Daniel Oliveros

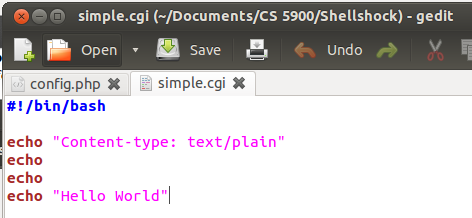
Garrett Bogart

Spring 2018 – Independent Study

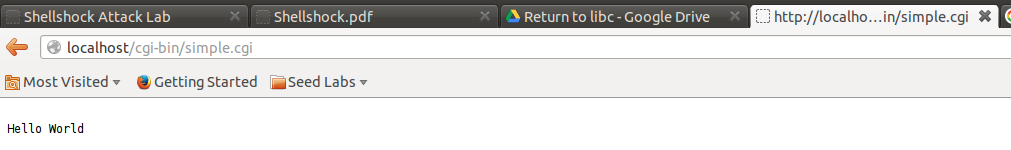
Shellshock Vulnerability Lab

**Task 1**

Setting up the CGI program in the /usr/lib/cgi-bin directory. To create a file in this directory you need to be a root user.

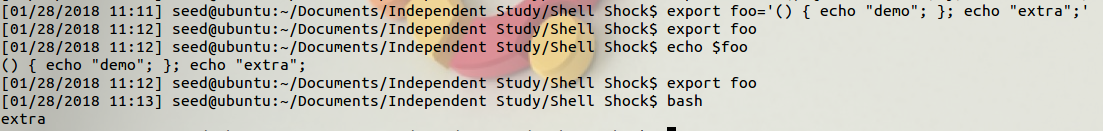


Looking at the file on our web browser we can see that hello world has been displayed. Generally you would have to use an ip address but since we have a local server running we can use localhost instead.

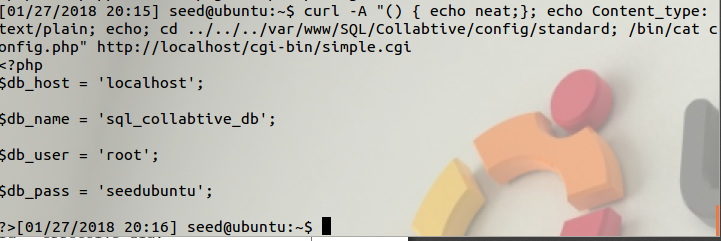


Using the command line we can use curl to look at the webpage.

The shellshock vulnerability takes advantage of how a parent process can pass a function definition to a child process. When bash converts the environment variable it should only parse the value not execute it. There is a bug in the parsing logic that allows for some of the environment variable to be executed. This creates the shellshock vulnerability.



In this example an environment function foo is declared. After we bash we can see that extra is echoed to our console. We are not limited to only echoing things to the console. We can use normal command line functions as well. To show the vulnerability we navigated through the directories of the machine. We then accessed a file that had user information and displayed it to the console.



This code snippet checks to see if there is an exported function. It does so by checking to see if the value of the environment variable starts with “() {“.



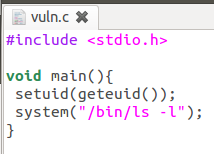
Parse\_and\_execute is a generalized function that can parse other shell commands; as well as, function definitions. Functions get parsed and shell commands will be executed by the function.



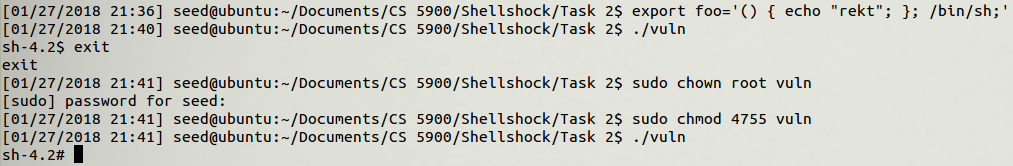
**Task 2**

**2A:**

Here is the vulnerable setuid program. It uses the system call to execute the command we want it to.

****

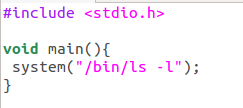
This image shows the execution of the program. By exporting a variable that attempts to exploit the Shellshock vulnerability before the program’s execution (and making it so the program is owned by root and is a setuid program program) we can get a root shell from its execution.

****

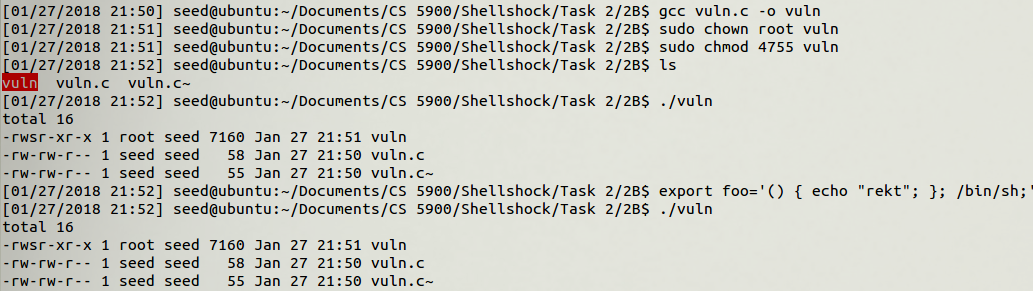
The reason why this happens is because the system() call goes through all environment variables, when it reaches foo, it has finished describing it as a variable and proceeds to execute the shell command we appended to it, in this case /bin/sh.

**2B**

After removing the setuid line, this is our program:

****

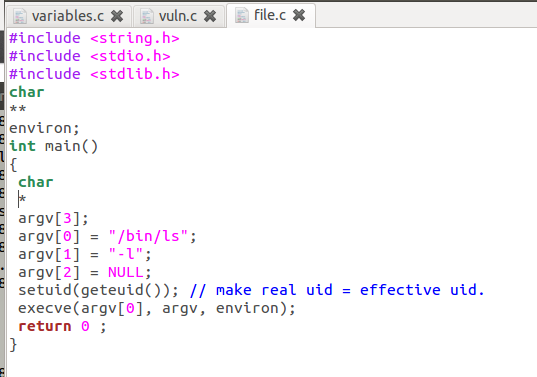
Once executed, the program still only performs the functionalities found within it, it does not accidentally run the /bin/sh command appended to the end of the variable foo

****

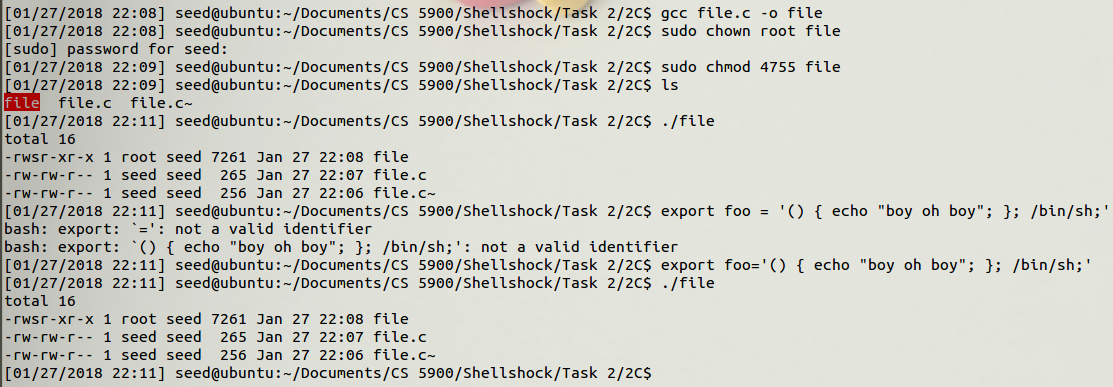
The reason for this working differently than **2A** is that the setuid line of code allows for an interesting interaction to occur alongside the system call. When system is called, the program looks through all the environment variables that are owned by the user whose ID matches that of the program. Since the uid for the export variable is the one for the seed user and the uid related to the program is linked to root, then foo is never read or executed, and the vulnerability is therefore unexploitable.

**2C**

This is the program we ran this time around:



When executed using the same method we followed for task **2A**, the program also did not give us a root shell. The reason for this is that the way this piece of code works is intrinsically different than the one given in task **2A**. The most important difference, namely, is that it’s using execve to execute the program, which, unlike system(), does not run the script given to it by running it on an instance of the /bin/sh command. It, instead, runs the command by looking for the executable file stored in the locations argv[0] and gives that binary file the contents of argv[1] and argv[2] as parameters. This also means that the program does not read through environment variables and end up meeting our malicious foo variable.

****

**Task 3:**

1. Any other scenario in which a system() call is invoked within a setuid program. As long as we can export a malicious variable to the system beforehand we will be able to execute the script we desire.
2. The fundamental problem of this vulnerability is that parsing was performed without enough restraint on what the user would give it. Another main issue was that it laid undetected for decades, which meant a great deal of our systems were built around this vulnerability, not knowing how exploitable this could be.

**Issues**

1. The commands for the shellshock attack have to be precise otherwise things are not parsed properly. When these don’t work, especially in the .cgi attack, there will be no feedback as to how the command is not well-structured.

**Ideas for improvement**

1. The background is lacking in how the function calls should be structured when trying to exploit the vulnerability both in a cgi file, and in a setuid program.
2. Give an explanation about curl and how to pass arguments through it