effective UID. SEED labs already gave us the necessary binary code so that I can invoke setuid(0). So I

took the binary code they gave me and added it to the beginning of the 64 and 32 bit shells that we are trying to open. Shown below is the change I made in call.shellcode.c.

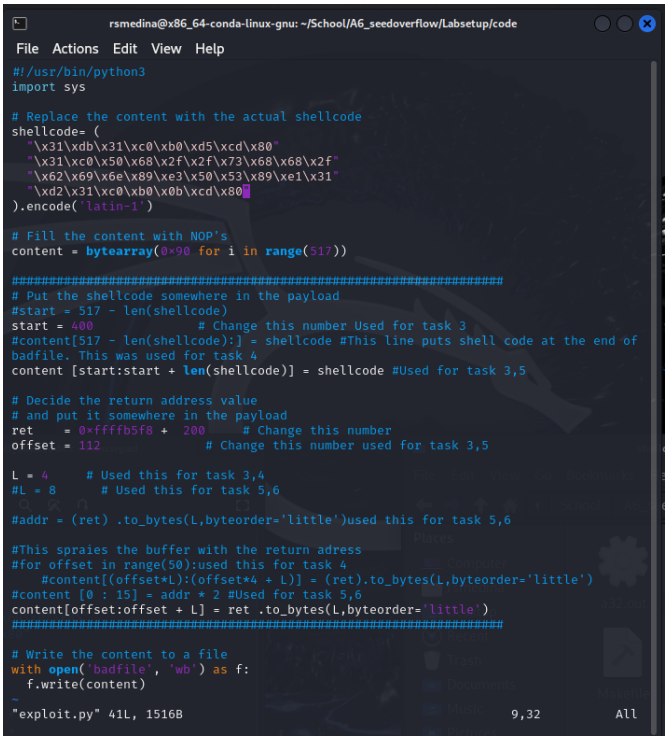
For the next step what I did was compile call.shellcode.c with and without setuid(0) then ran the outputs and observed what they did. I first complied without the setuid(0). I used the command “make all” so that the make file can compile call.shellcode.c without the setuid(0). When I ran that command, it gave me two run files, one named “a32.out” and “a64.out”. I ran both binaries and witnessed both under the control of just the current users not root. Also, the id was 1000 instead of 0 meaning if I tried an attack with these shell the dash countermeasure will prevent the attack.



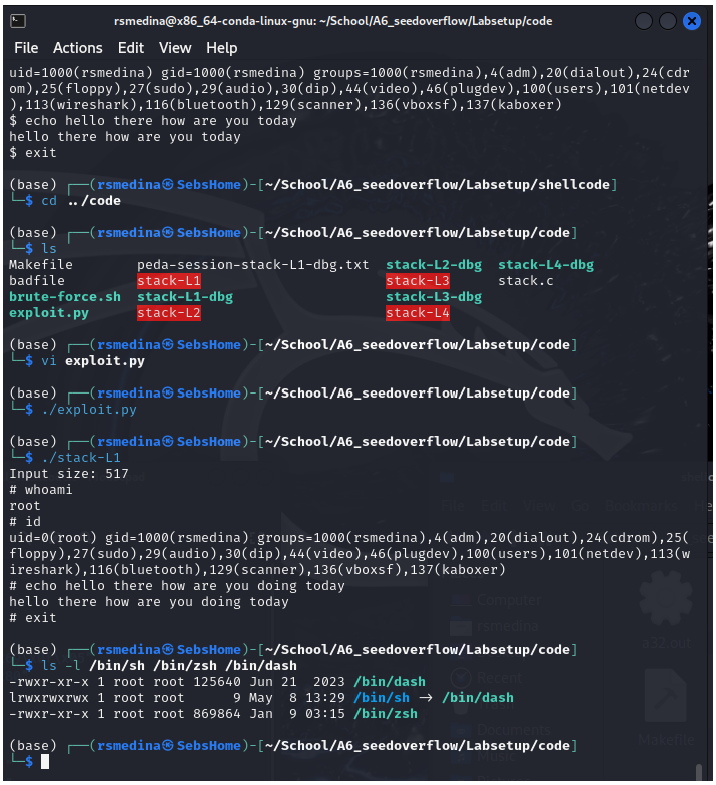
After this experiment I used the command “make setuid” witch complied call.shellcode.c into the binaries “a32.out” and “a64.out” however they were both ran with setuid(0). When I ran both shells, I noticed that this time they where under root control and both shells were under the uid 0. This means I can use both shells to get around the dash counter measure.



Once I changed the shells to be ran with uid 0 I redid the attack from level 1. This was just a buffer overflow attack on a 32-bit program. I took the new binary that I edited on call.shellcode.c and put it into exploit.py so it can call the correct shellcode in order to get by the countermeasure.



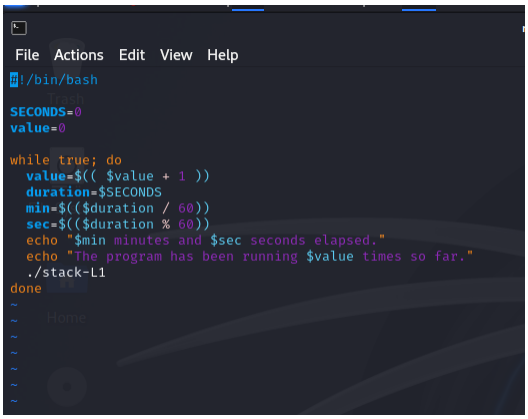
Once the exploit.py was edited I launched the attack. I ran exploit.py which inputted the payload into the file named badfile. Then I ran “stack-L1” and the attack proceeded. Afte running “stack-L1 I was able to get root shell access and successfully completed the buffer overflow attack with the dash counter measure turned on. Also to check that the dash countermeasure is turned on I ran “ls -l /bin/sh bin/zsh /bin/dash” to make sure that my buffer overflow was able to run with the dash counter measure on.



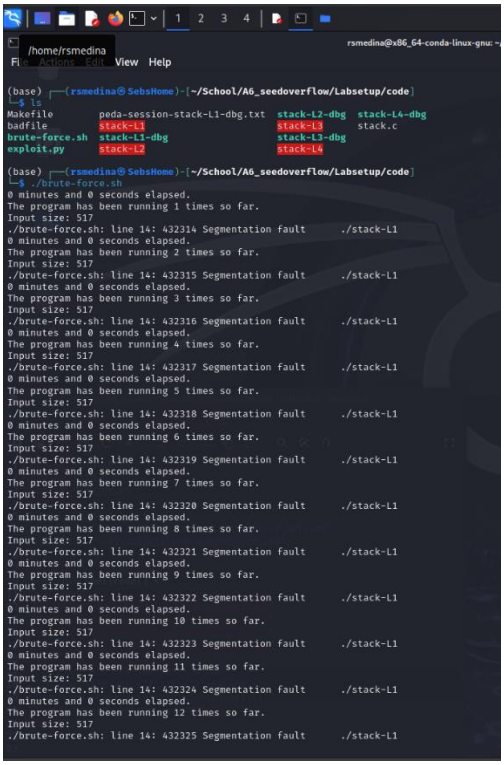
10. Defeating Address Randomization

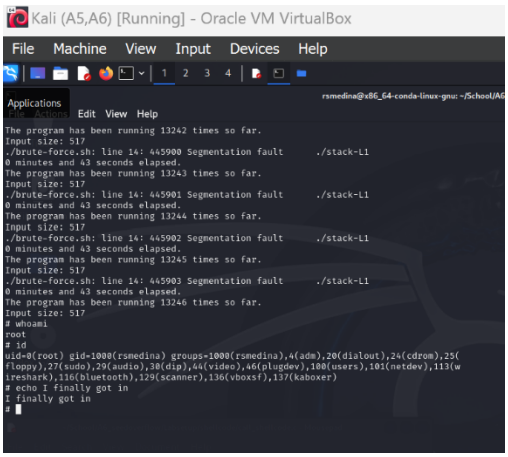
In this task I must turn on address randomization countermeasure on the 32-bit program. What this countermeasure does is assign a random stack base address to the program and if my file does not match that address that will fail because the countermeasure blocked it. You can see this in the screenshots below. I enabled the countermeasure and ran the program expect it gave me a segmentation error because the stack base address did not match the address of the program.

The work around that I did was a brute force approach. Since a 32-bit program only has 524,288 possibilities that the stack base address can be, a brute force attack is a good way to get past this countermeasure because that is not that many possibilities. I used the brute force code that SEED labs gave me and what it does is take the program we want to run witch in this case is “stack-L1” and but it in a while loop. What this means is that the brute force program will run stack-L1 over and over again with a new address attached to the file going into stack-L1. The loop will keep running and failing until the two addresses match and give me access to the root shell.



Since the attack on “stack-L1” was already setup for the last section I just had to run the brute force script until I gained access to the root shell. I executed the brute force script then waited and eventually was able to get into the root shell. This means I was successful in executing a buffer overflow attack with the address randomizer countermeasure turned on.

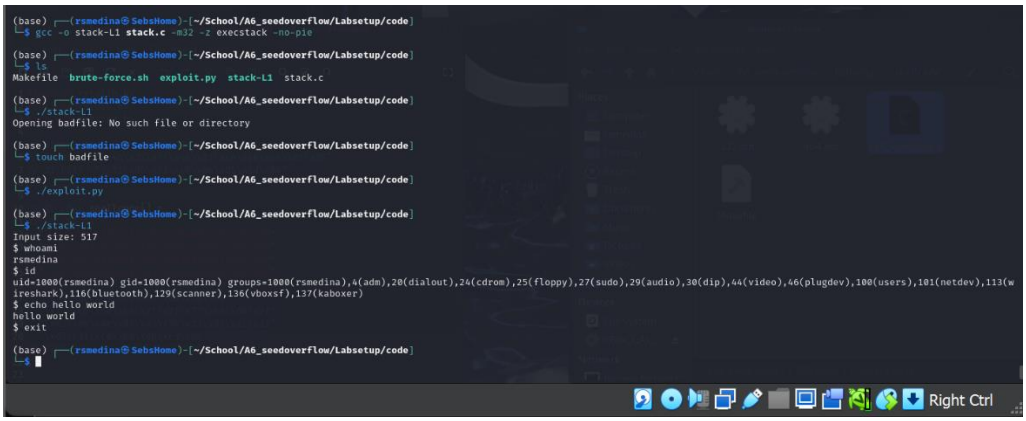




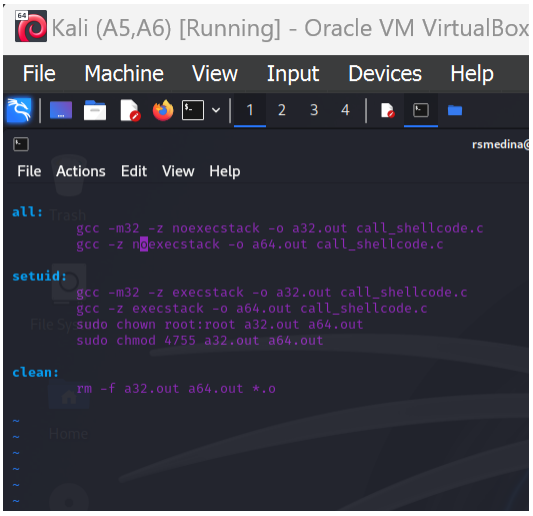
11. Experimenting with Other Countermeasures

First SEED Labs ask me to run the BufferOverFlow first before turning on the StackGuard Protection. So, I ran the program, and the attack still goes though successfully as seen below.

Now I turned on the StackGuard Protection countermeasure by compiling the program without the -fno -stack -protector flag. When I ran the program, I still gained access to the root shell successfully completing the attack.



Now the next task in this I made the stack non-executable. So, to do this, I went into the Makefile that SEED Labs made for me in the shellcode file and where it said -z execstack I changed so that it said -z noexecstack making the stack non-executable when compiled.



Now that I changed the Makefile to do the correct steps in this task I now compiled call_shellcode.c with the “make all” command and then ran “./a32.out” and “./a46.out”. When I ran both, I got a segfault. This countermeasure is preventing the direct execution of shellcode when it was placed on the stack.

This completes all of section for this SEED Lab.