Zvi Greenspan

CISC 4900

Fall 2019

Seed Labs progress log

**Buffer Overflow Lab:**

9/24 – set up VM properly, downloaded “Buffer overflow” files. Began to watch video series on Buffer Overflow.

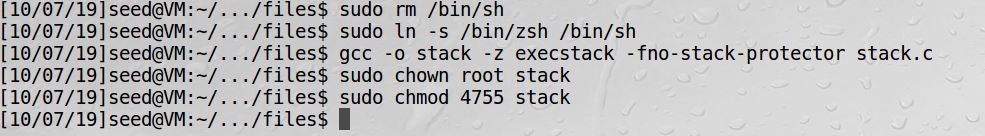
**Pre Task- Turning off counter measures.**

Before a buffer overflow attack can happen, certain counter measures have to be changes or shut off.

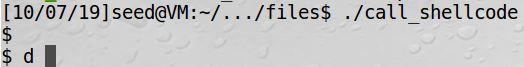
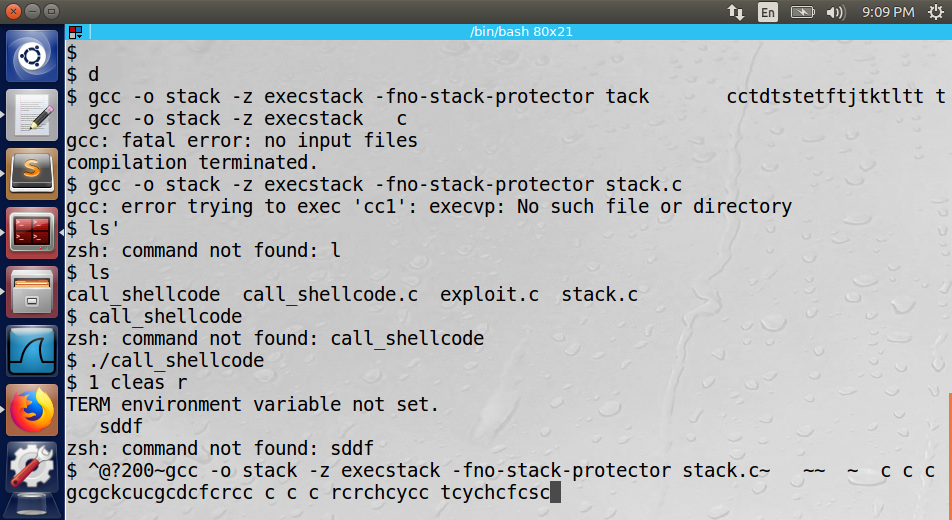
* Change space randomization to 0:



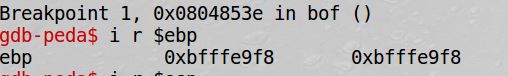
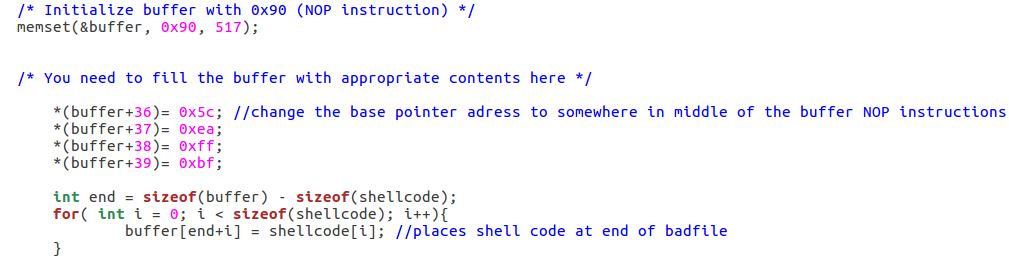
* Linking to different shell that doesn’t have counter measures. And then compiling the stack code with counter measures shut off:

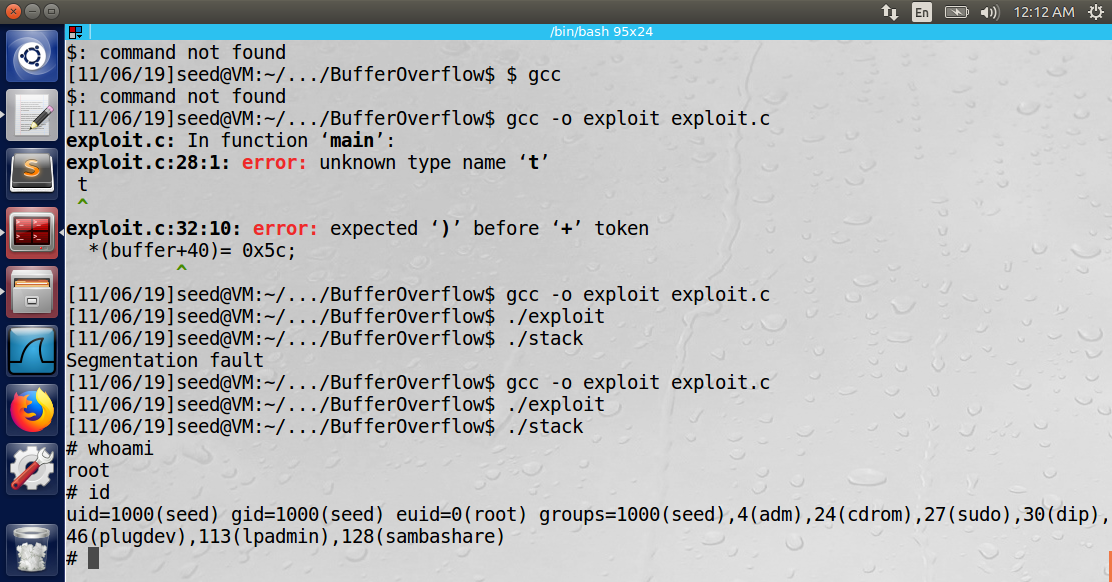


**Task 1 – Running Shell code.**

* Compile with counter measures shut off: 
* Running the shellcode program-
* After running the shell program the system was in shell mode. Without realizing what happened I kept trying to call different cmds in the terminal and kept getting a call back from the ZSH shell program that the cmds are not allowed. 

**Task 2 – Exploiting the vulnerability:**

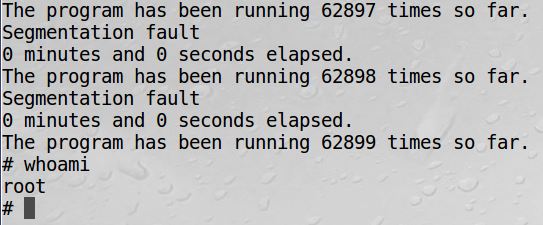
* I used gdb to find the memory address for the stack frame base pointer ($ebp) which was 0xbfffe9f8.
* In the exploit.c file I changed the 4 memory addresses after the buffer which is the return address location in the stack frame to a memory location in middle of the buffers NOP location. I added 100 to the ebp to get 0xbfffea5c memory location. All the NOP operations are placed in a file called badfile that gets reads into the buffer location.
* I then ran a loop to place the shell code at the end of the buffer.
* After compiling and running the exploit.c program and then running the stack program, I was able to get ROOT shell privilege.



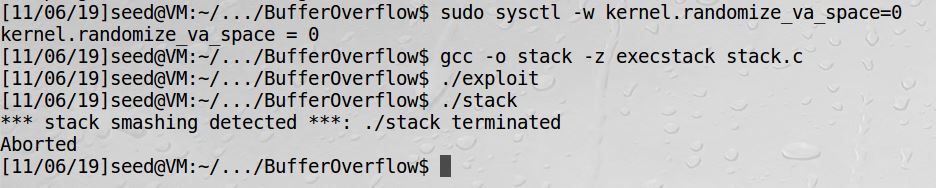
**Task 3 - Defeating dash’s Countermeasure:**

* Added command to the shell code that bypasses dash counter measures to prevent buffer overflow if the effected UID des not equal the real UID. After running the program I still was able to get ROOT shell. 

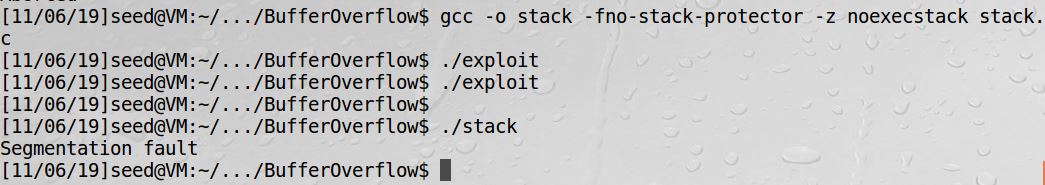
**Task 4 - Defeating Address Randomization**

* With address randomization turned back on, I ran the attack again using brute force trying to eventually get the right location for the shell and I was able to gain ROOT privilege. The
* Program looped for about 15 minutes and ran 62899 times until I was able to get Root Privilege.

**Task 5 -Turn on the StackGuard Protection:**

* In this task I turned address randomization back off, and then recompiled the stack.c program without the StackGuard protection. When I re-executed the attack, I received an error of “stack smashing detected”.

**Task 6 -Turn on the Non-executable Stack Protection:**

* In this lab I recompiled the stack.c program with the “noexecstack option that prevents running shell code on the stack. When I re attempted the attack, I received a Segmentation fault error. 

**Conclusion:**

* I was able to launch a root shell via a buffer overflow vulnerability.
  + I used the debugger to find the memory location of the return address.
  + I used the information to add contents to a bad file to be used in the stack program that has the buffer overflow vulnerability.
* The countermeasures to prevent a buffer overflow where that where effective in preventing the exploit from being carried out are,
  + StackGuard
  + Non-Executable stack.
* The countermeasures that failed to prevent a buffer overflow exploit from being carried out were,
  + Dash counter measures
  + Address Randomization.

I completed the Buffer Overflow lab, in which I spent many hours learning how and why buffer overflows happen and how I can exploit the vulnerability. I spent time learning how the stack frame is set up in a computer’s memory. I had to learn how to use and understand the linux debugger (gdb) to find the memory address of the stack frames. I then used the gdb to find the memory address of the stack frames return address, to know what I need to overwrite the buffer with. Once I had the memory address, I was able to add a few lines of code into a given vulnerable program to change the return address of the program to point to several NOP instructions which ended with shell code (given as part of the lab) that I placed at the end of the buffer. Once that was completed, I was able to execute the buffer overflow attack by running the vulnerable program and I gained Root privilege in the system.

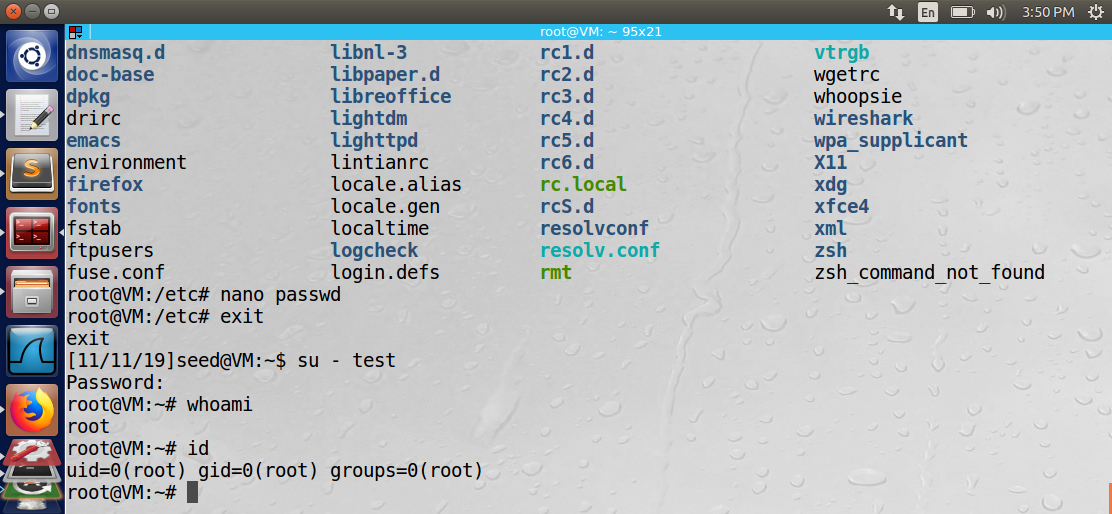
Before I was able to run the attack, I had to turn off the systems (dash) built in counter measure and compile the vulnerable program with other counter measures shut off. I then ran the attack and exploit the buffer overflow vulnerability. After I was able to exploit the vulnerability, I turned each counter measure back on one at a time to reattempt the attack and see which measure is good to protect against buffer overflow. For some protection schemes, I was still able to gain root privilege and for others the counter measure were able to prevent the attack.

**Race Condition Lab:**

**Task 1 Choosing Our Target:**

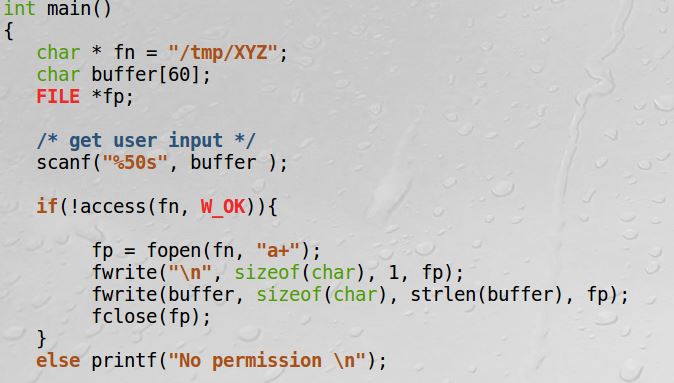
* This Lab is to verify that the target file which is the system password file, with the Ubuntu magic password works to create a root user that does not require a password;
* I added the following line to the password file that has the “magic value” that does not require a password to sign in to test user and the user has root privilege.

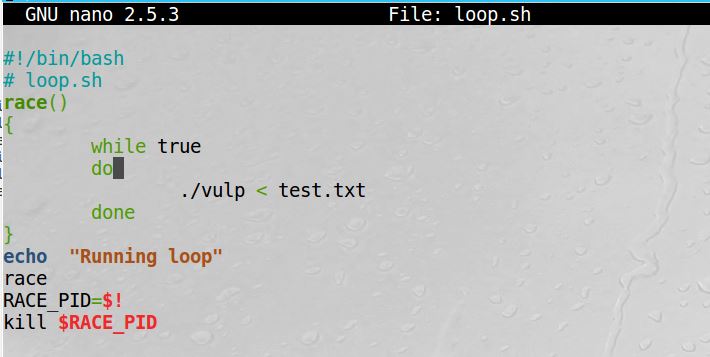
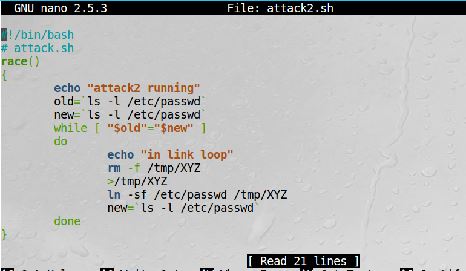
test:U6aMy0wojraho:0:0:test:/root:/bin/bash

****

**Task 2 - Launching the Race Condition Attack**

* The goal of this task is to exploit the race condition vulnerability in the given program. The step needed is make the ‘/tmp/XYZ’ file be symbolically linked to the password file within the time window for the program to check permission on the XYZ file. If the linking is successful, the goal was to have the user input be appended to the password file. This input is a line of text that creates a root user in the system.





* I created the following 2 scripts that when I run them they keep looping to create the symbolic link at the right time so I would be able to win the race condition. The script on the left, checks the time stamp on the passwd file to see if it was changed If it has not been changed run the vulnerable program again. The script on the right keeps looping and creating the symbolic link from the tmp file to the password file.
* However, when I ran the scripts I would get a “no permission” message, meaning I was not able to beat the race condition. I let the loop to run for about an hour, hoping that the linking would take place within the right moment, but I was unsuccessful to exploit the vulnerability,

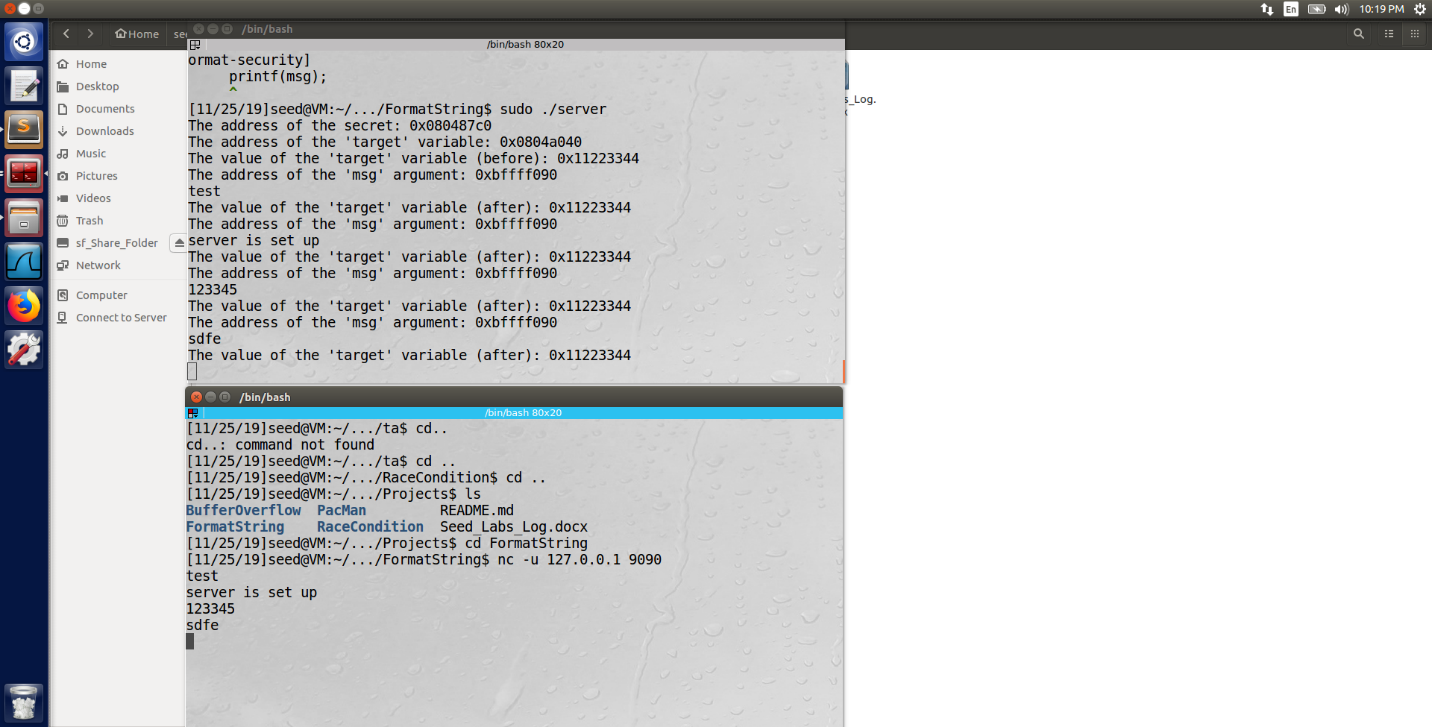
**Conclusion**

* In order to understand how the symbolic links works I had to learn how to create a few files that are links to each other. After modifying the contents of one file I saw how the modified/updated contents appeared in the linked file as well. I also manually modified the password file in the system with root privilege on the command line to manually add a line to the file to create a test user that has root privilege. I then understood what needs to get added to the password file thru exploiting the vulnerability to create a new root user. However, I was unsuccessful in exploiting this vulnerability.

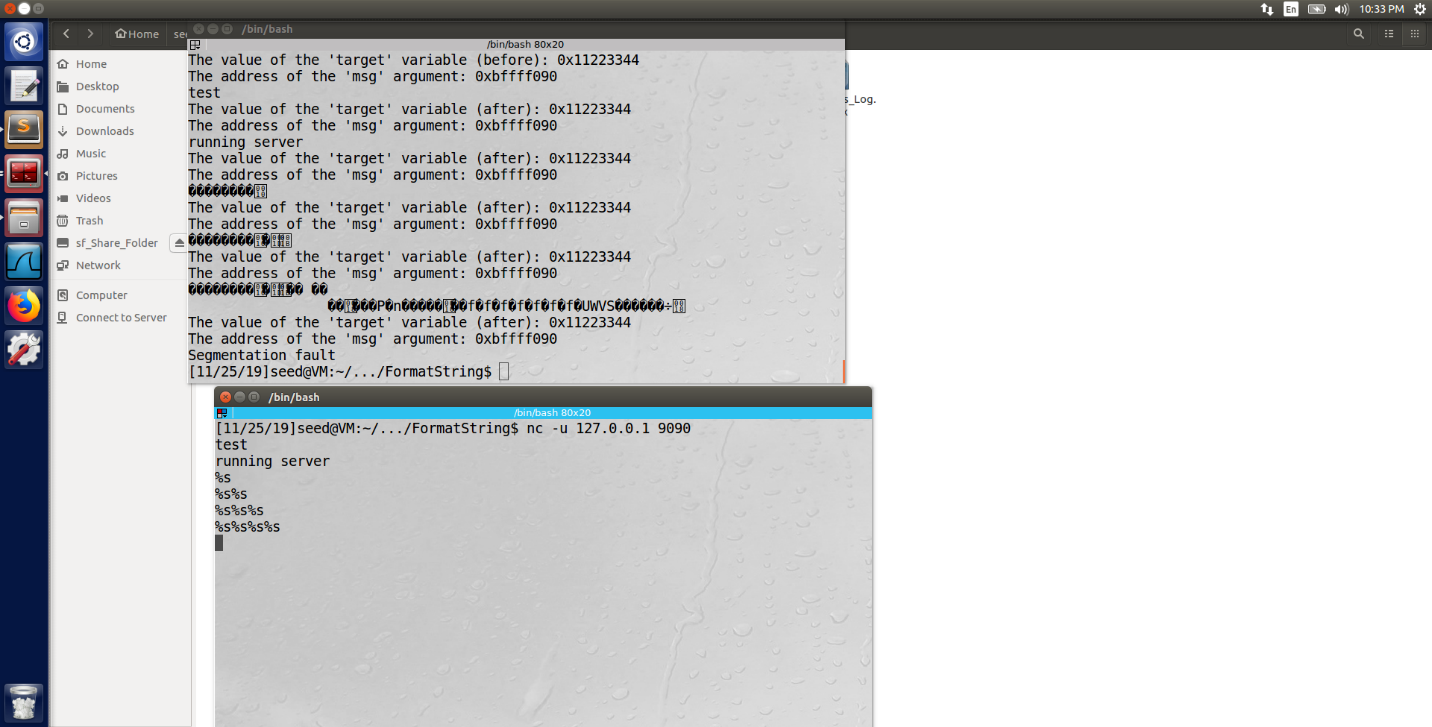
**Format String**

This lab is to exploit the vulnerability that the format string function printf() that uses % characters to fill in data before printing. If the input in not sanitized an attacker can get the program to run arbitrary code. The labs that will be attempted are (1) crash the program, (2) read the internal memory of the program, (3) modify the internal memory of the program, and most severely, (4) inject and execute malicious code using the victim program’s privilege.

**Task 1: The Vulnerable Program.**

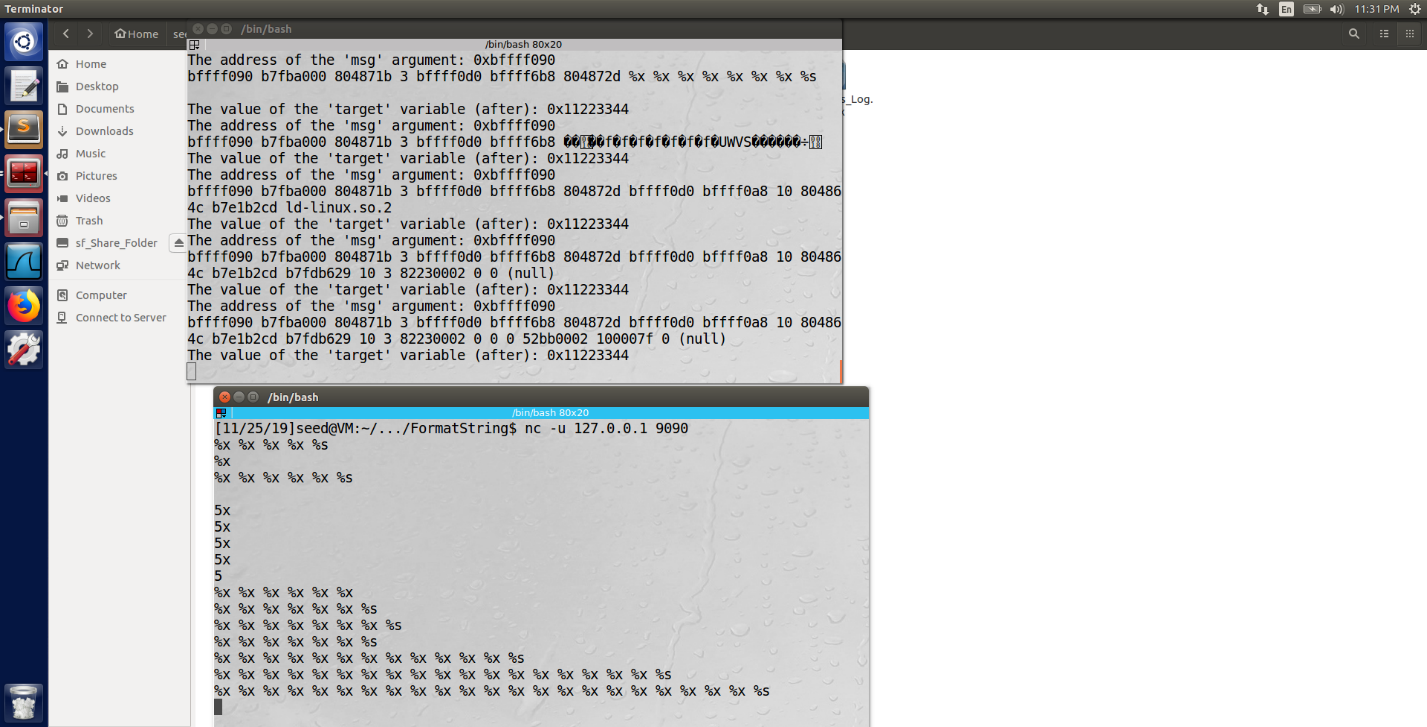
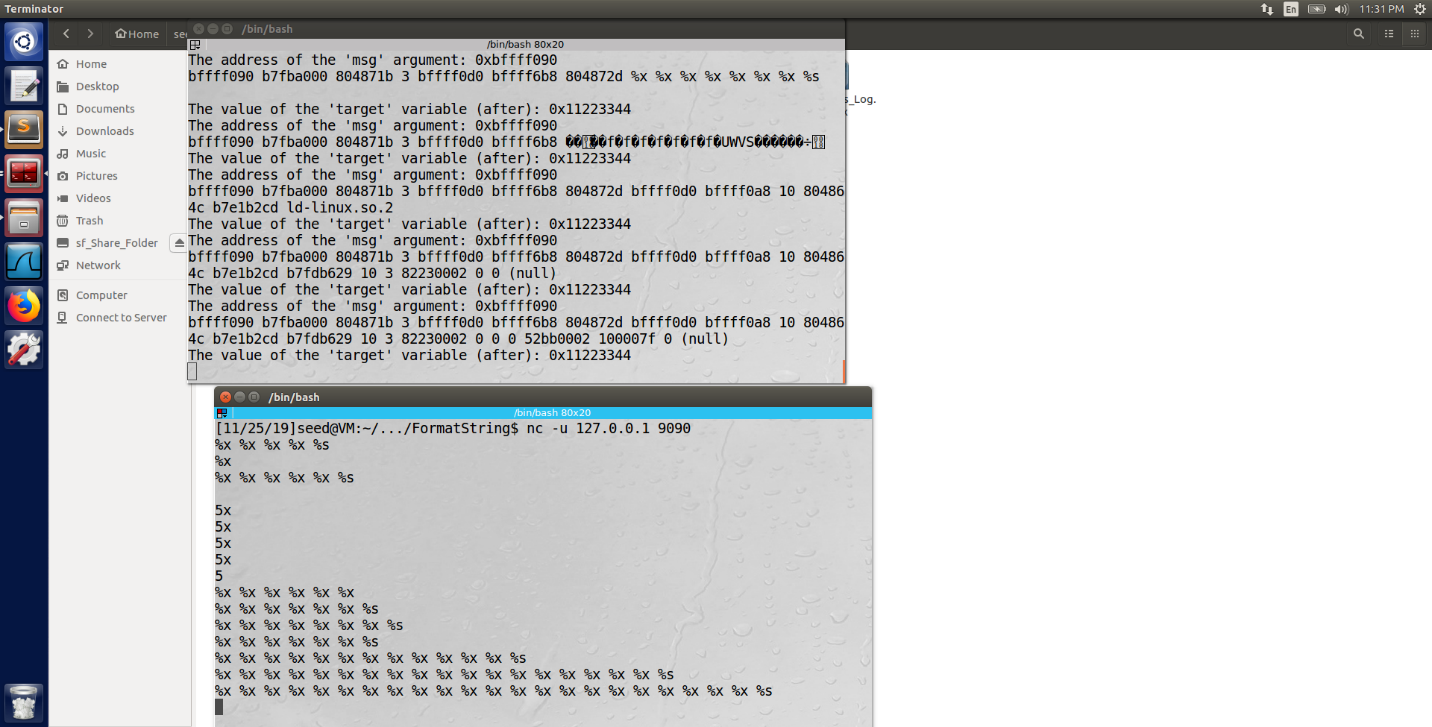
* In this task a vulnerable program is given as a server program that listen on a predefined port for input from a user and prints out that input.
* The task entails compiling the program as a stack executable and then run in and in a different terminal window give input to test that the server works and will print out that input.

**Task 2 Crash the Program**

* The objective of this task is to provide an input to the server, such that when the server program tries to print out the user input in the myprintf() function, it will crash.
* What I did to crash the program was send a couple of “%s” to the server which then tried to print out that value on the stack. Eventually with enough %s being sent, the program crashed.

**Task 3 Print Out the Server Program’s Memory**

* The objective of this task is to get the server to print out some data from its memory. The data will be printed out on the server side, so the attacker cannot see it.

**Stack Data:** The goal is to print out the data on the stack (any data is fine).

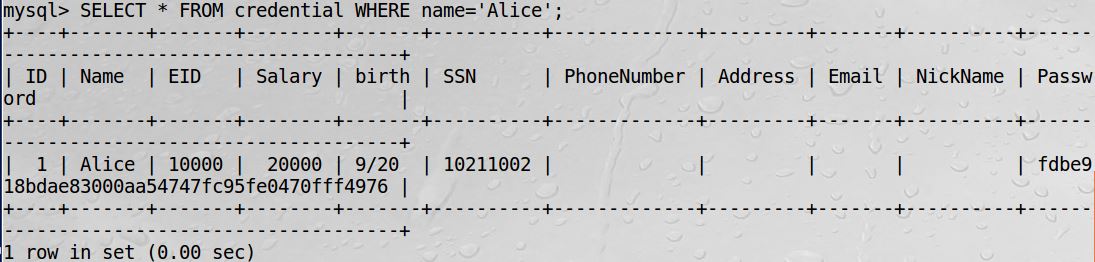
* + **Task 4**
  + **Task 5**

**SQL Injection**

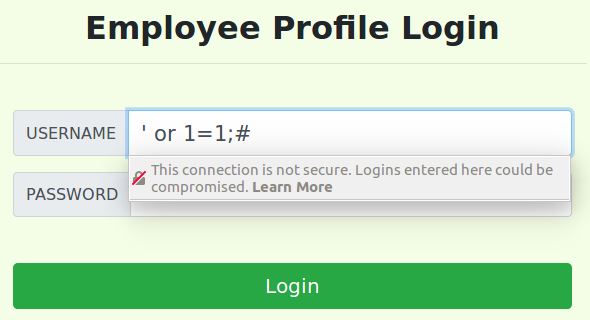
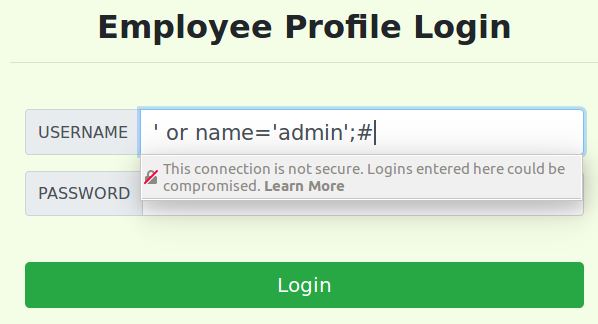
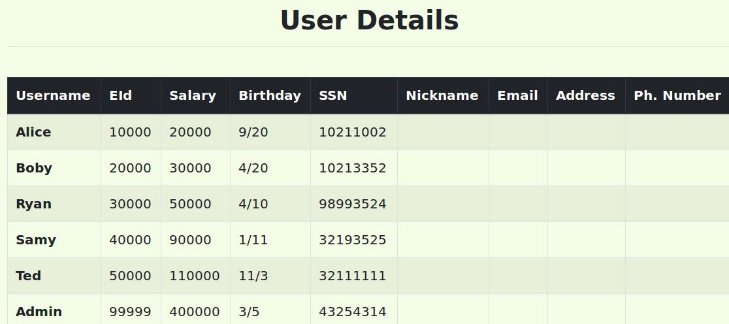
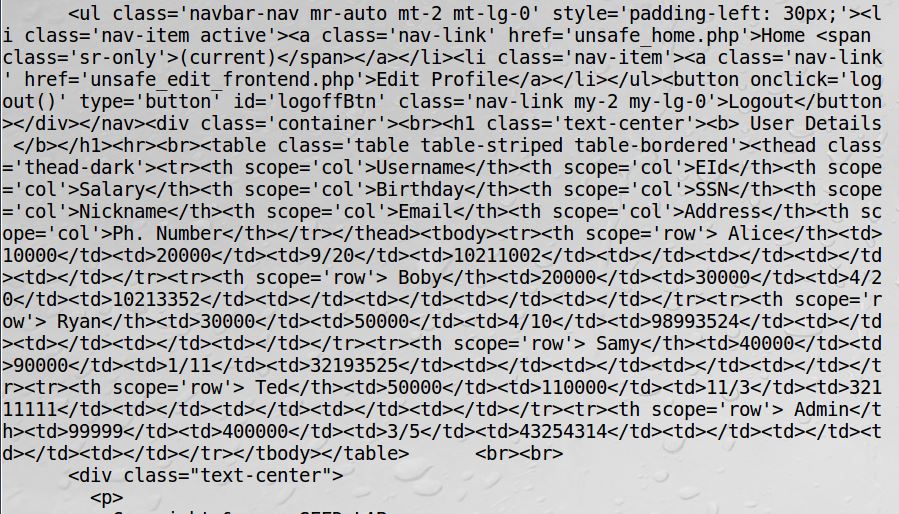
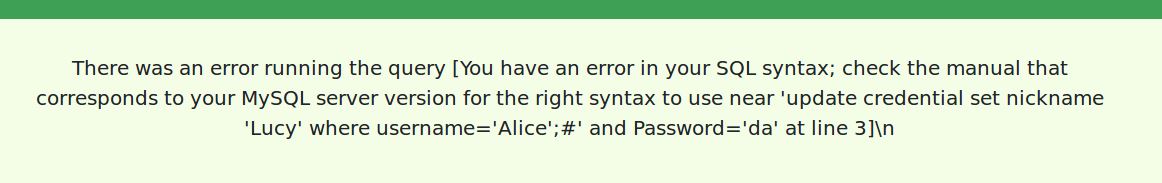
SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is present when user's inputs are not correctly checked within the web applications before being sent to the back-end database servers. Many web applications take inputs from users, and then use these inputs to construct SQL queries, so the web applications can get information from the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When SQL queries are not carefully constructed, SQL injection vulnerabilities can occur. The SQL injection attack is one of the most common attacks on web applications.

In this lab, Seed Labs created a web application that is vulnerable to the SQL injection attack. The web application includes the common mistakes made by many web developers. The goal is to find ways to exploit the SQL injection vulnerabilities, demonstrate the damage that can be achieved by the attack, and master the techniques that can help defend against such type of attacks.

**Task 1**

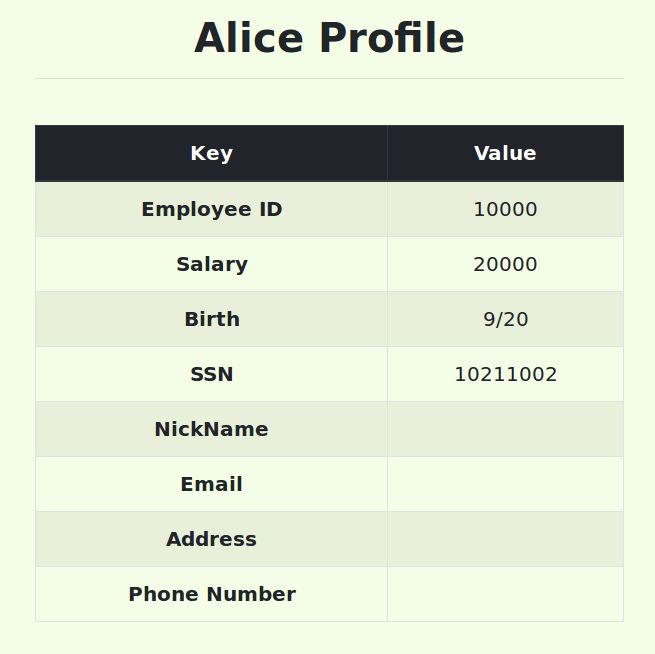
* The objective of this task is to get familiar with SQL commands by playing with the provided database. We are given a created database called Users, which contains a table called credential; the table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee. In this task, we need to play with the database to get familiar with SQL queries.
* ****The task is to use a SQL command to print the profile information of the employee “Alice”.

**Task 2: SQL Injection Attack on SELECT Statement:**

* **Task 2.1: SQL Injection Attack from webpage:**
* In this task we are given a vulnerable website to SQL injection attacks we are trying to exploit that by logging in as an admin. We know that there exists an administrator account ‘admin’. We need to inject code to be able to log in as an admin without knowing the password of the admin.
* ****I first injected the statement [‘ or 1=1;#] into the login page which is a true statement that tricks the database to authenticate and return the information of the first user in the database who was Alice.
* ****I then wanted to exploit the vulnerability and log in as admin to see the inforamtion of all the users in the database. I inject the following code [‘ or name=’admin’;#] which also send a true statement to the database that the name should be “admin” which I then get authenticated as an admin and the database returns the information on all the employees.
* Explanation of the injection statement: The singe quotes closes the argument for the input id, the OR statement allows us to login as admin. The # is inserted are the end to comment out everything else that follows the input is skipped.
* **Task 2.2: SQL Injection Attack from command line.**
* In this task we are to run the same attack as before using CURL command from the command line. The attack is successful again as you can see the print out of the html that contains all the employees information.
* ****The curl command is below, I had to change any special characters such as space or single quote with its URL encoding value.
* Below is the return I got after running the SQL injection from the command line and you can see all the employee information within the tags.
* **Task 2.3: Append a new SQL statement.**
* This task is to attempt to modify the database by using the SQL injection and turn one SQL statement into two with the second one being the update or delete statement. The two statements need to be separated by a ‘;’ for the attack to work.
* The statement that I attempted to Inject is …' or 1=1; update credential set nickname 'Lucy' where username = 'Alice';#
* However, this statement returns an error because there is a countermeasure in MySql that prevents multiple statements from executing when invoked from php.

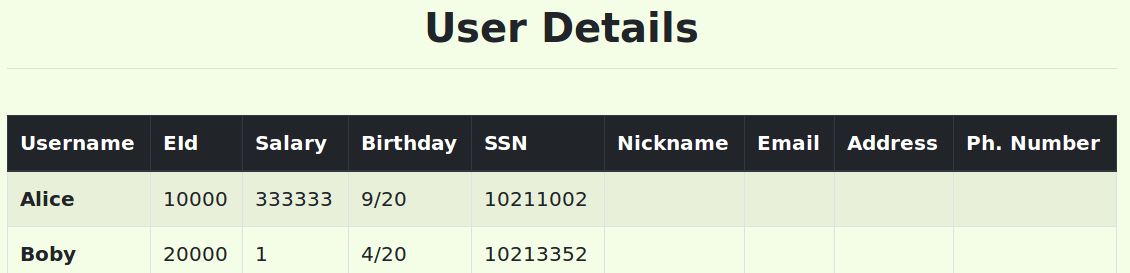
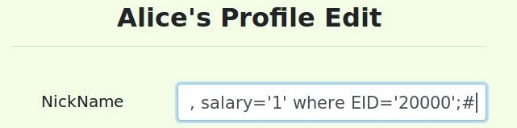
**Task 3- SQL Injection Attack on UPDATE Statement:**

* **Task 3.1: Modify your own salary:**
  + In this task we need update the salary of the signed-in user via their Edit Profile page. The user can change any of their information except their own salary, which only an admin can do. We need to change the salary by exploiting the SQL injection vulnerability in the Edit Profile page.
  + To achieve this goal I injected the code [', salary='333333' where EID='10000';#] into the first edit field in the Edit Profile page which change the salary for Alice to 333333.



* + Explanation of the injection statement:

What the injected code does is the first ‘ closes the statement the “,” separates the statements and then salary is the field we need to change and we know that it exists with the EID given. The # is to comment out all other values that follow so we wont have any issues with them.

* **Task 3.2: Modify other people’ salary.**
  +  This task is to change the salary of a different employee (the boss) to 1 using a similar injection as above. I used [', salary='1' where EID='20000';#]. I just changed the EID for the other employee and the salary to 1. You can see from the screenshots how the SQL injection made a change in the database.