# CSCI 476: Lab 03

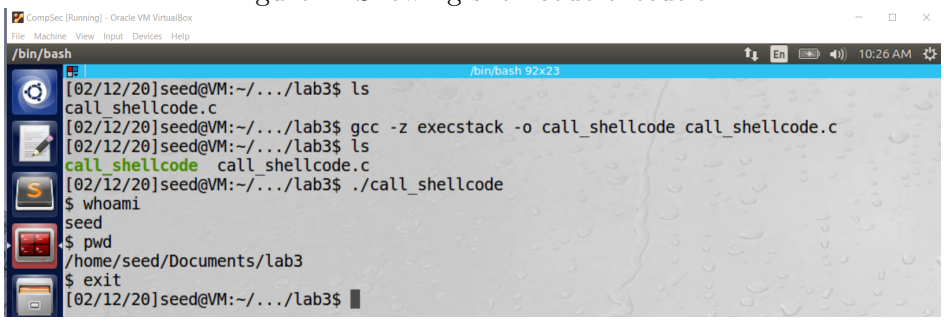
Nathan Stouffer

February 13, 2020

# Task 1

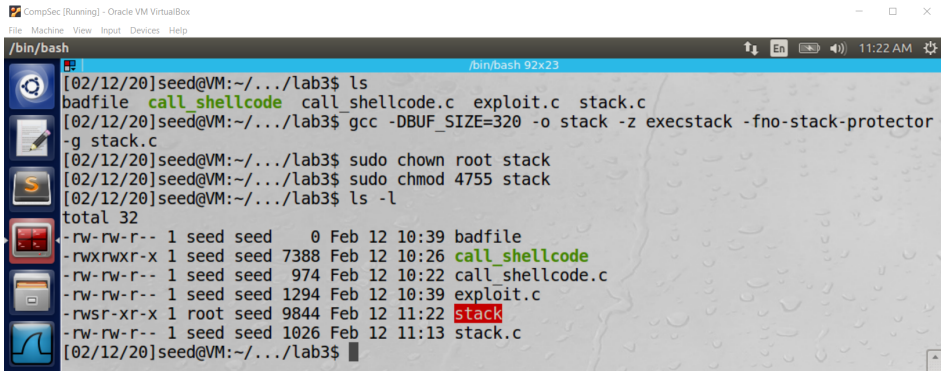
Task 1 first asks that we familiarize ourselves with executing shell code. Figure 1 shows a program called "call_shellcode" which opens a shell. As the parent user is seed, the resulting shell's user is also seed.

Figure 1: Showing shell code execution




The next task was to set up the "stack" program to be ready for use in future tasks. This is shown in Figure 2. We first compile stack.c with the flags for setting the buffer size defined by the lab assignment, making the stack executable, and removing the stack protector. Then we make "stack" a Set-UID program.

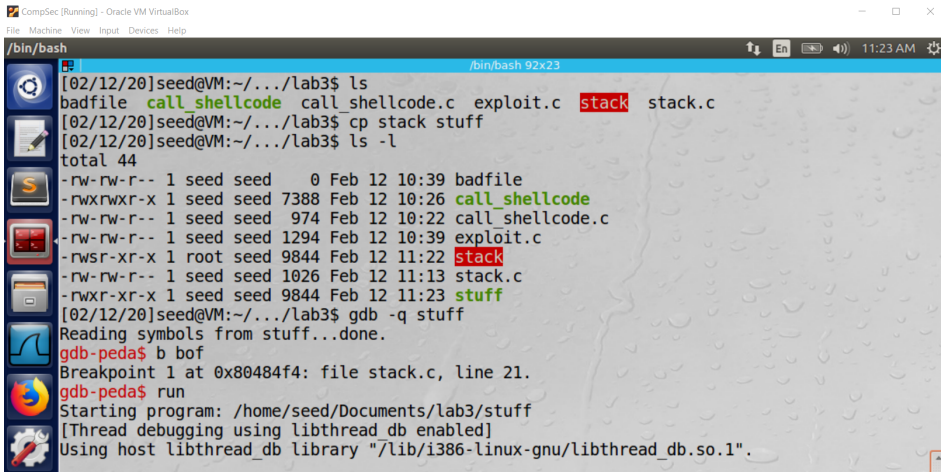
Figure 2: Setting up the stack executable



## Task 2

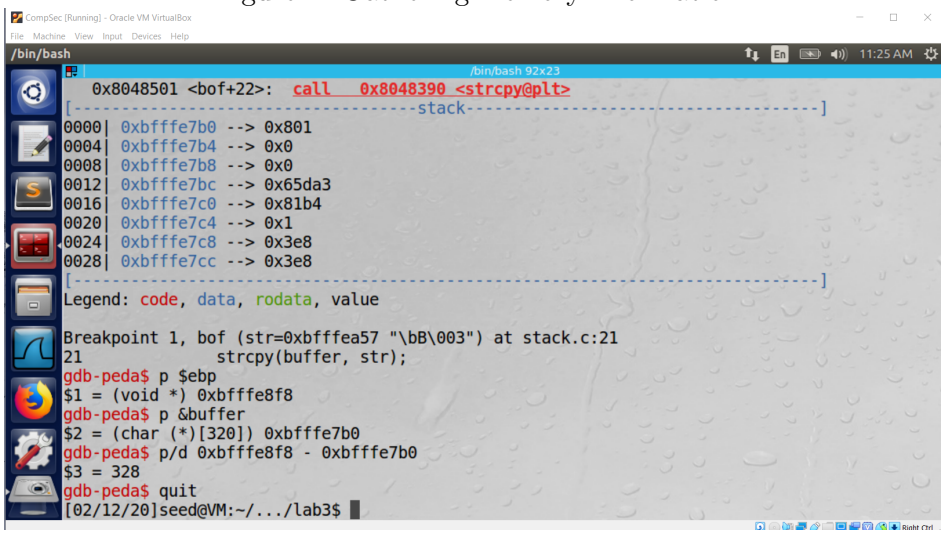
Task 2 is the bulk of this lab. We begin by finding the necessary information for the buffer overflow attack. The buffer overflow attack relies entirely on an appropriately crafted input file. For this, we will need to know how large the buffer is and how far away the buffer is from the return address. We gain both pieces of information through a debugger. In Figure 3, we show the debugger set up.

Figure 3: Setting up debugger




Now view Figure 4. Here I have printed out the addresses of $ebp and &buffer. To be clear, $ebp is the current frame pointer in the stack. We are interested in this because the return address is always placed 4 bytes above the current frame pointer. We want to know the address of the variable "buffer" because this is where our file will begin in memory. If we didn't know that memory location, we would have to guess at where to place the new return address in our input. For this reason, we also print the difference of the two memory locations.

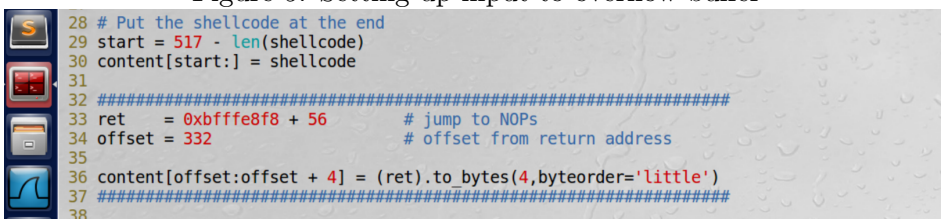
Figure 4: Gathering memory information




With the information gathered in Figure 4, we can set up our cleverly crafted input. To do this, we use a program that was provided on the course GitHub. Only small changes needed to

be made to be specific to my machine. These changes are determined by the memory locations we found for $ebp and &buffer. The difference between $ebp and &buffer was 328, and we know that the return address is 4 bytes above $ebp. To match this, we should place the return address 332 bytes into our input. This is realized in the "offset" variable in Figure 5. But what should we put as the new return address? It seems like a difficult task to exactly land on our malicious code. To make this easier, we insert a bunch of "NOPs" into our input. This way, we only have to hit one of those "NOPs" to slide directly to our malicious code. Figure 5 shows this by setting "ret" to the memory address of $ebp plus 56.

Figure 5: Setting up input to overflow buffer



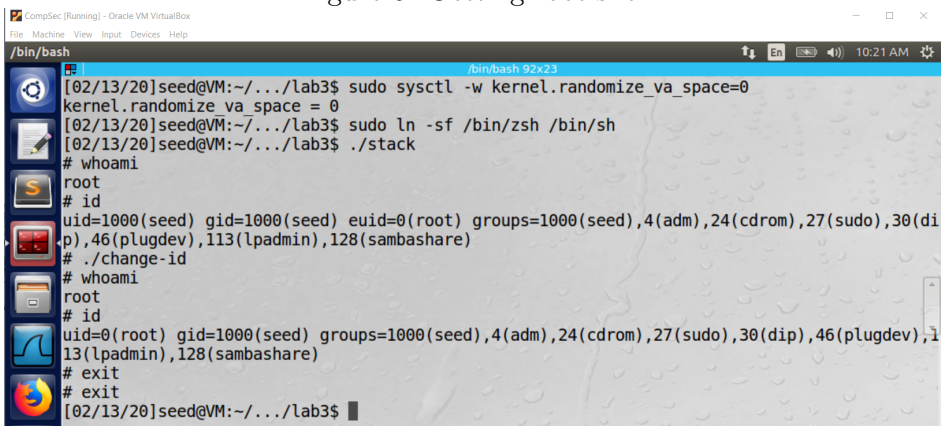
```
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

###################################################################
ret    = 0xbfffe8f8 + 56        # jump to NOPs
offset = 332                    # offset from return address

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
###################################################################
```

Figure 6 shows this code working and us getting the root shell. Notice, however, that the real uid when we get the root shell is still 1000 (seed). Only the effective uid is root. To change this, we run the change-id program. This program first sets the user id to 0 (root) and then opens a new shell. As seen in Figure 6, the new shell has the uid as 0. It only displays uid (and not uid/euid) because they are equal. Also notice that this launches another shell, so we exit twice before getting back to the original terminal.

Figure 6: Getting root shell



```
/bin/bash
/bin/bash 92x23
[02/13/20]seed@VM:~/.../lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/13/20]seed@VM:~/.../lab3$ sudo ln -sf /bin/zsh /bin/sh
[02/13/20]seed@VM:~/.../lab3$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
# ./change-id
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
# exit
# exit
[02/13/20]seed@VM:~/.../lab3$
```

## Task 3

One way we made Task 2 easier was using an unsafe shell. An attacker would not always have this liberty since /bin/dash will drop privileges whenever the detects a difference in the effective and real user id. Task 3 finds away around this problem. Before turning to shell code, we first run an experiment with the function setuid(0). Figure 7 shows code that opens a new shell. Note that setuid(0) is not called in this code.

Figure 7: Showing file that does not change uid

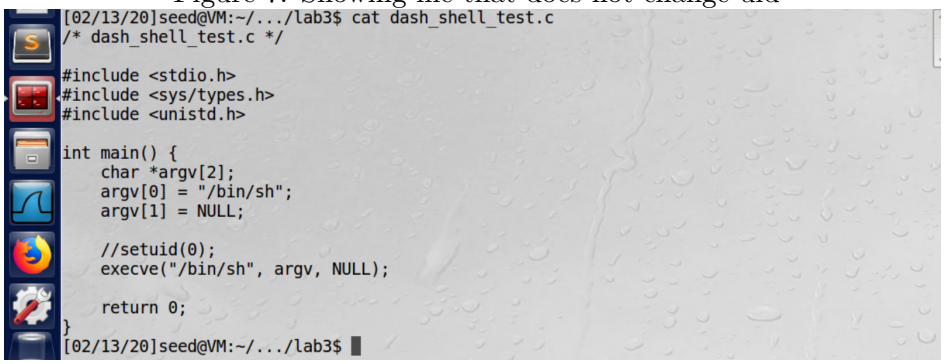
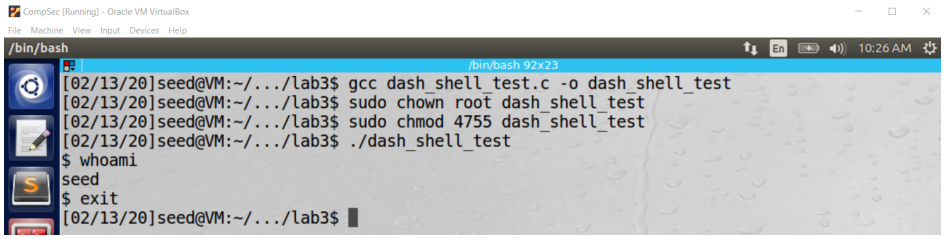



Figure 8 shows the result of running the code in Figure 7. Note that we get a non-root shell.

Figure 8: Running dash_shell_test




Now view Figure 9. The only difference between this and Figure 7 is that we now call setuid(0). If the result of running the executable produced by this c file changes, we know that setuid(0) is the source of that change.

Figure 9: Showing file that changes uid

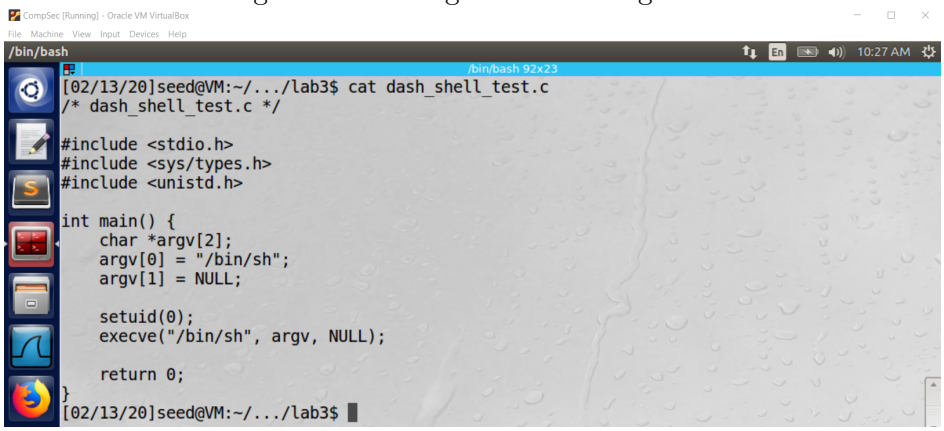


Figure 10 shows the result of the code shown in Figure 9. Note that we get a root shell! So, we don't have to rely on an unsafe shell when using a buffer overflow attack, we can just change our user id before opening the shell.

Figure 10: Running dash_shell_test

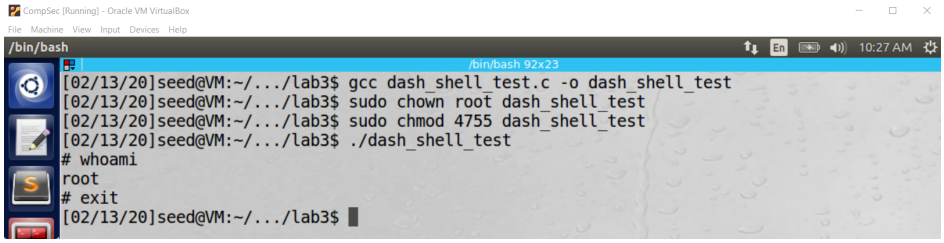
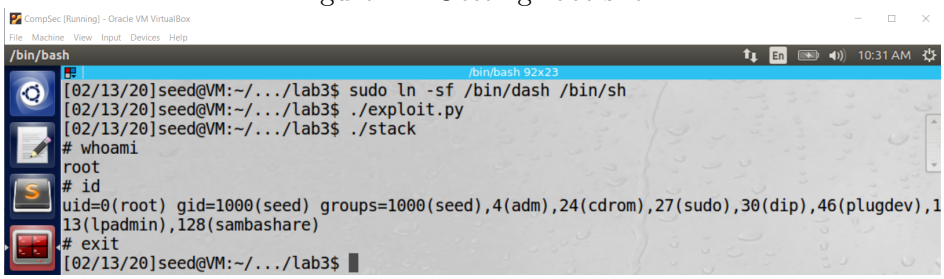



Figure 11 shows the updated assembly that will change the uid before we call the new shell. The rest of the code in our input file remains unchanged.

Figure 11: Showing updated assembly code


Now view Figure 12. We first link /bin/sh back to a safe shell. Then we produce the input file with the updated assembly code (by running exploit.py). Then we run stack and see that we get a root shell. In addition to reducing our reliance on vulnerabilities, we also have no need to gain root privileges and then change our real id afterwards (like we did in Task 2). This solution is much cleaner and more powerful.

Figure 12: Getting root shell



## Task 4

Task 4 focuses on defeating a different protection against buffer overflow attacks: address randomization. To defeat this, we will use brute force. By this, I mean that we will run the stack program until the memory address set in our input match where everything is laid out on the stack. Figure 13 shows that we first turn address randomization on. We then print out the contents the script used to repeatedly call the stack program.

Figure 13: Setting up script

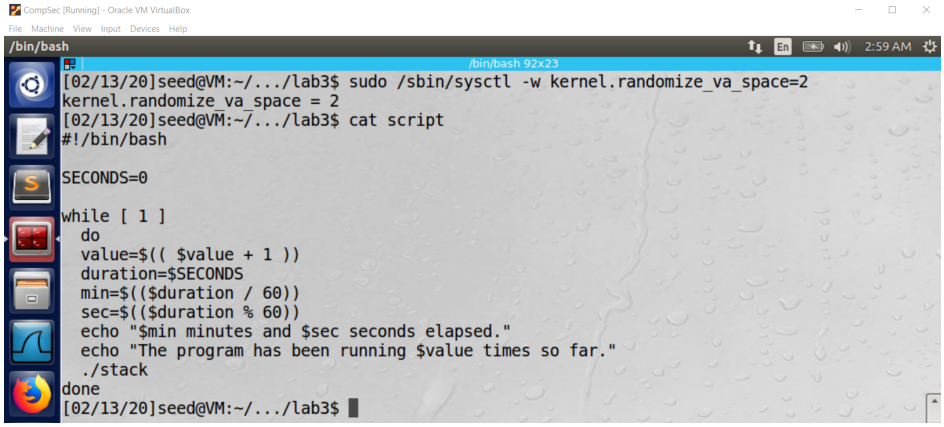
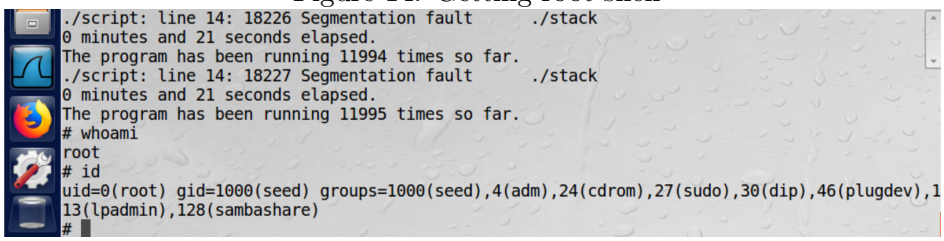



Figure 14 shows the result of running the script. We got really lucky and only had to wait 21 seconds, but this process could have taken much longer. On the $11995^{th}$ run, we got the root shell! Just as in Task 3, we get a root shell with both real and effective uid as 0 so there is no need to further increase privileges.

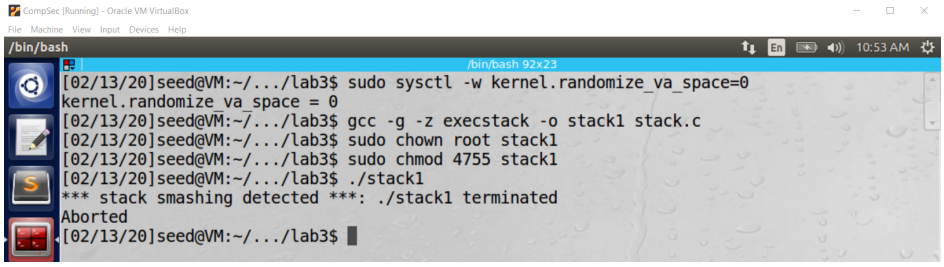
Figure 14: Getting root shell



## Task 5

Task 5 explores what happens when the stack guard is removed. The stack guard is a protective measure that places a value on the stack when a method is called and checks to make sure that value is the same when that method is popped off the stack (the value to check this condition must be stored somewhere that is not on the stack). If the values do no match then there is a high likely hood that the return address of that method was overwritten. The test is shown in Figure 15. We first turn off address randomization and then compile stack.c without the flag "-fno-stack-protector." We then make stack1 a Set-UID program and run it (using the same in put file as in previous tasks). As Figure 15 shows, the "*** stack smashing detected ***" error occurs and the program is terminated.

Figure 15: Showing StackGuard in action

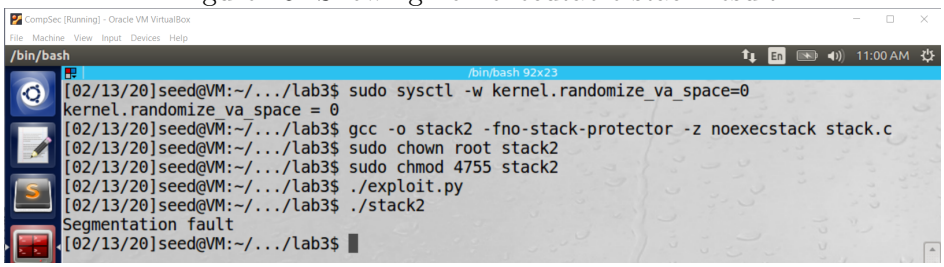



I am not sure how StackGuard could be defeated. My best guess is as follows. We would require two things: the StackGuard's location relative to $ebp and bytes that (when placed in memory) leave current bytes unchanged. I am unsure if either of those exist. If they did, we could place the bytes that leave memory unchanged in the location where the StackGuard should be. That way it would remain unchanged and the program would not crash.

## Task 6

Task 6 deals with the non-executable stack protective measure. Figure 16 shows the result of making the stack not executable. First, we make sure that address randomization is turned off. Then, stack.c is compiled with a flag that makes the stack non-executable. Finally, the program is made into a Set-UID program. We again produce the necessary input using exploit.py and then run stack2. As shown in Figure 16, we get a segmentation fault.

Figure 16: Showing non-executable stack result




It seems as though we cannot get a shell. This is because, even though the code we placed on the stack is valid, the binary does not have permission to run anything on the stack. This makes attacks much more difficult. An attacker has a lot of control over what can be on the stack, but if it cannot be executed, then it useless. However, it is a good security measure because there is typically no need execute code that is on the stack.