**Kaleb Alstott**

**Lab 3 – Buffer Overflow Attack**

**TASK 1**

**A screenshot of a computer

Description automatically generated**

**Getting Familiar with Shellcode:**

In this task we were asked to become more familiar with shellcode. Well the shellcode we were provided with basically invokes the execve() system call to execute /bin/sh. In our first steps to getting familiar with shellcode we had to change directory to the correct location being /shellcode. Once in the correct file we ran the “make” command, this command provided us with both the compiled code for 32-bit and our 64-bit program. The “make” command used, compiled the code in call\_shellcode.c making two binaries that were our 32-bit program (a32.out) and our 64-bit program (a64.out). While using the command "-z execstack" to make stacks executable allowing execution from the stack. The -m32 flag used in the gcc command allows for the 32-bit version to be used while without or with -m64 this flag the 64-bit version will be used.

As you can see when we run these programs by using the “./” command both the 32 and 64-bit programs run as expected both launching us in a non-root shell.

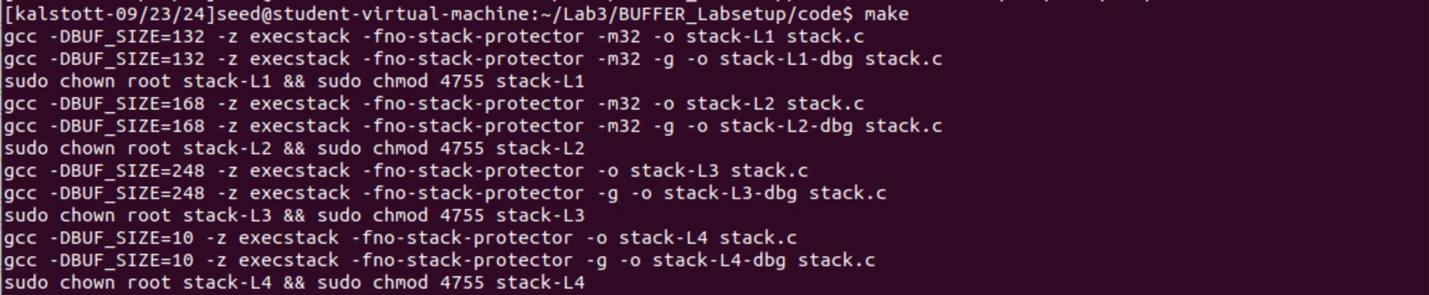
**TASK 2**

**A screenshot of a computer

Description automatically generated**

**Understanding and Compiling the Vulnerable Program:**

Showing change of L1-L4 values for assignment according to last name.

****

In this step, task 2, we were tasked with setup commands in which to compile the vulnerable program stack.c The compilation and setup commands are already included in our Makefile, so the only thing we needed to do here was to type “make” to execute those commands. As you can see above the “make” command was executed allowing for us to allow for specific requirements such as “-fno-stack-protector” to disable stack protection, “-z execstack” making the stack executable, as well as the “-m32” flag in which compiles the program in a 32-bit program.

Once this vulnerable program was set up and compiled, we moved on to creating our badfile that we will see in our next task.

**TASK 3**

**A screenshot of a computer program

Description automatically generated**

**Launching Attack on 32-bit Program (Level 1):**

Above you can see our previous task of running the “make” command as well as our environment setup in which turned off our counter measures such as our address space randomization, “sudo sysctl -w kernel.randomize\_va\_space=0”, as well as our configuring /bin/sh with the command, “sudo ln -sf /bin/zsh /bin/sh” which allows us to link /bin/sh to zsh. We ran the ./stack-L1 command letting us know that our next step was to actually create the badfile that we needed to exploit the buffer-overflow vulnerability in the target program.

After creating out badfile we needed to add the -g flag to gcc command so that our debugging information is added to the binary. We used the command “gdb to debug stack-L1-dbg” to determine our offset as well as provide breakpoints where it is needed for us. For our case it will be used to find the difference between our EBP and our buffer address.

**A screenshot of a computer

Description automatically generated**

Once in our debugger we use the “b bof” command to set a break point at function bof(). We follow this command with the run command which starts executing the program. “We need to use next to execute a few instructions and stop after the ebp register is modified to point to the stack frame of the bof() function.” This is done for our next step explained below.

**A screenshot of a computer

Description automatically generated**

After fat fingering our EBP a couple of times I was able to obtain our EBP value by using “p $ebp”. We follow the same command to obtain our buffer address, “p &buffer”. Once we have figured out our two values, we must subtract the address of our EBP value from our buffer value leaving us with the offset of 140. It is important to note here that the offset will be +4 to the 140 due to the return address (144).

Once we have the information needed such as our return address value, our offset, an idea of where to start to put our shellcode into the payload we can exit. Once exiting we had to make the exploit.py file have executable permissions to run our program. **A screenshot of a computer

Description automatically generated**

The command “cat” here is used just to show the contents of what is inside our badfile, which is our payload in this case. Keep in mind that our badfile is where our program gets its input from. Exploit.py when executed will fill the contents in the badfile. Then we run our vulnerable program stack. Let’s start testing to try and gain root shell access.

**A screenshot of a computer

Description automatically generated**

After a couple of attempts and changing around our return address value somewhere in our payload we were able to gain root shell access.

**A screenshot of a computer

Description automatically generated**

This is our final changed code from our exploit.py file after running numerous attempts to try and invoke a root shell. In this code we were able to obtain and change our shell code from our given file of call\_shellcode. This is important because this shell code is made up of machine code that allows for us to execute a shell, in our case being /bin/sh. The start is important because this is where we determine where our shellcode will be placed within the payload. We were able to guess 400 because of the range of our NOP’s (517), buffer size, offset, and memory layout.

Buffer return address + 300 is used to ensures the program correctly jumps to the desired location, in our case this is where our shellcode is. We figured this number out by taking in account of such factors as padding, location of shellcode, location of buffer return address, possible NOP’s, and of course trial and error. Lastly our offset is decided by subtracting the address of our EBP value from our buffer value leaving us with the offset of 140. It is important to note here that the offset will be +4 to the 140 due to the return address so our offset is 144. Note: 4 is used for the 32-bit address, while 8 is used for 64-bit address.

Our buffer-overflow attack works because it successfully overwrites the return address on the stack and redirects the program's execution to your shellcode. This is done by the values I described above and how we were able to find them as well as why we use them.

**TASK 4**

**A screenshot of a computer

Description automatically generated**

**Launching Attack without Knowing Buffer Size (Level 2):**

Since we did not user the brute force method for this attack like in level 1, we were tasked with minimizing the number of trials we used to conduct this attack. We were able to achieve this by implementing a for loop to out exploit-2.py. First, we had to copy our exploit.py file to our new file named exploit-2.py. I edited our code and implemented a for loop down below in the finished code. Once doing so we were set to run the exploit and run our stack. As seen above once executed we have the expected output of gaining root shell access.

**A screenshot of a computer program

Description automatically generated**

The for loop implemented above is used to test various offsets in order to find the correct position for the return address in our buffer overflow exploit. The loop iterates 50 times where we have our offset take values from a 0-49 range. The code in line 29 is used to calculate the position of the content where the return address will be inserted, followed by the return address being placed at different offsets in the payload which would allow for us to overwrite the return address.

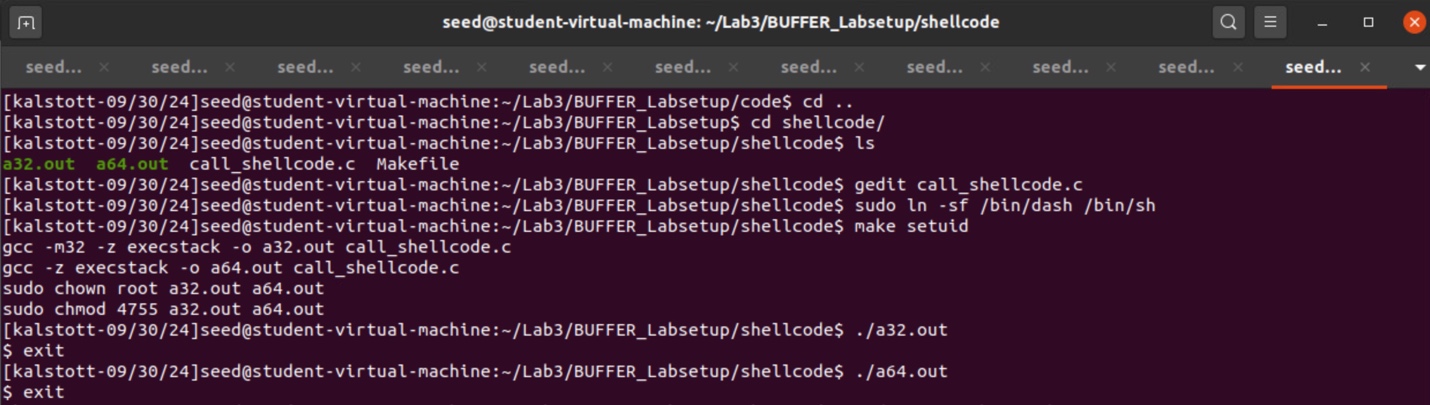
**TASK 5 (optional)**

**TASK 6 (optional)**

**TASK** **7**

**Experiment**

**No setuid(0)**

****

By running the shellcode without setuid(0), this will result in a standard user shell being spawned. Non-root.

Code changed after the no setuid(0) test. We had to implement and add both binary code to the start of the appropriate shellcode[]. This will allow for our setuid(0) so that both RUID and EUID are the same root setuid.

**A screenshot of a computer

Description automatically generated**

**After setuid(0)A screenshot of a computer program

Description automatically generated**

As you can see here we were able to escalate privileges and bypass the countermeasure implemented by dash, by doing so with our setuid(0) we could spawn a root owned shell in which we are able to execute shell commands with root privilege.

**Launching the Level-1 attack again**

**A screenshot of a computer

Description automatically generated**

After running the command above “sudo ln -sf /bin/dash /bin/sh”, this allows for /bin/sh to point back to /bin/dash. This command ensures us that we have set and turned on our countermeasure that is implemented by dash. This countermeasure is in place so that when detected that EUID does not equal RUID the privileges are dropped. We run the command gedit to edit our exploit.py document that is below and explained. By running the next sudo command in the terminal this will allow for us to turn the countermeasure off for address space randomization. After all that is done, we can run our exploit.py and our stack-L1.

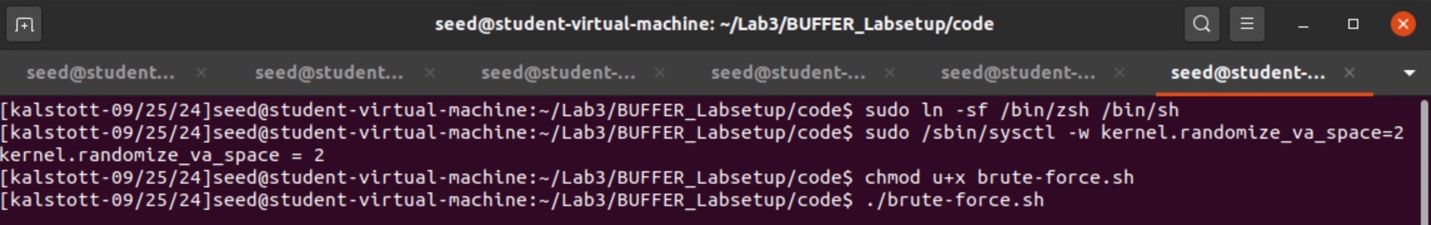
We were able to defeat the countermeasure here by changing the RUID to be the same as the EUID, in our case setting both to 0 (root). We achieved this by invoking setuid(0) before executing execve() in the shellcode. This is what allowed for the invoked the shell program to run as a root-owned Set-UID program, by setting both the RUID and EUID to 0 (root).

**A screenshot of a computer

Description automatically generated**

The new code added to our exploit.py file here is our binary code for setuid(0) 32-bit shell code. This is used so that the RUID and EUID can equal (0) root. This will allow for us to run as a root-owned Set-UID program.

**TASK 8**

****

**Defeating Address Randomization:**

In this task we were challenged with using a brute-force method to defeat address randomization. Our first command is to ensure that we have created a symbolic link from the /bin/sh to the /bin/zsh showing we are in the correct shell. After that our next command is turning on the address randomization. The 2 at the end of the command means that it is “full randomization”. Once we know that the address randomization is in full randomization mode, we change the permission of the program to have executable permission so we can run the program. Following this we run our program “./brute-force.sh”.

**A screenshot of a computer program

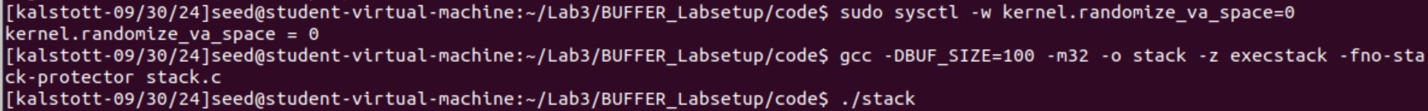
Description automatically generated**

After a long waiting process, (I was able to travel to the grocery store, pick up groceries, and cook dinner when this was happening lol), our brute force attack worked granting us the root shell indicated by the # sign. As you can see my attack took 94 minutes and 56 seconds and had run 11367 times! I was on cloud 9 when I was eating dinner and saw the attack stopped and was successful.

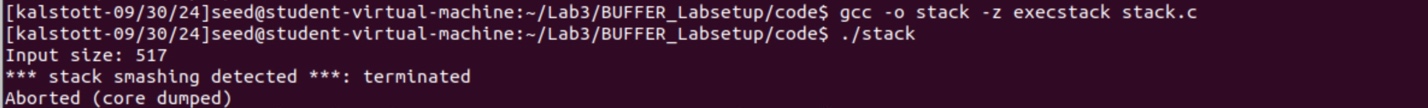
**TASK 9**

**Experimenting with Other Countermeasures**

**A - Turn on the StackGuard Protection**

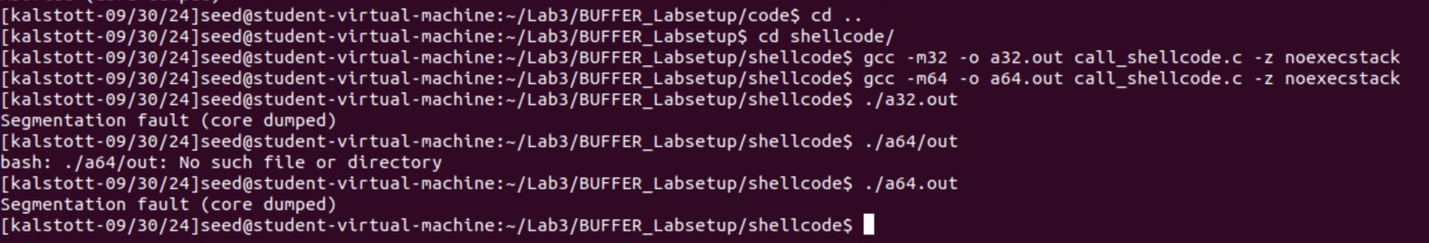


**With StackGuard disabled** – After turning off our address randomization (the 0 meaning no randomization) we were able to repeat the level-1 attack by following the steps of our prior attack. The steps were to compile the program with StackGuard being on meaning we use the -fno-stack-protector. Once done we were able to run our program and show that our attack was and still is successful by showing us root shell access #. This was the expected output due to the prior knowledge of us completing this attack earlier in the lab.



**With StackGuard enabled –** In this part of the task we had to enable the StackGuard to show that the countermeasures we are testing still work and the purpose of the countermeasures.Once again, we ensured that our address randomization was set to 0 meaning no randomization, followed by compiling the program with the default countermeasure (StackGuard protection). We did this by removing the -fno-stack-protector command. Once done we ran our program showing us that the countermeasure has worked. This was the expected output due to us enabling our countermeasure and showing that the attack was unsuccessful with granting us root shell access. It is important to note here that the StackGuard protection is in place to prevent brute-force attacks from happening due to stack manipulation.

**B - Turn on the Non-executable Stack Protection**



By running the command “-z noexecstack” this allows for us to make the stack non-executable. Due to this function our shellcode should no longer work. The stack non-executbale is used as a countermeasure to ensure that the stack is not able to be executed or modified and would only be meant to store appropriate data. Once we compiled our programs with the correct bits and used the “-z noexecstack” command, we see that both programs now cannot be executed and is blocked by a segmentation fault (core dumped). This is the expected output due to us making our stack non-executable.

**Funny Joke Time**



I laughed too hard at this one ^

* I’d tell you another buffer overflow joke,
  + but honestly, it’s a real stack of problems.