**Shellshock Attack Lab Report**

**Task 1:**

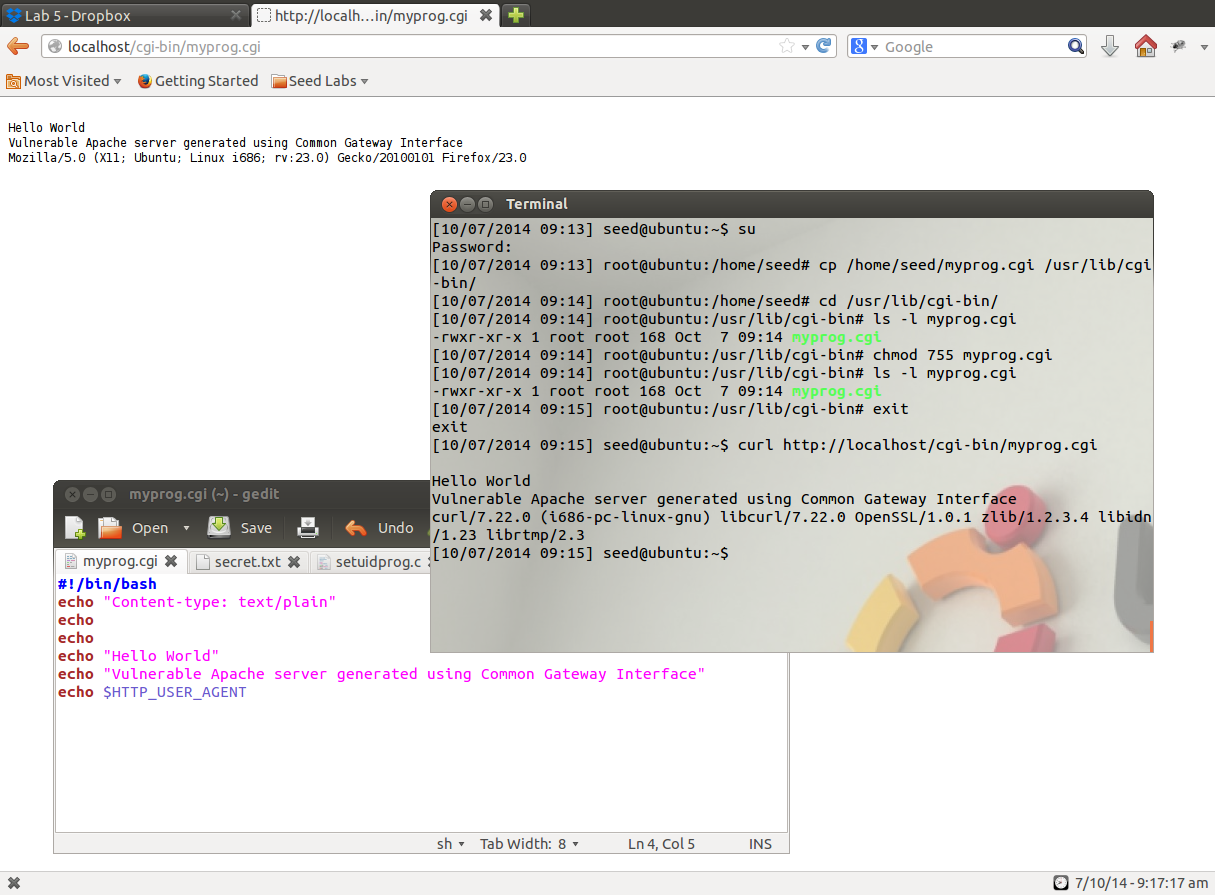
****

Figure 1.1

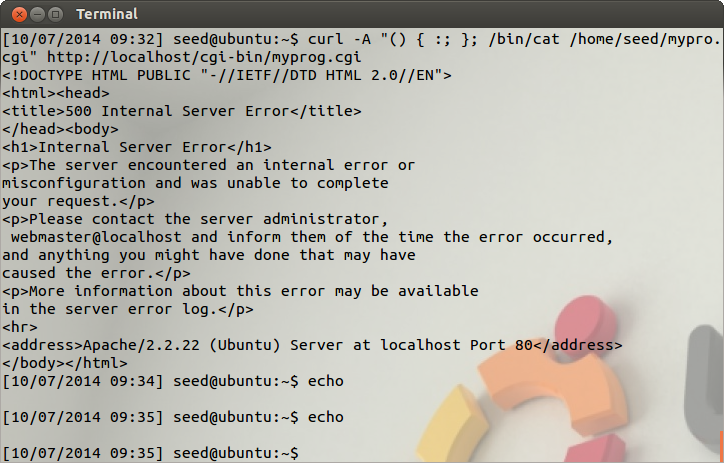
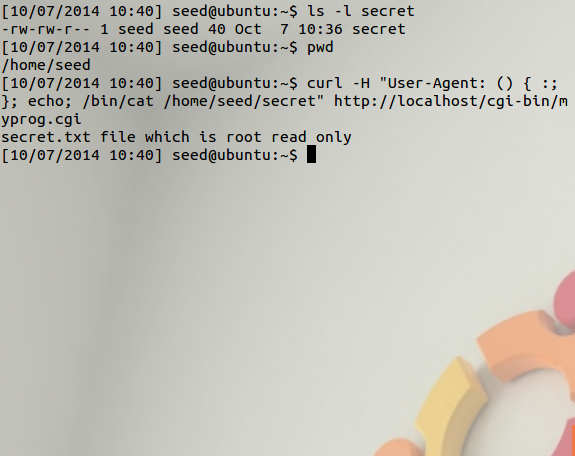


Figure 1.2



Attack successful the file is printed to shell.

Figure 1.3

**Observations and Explanation:**

1. Set-up of Common Gateway Interface (CGI) program, we use curl to output the result of the file built using CGI scripting.

We make the .cgi file executable and place it in the /usr/lib/cgi-bin folder.

1. In this step we launch the attack on the file and are able to fetch a file that is not accessible to attacker and display the contents of that file.

We use the following command to be able to do this

**curl -H "User-Agent: () { :; }; echo; /bin/cat /home/seed/secret"** [**http://localhost/cgi-bin/myprog.cgi**](http://localhost/cgi-bin/myprog.cgi)**,**

**The –H allows us to add a user-agent, the parser processes the string and also performs the command program**

1. The problem is in variables.c file in the ***initialize\_shell\_variables(env, privmode)*** function.

***if (privmode == 0 && read\_but\_dont\_execute == 0 && STREQN ("() {", string, 4))***

***{***

***string\_length = strlen (string);***

***temp\_string = (char \*)xmalloc (3 + string\_length + char\_index);***

Check for privilege, because of this if the privilege mode is not root, the attack fails.

***strcpy (temp\_string, name);***

***temp\_string[char\_index] = ' ';***

***strcpy (temp\_string + char\_index + 1, string);***

If an environment variable starts with a string such as ‘() {’ then **initialize\_shell\_variables** recognizes it as a function definition.

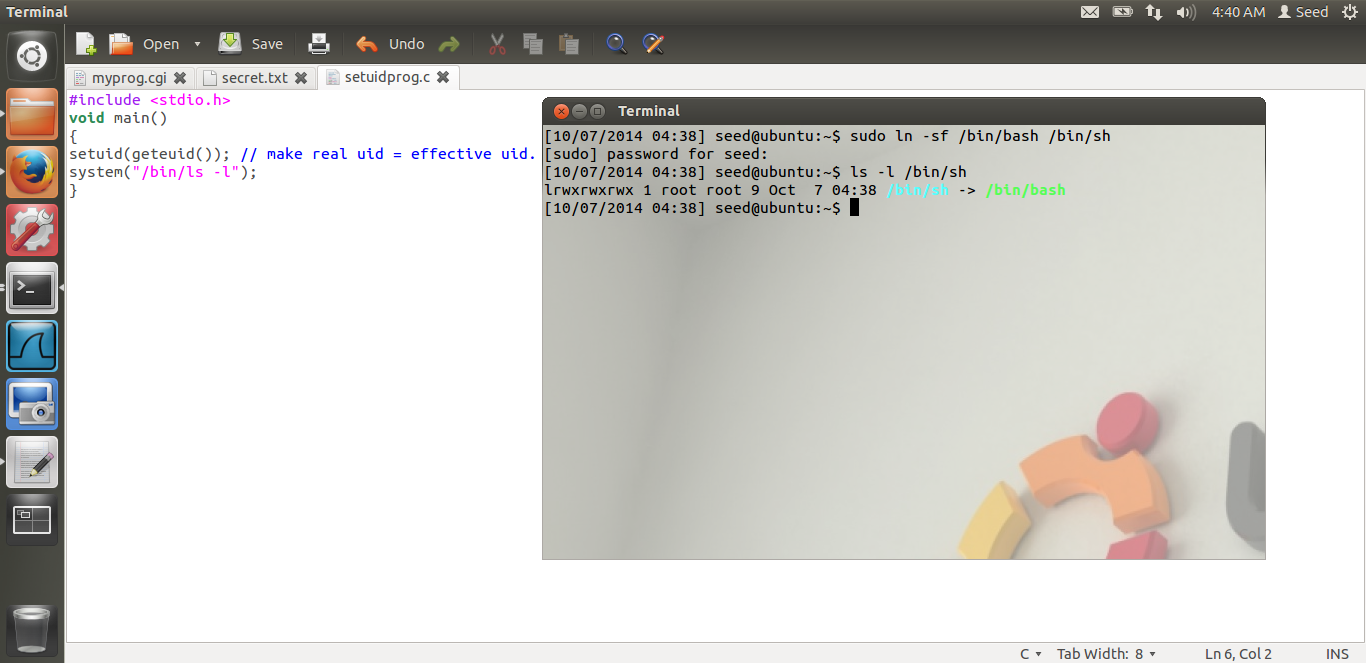
***parse\_and\_execute (temp\_string, name, SEVAL\_NONINT|SEVAL\_NOHIST);***

The rest of the string after ‘() {’ is passed to parse and execute.

It is assumed that the rest of the string contains only the function execution and so it is passed into the function without providing the checks.

This is still only a minor problem, the actual problem is that ***parse\_and\_execute()*** does not stop executing after the end of the function. Bash, executes all the commands in the string, even after the function definition is finished. So, if an attacker can control an environment variable in a program that will spawn a shell with an environment containing that variable, command injection is possible.

**Task 2:**

****

Linking bash to sh

Figure 2.1

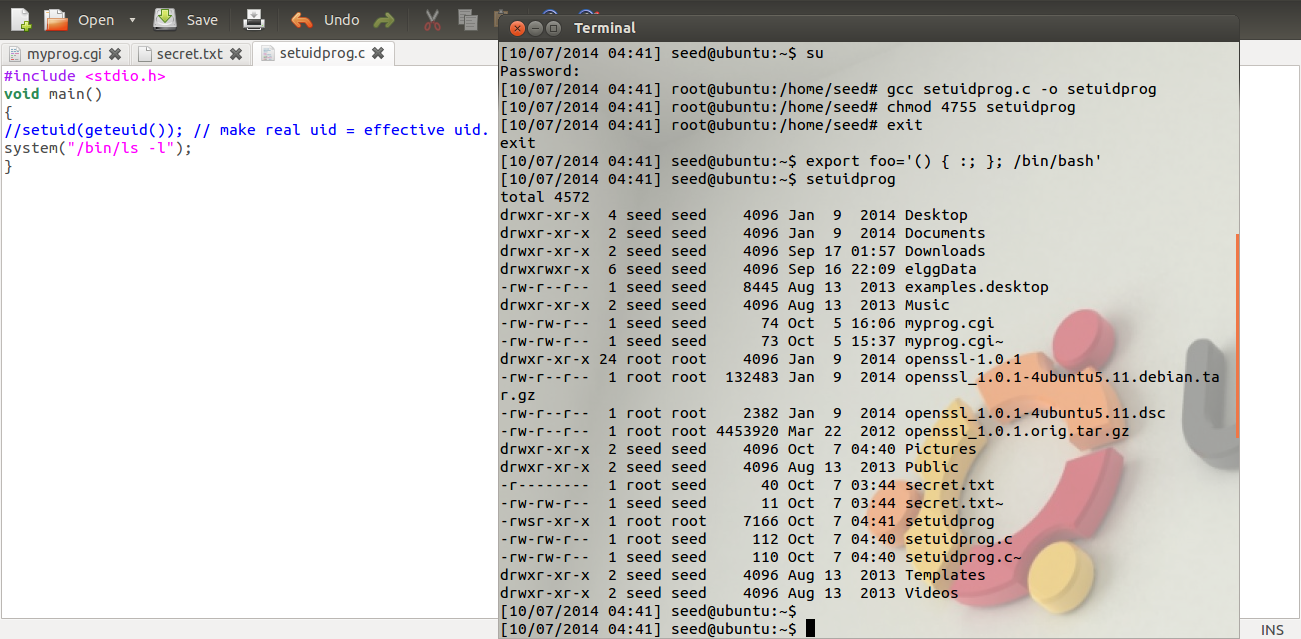


Setting up the vulnerability

Shellshock successful. Root access gained

Root owned setuid program

Figure 2.2



No root access. Attack fails

Setuid commented out

Figure 2.3

**Observations and Explanations:**

1. Figure 2.1 and 2.2 provide basic setup for this task.

In figure 2.2 we compile *setuidprog.c* and give it root permissions.

***void main()***

***{***

***setuid(geteuid()); // make real uid = effective uid.***

***system("/bin/ls -l");***

***}***

We export the vulnerability causing sequence, the injection causes the inadvertent execution of commands at the end of the function definitions, stored in the values of environment variables.

1. On removing the,

***setuid(geteuid()); // make real uid = effective uid.***

Statement from the program(Figure 2.3), we are unable to gain root access, the reason for this is that, if the real user id and the effective user id are the same, the function defined in the environment variable is evaluated, and so the bash door vulnerability can be exploited.

But since here, the real uid and effective uid are not the same, the attack fails.

1. The problem is in variables.c file in the ***initialize\_shell\_variables(env, privmode)*** function.

***if (privmode == 0 && read\_but\_dont\_execute == 0 && STREQN ("() {", string, 4))***

***{***

***string\_length = strlen (string);***

***temp\_string = (char \*)xmalloc (3 + string\_length + char\_index);***

***strcpy (temp\_string, name);***

***temp\_string[char\_index] = ' ';***

***strcpy (temp\_string + char\_index + 1, string);***

***.***

***.***

***.***

***}***

If an environment variable starts with a string such as ‘() {’ then **initialize\_shell\_variables** recognizes it as a function definition.

***parse\_and\_execute (temp\_string, name, SEVAL\_NONINT|SEVAL\_NOHIST);***

The rest of the string after ‘() {’ is passed to parse and execute.

It is assumed that the rest of the string contains only the function execution and so it is passed into the function without providing the checks.

This is still only a minor problem, the actual problem is that ***parse\_and\_execute()*** does not stop executing after the end of the function. Bash, executes all the commands in the string, even after the function definition is finished. So, if an attacker can control an environment variable in a program that will spawn a shell with an environment containing that variable, command injection is possible.

**Task 3:**

1. **Other than the two scenarios described above (CGI and Set-UID program)**

There are a few other scenarios where Shellshock can be used to exploit a system,

1. SSH server- OpenSSH uses ForceCommand feature, where a fixed command is executed instead of running an unrestricted shell, even if the user specifies a special shell program to be run. This command if put into the environment variable “SSH\_ORIGINAL\_COMMAND”. When the forced command is run as Bash shell, the bash shell will parse this environment variable on startup, and run the commands embedded in it. The user has used their restricted shell access to gain unrestricted shell access, using the Shellshock bug.
2. IBM restricted shell- The IBM restricted shell can be overcome and the user can gain access to unrestricted shell.
3. DHCP- DHCP can pass commands to Bash, when connected to an open Wi-Fi network. A DHCP client can request and fetch more than the IP address from a DHCP server, but can also be provided a series of additional options. Strings can be sent to local machines for execution.
4. Email systems- Depending on the system configuration, a qmail mail server can pass external input through to bash, that will enable exploitation of a vulnerable version of qmail.
5. On **6th October,** it was reported that **Yahoo! Servers** were compromised as a result of a variation of the shellshock attack.
6. **What is the fundamental problem of the Shellshock vulnerability? What can we learn from this vulnerability?**

Basically shellshock lets an attacker execute arbitrary code on web servers.

Bash lets users define functions as a way to pass text on to other systems and processes. The problem with this vulnerability, which includes specific characters as part of a definition

**() { :; };**

is that bash doesn’t stop processing the string containing the function after it is defined, it will continue to read and execute shell commands following the function definition.

This will allow the attacker to get access to the shell. This doesn’t mean that the attacker has gotten root. It simply allows the attacker to continue the attack and attempt privilege escalation, which might lead to access to root.

This vulnerability brings to light the fact that sanitation of strings being sent to a shell needs to be done.

The problem is that parser has a vulnerability where if there is code after the end of the function declaration, that code will be immediately executed.