ISEC3004

Cyber Crime and Security Enhanced Programming

Assignment

Student Name: Syed Faraz Abrar

Student ID: 19126296

TABLE OF CONTENTS

Setting up the website

Follow these steps to set up the web application from a fresh instance of the provided VM:

1. Move the assignment tar.gz file to the VM and extract its contents to the home directory of the student user.
2. Remove all existing files in the /var/www/html folder, then move all the extracted contents of the folder containing the index.php file into /var/www/html.
3. Go to /var/www, then run the following command, typing in the password as CCSEP2019 when prompted: **sudo chown -R student:www-data ./html**
4. Change directory to /var/www/html, and run the following command, typing in the password as CCSEP2019 when prompted: **mysql -u root -p < prepare_db.sql**

The website is now ready to use, test, and view. Throughout this document, I will be referring to the website as “assignment.com”.

Overview of the web application

The assignment brief required the creation of a web application that behaves somewhat like an e-store. It should have two types of users: regular users (customers) and admin users.

Regular users should be able to add funds to their accounts, put items up for purchasing, purchase items using their funds, and have some form of functionality to search the store for items put up by other users. They should be able to search for items both by item name and by seller name.

Admin users should be able to do all of the above, but they also need to be able to lock and unlock regular user accounts as well as remove items that are up for sale on the store in case the item is not deemed to be appropriate to be up for sale on such a store.

There are four users created by default. All of them have their passwords set to “password” without quotes. These are the following users: **admin**, **locked_user**, **normal_user**, **delete_test_user**.

There are two items created by default that belong to **normal_user**. These items are “**Orange**” and “**My Grandma’s knitting tools**”.

The Directory Structure

The directory structure for my web application is as follows:

1. The root directory (**/**) contains all the php files that should be visible to the user for interaction.
2. The **/includes** directory contains mostly PHP scripts that handle user interaction with the web application (such as GET and POST requests to the various pages in the root directory). It also contains the navigation bar, and some other PHP scripts that deal with various parts of the web application.
3. The **/fonts** directory contains all the files required for FontAwesome fonts.
4. The **/js** directory contains all the JavaScript files required for jQuery and Bootstrap.
5. The **/css** directory contains the CSS files required for Bootstrap and FontAwesome.
6. The **/images** directory contains any images used in the web application.

Assumptions made

I have made two assumptions for the web application that is not a requirement stated in the assignment brief. The first is that no one should be able to access the main functionalities of the web application (i.e the e-store) without being logged in as a valid user. Due to this assumption, any attempts to access any part of the application without having a successfully established session will result in a redirect to the login page.

The second assumption is that there will be only one admin account with an account id of 1.

Brief overview of each file

I will now provide a brief description of each php file.

- **index.php** – This is the front page of the web application.
- **account.php** – This is the user's profile page. They can view their account ID, email, and current balance here. They may also add balance to their account from here.
- **your_items.php** – This is where a user can view the items that they've put up on the store. They are given the functionality to search for their own items by name, as well as view the item's name, description, and price.
- **store.php** – This is the store page. Users may view item names, descriptions, and prices, as well as purchase items put up by other users (but not their own) from here. Administrators have a special button to allow them to remove items put up on the store should they require. There is also functionality to search for items by name or by seller name.
- **add_item.php** – This is where users may put up their own items for sale. They must provide an item name, an item description, and a price for the item.
- **logout.php** – If the user wishes to logout, they are redirected here.
- **admin.php** – Only viewable by administrative users. There is functionality to allow admins to search for users by name. They can view the account ID, username, and email address of each user. They can also lock, unlock, and delete regular user accounts (but not the admin account).
- **auth.php** – This page is where the users are redirected should they try to access any other page without having a valid session. This page will include either **login.php** or **register.php** by a GET parameter, depending on whether the user chooses to login or register an account.
- **login.php** – This is the login page. Users must provide a username and password to login. If a user tries to access this page directly, they are instead redirected to **auth.php** with the login page included.
- **register.php** – This is the registration page. Users can provide an email, username, and password to create an account. If an account already exists with the same username or email provided by the user, then they are given an error message to indicate as such. If a user tries to access this page directly, they are instead redirected to **auth.php** with the registration page included.
- **db.php** – This is a helper page. The user isn't supposed to be able to view this page. It is included in most other pages. It only contains code to establish a connection to the database, and stores the connection in a global variable called **$db**.
- **/includes/check_session.php** – This page is included in almost every other page. It checks to make sure that a user has a session established. If they don't, they are immediately redirected to **auth.php?page=login.php**.

- **/includes/handle_add_item.php** – This page is included in **add_item.php**. It handles any POST requests made to **add_item.php**, meaning it essentially handles adding items to the store by a user.
- **/includes/handle_admin.php** – This page is included in **admin.php**. It handles any POST requests made to **admin.php**, meaning it essentially handles locking, unlocking, and deleting user accounts, as well as prevents the aforementioned actions from being performed on admin accounts.
- **/includes/handle_balance.php** – This page is included into **account.php**. It handles any POST requests made to **account.php**, meaning it essentially handles adding balance to a user's account.
- **/includes/handle_login.php** – This page is included into **login.php**. It handles any POST requests made to **login.php**, meaning it essentially handles logging a user in and creating a session for them, followed by redirecting them to **index.php**. If a user already has a session established, they are instead redirected to **index.php**.
- **/includes/handle_logout.php** – This page is included into **logout.php**. It handles logging the user out by just destroying the user's current session and then redirecting them to **auth.php?page=login.php**.
- **/includes/handle_register.php** – This page is included into **register.php**. It handles any POST requests made to **register.php**, meaning it essentially handles creating an account for a user, followed by redirecting them to **auth.php?page=login.php**. It also sanity checks to make sure a user isn't trying to create an account with a duplicate username or email. If a user already has a session established, they are instead redirected to **index.php**.
- **/includes/handle_store.php** – This page is included into **store.php**. It handles any POST requests made to **store.php**, meaning it essentially handles any admin's request to delete an item, as well as any user's requests to purchase an item. It also sanity checks to make sure a user isn't buying their own item.
- **/includes/header.php** – This page is included in almost every other page. It includes the html code for the navigation bar. It also includes a hidden button for the admin panel that is only viewable to administrative users.
- **/includes/includes.php** – This page is included in most other pages. It simply includes the <link>, <title>, and <script> tags of the code that is included into most of the other pages' <head> sections.

Beside the above files, there is also a **prepare_db.sql** file that is used to set up the database. The usage of this file has been documented in the **Setting up the website** section of this document.

Vulnerabilities

This section will detail any vulnerabilities that exist in the web application as required by the assignment brief. For each vulnerability, I will state the type of the vulnerability, a description of the vulnerability, the location in the specific bit of code that has this vulnerability, a description on how to trigger the vulnerability, and finally a description of how to modify the program to mitigate the vulnerability.

Reflected XSS

Description

A cross site scripting (XSS) vulnerability is where a user provided input (such as a username, or a search parameter, or etc) is then output back to the user on a web page verbatim within a specific without any sanitization having been performed.

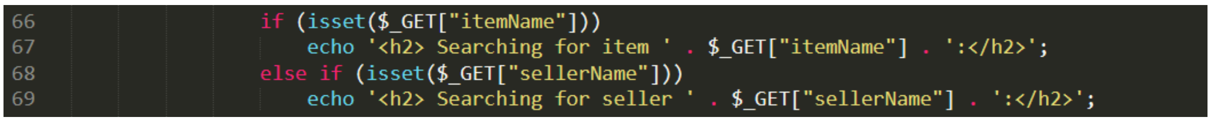
A reflected XSS vulnerability is where this user provided input is reflected in a temporary message, such as a “Searching for <username>” search result, or an error message such as “An error occurred when looking for <itemname>”. In these cases, a malicious user can inject their own JavaScript, such as “**<script>alert(document.domain)</script>”**, which will then get injected into the page.

This can be used to perform a malicious attack onto other users, such as to steal their session cookies or redirect them to some other phishing website that looks exactly like the one they are currently on, but actually stores their credentials for future use by a malicious actor.

This is also slightly less danger than its sister variant (stored XSS) due to requiring user interaction to occur in the first place. The user is asked to usually click a link (or etc) which leads them to a page that has the reflected XSS attack

Location in code

There exists two reflected XSS vulnerabilities in this web application. They are both in the **store.php**, within 66-69, where a user’s search parameter is reflected directly back onto the page without being sanitized.



```
66                  if (isset($_GET["itemName"]))
67                      echo '<h2> Searching for item ' . $_GET["itemName"] . ':</h2>';
68                  else if (isset($_GET["sellerName"]))
69                      echo '<h2> Searching for seller ' . $_GET["sellerName"] . ':</h2>';
```

Trigger

In order to trigger the vulnerability, simply search for an item by name or by a seller’s name, and type in the following as the search parameter:
<script>alert(document.domain)</script>

For a more realistic trigger, simply make a user visit either of the following links:
http://assignment.com/store.php?itemName=<script>alert(document.domain)<%2Fscript>
http://assignment.com/store.php?sellerName=<script>alert(document.domain)<%2Fscript>

Mitigation / Removal

To be completely honest, XSS is very hard to mitigate. However, if it was me, in this specific case I would mitigate it by using an inbuilt function that PHP already comes with that sanitizes user input and prepares it to be put into HTML documents, called **htmlspecialchars**. This function is (to the best of my knowledge) sufficient for this case in sanitizing user input to mitigate XSS.

Stored XSS

Description

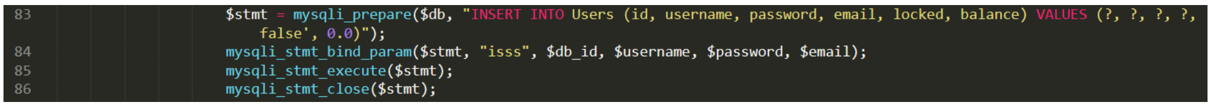
Cross Site Scripting (XSS) vulnerabilities have been described above. A stored XSS vulnerability is where user input is actually stored into some form of a database and permanently reflected onto a page (or pages) in the web application. Since this is permanent, it is far more dangerous as it can affect multiple people without any user interaction from them, whereas reflected XSS would (at the very least) require a user to click a specially crafted link.

The effects of stored XSS is the same as reflected XSS. Malicious actors can steal other user's cookies, or redirect users to phishing websites in order to steal their credentials.

Location in code

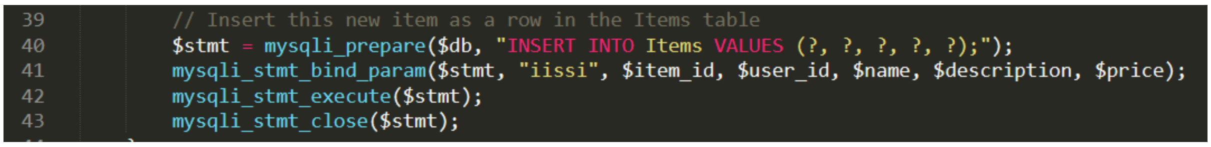
There are two separate instances of stored XSS in this web application.

The first instance is in **handle_register.php**. It's hard to pinpoint the lines of code, since the vulnerability is there due to the mitigation not being present in the first place, however, we can see specifically in lines 83-87, a new user is entered into the database using an INSERT statement, however the username / email address is not sanitized beforehand. Because of this, in both **admin.php** and **account.php**, a maliciously crafted username / email address will be output to the screen. This can be used to steal the admin's cookies (in the case of **admin.php**), or other user's cookies (in the case of **account.php** coupled with a broken access control vulnerability showcased further below).



```
83                  $stmt = mysqli_prepare($db, "INSERT INTO Users (id, username, password, email, locked, balance) VALUES (?, ?, ?, ?,
                        false', 0.0)");
84                  mysqli_stmt_bind_param($stmt, "isss", $db_id, $username, $password, $email);
85                  mysqli_stmt_execute($stmt);
86                  mysqli_stmt_close($stmt);
```

The second instance is in **handle_add_item.php**. Again, hard to pinpoint a specific line(s) of code, but we can see in lines 40-43, a new item is inserted into the database using an INSERT statement, however the item name and description is not sanitized beforehand. Because of this, both the item name and description will be output in **store.php** and **your_items.php**, letting a malicious actor steal both user and admin cookies, amongst other things.



```
39          // Insert this new item as a row in the Items table
40          $stmt = mysqli_prepare($db, "INSERT INTO Items VALUES (?, ?, ?, ?, ?);");
41          mysqli_stmt_bind_param($stmt, "iissi", $item_id, $user_id, $name, $description, $price);
42          mysqli_stmt_execute($stmt);
43          mysqli_stmt_close($stmt);
```

Trigger

In order to trigger this vulnerability, simply register a user with the username set to:
<script>alert(document.domain)</script>
and visit **account.php**, or **admin.php** as the admin, and you will see it in action.

Another way to trigger this vulnerability is to add an item with either its name or description set to:
<script>alert(document.domain)</script>
and visit either **store.php** or **your_items.php**. Of course, **your_items.php** will cause a self XSS which isn't that dangerous since there isn't a way for a user to access another user's **your_items.php** page, but the real danger comes from **store.php**.

Mitigation / Removal

This can be dealt with in the same way reflected XSS can be dealt with. The PHP built in function **htmlspecialchars** should be used on usernames and emails before inserting them into the database during the registration process. It should also be used on item names and descriptions before inserting those into the database during the add items process.

SQL Injection

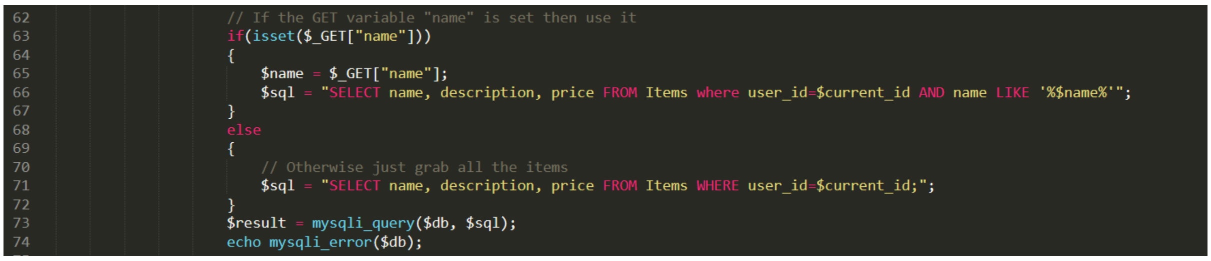
Description

An SQL injection attack happens when user input is inserted directly without sanitization within an SQL query. This effectively lets a malicious actor "inject" their own query into the already existing SQL query, which could potentially allow them to view information that they are not supposed to be able to view, or even in certain cases it may allow them to maliciously alter the database (such as drop tables or insert a new admin account that only they have access to).

In a normal SQL injection vulnerability (this), the results of the query are output straight onto the page in some fashion, which is what allows for the information leak in the first place. There is another form of SQL injection called a blind SQL injection, which will be covered in the next section.

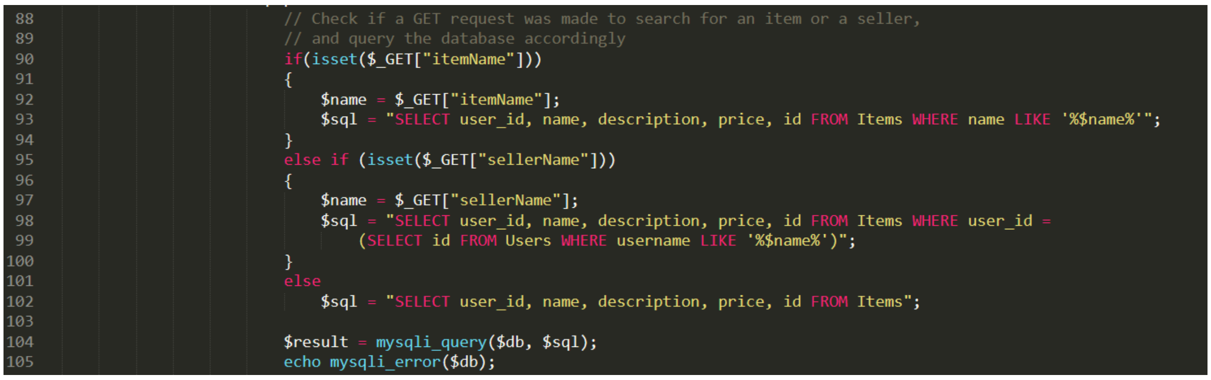
Location in code

There are six instances of an SQL injection vulnerability in the web application. The first is in **your_items.php** in the following lines of code, where a user is allowed to search for their own items through a GET parameter called **name**. The item name is not sanitized before inserting into the query, therefore an SQL injection is possible here.



```php
// If the GET variable "name" is set then use it
if(isset($_GET["name"]))
{
    $name = $_GET["name"];
    $sql = "SELECT name, description, price FROM Items where user_id=$current_id AND name LIKE '%$name%'";
}
else
{
    // Otherwise just grab all the items
    $sql = "SELECT name, description, price FROM Items WHERE user_id=$current_id;";
}
$result = mysqli_query($db, $sql);
echo mysqli_error($db);
```

The next two are in **store.php**, where any user can search for items in the store by either the item's name, or by the seller's name. Again, the item name / seller's name is not sanitized, but inserted into the query directly, which paves the path for a possible SQL injection attack.

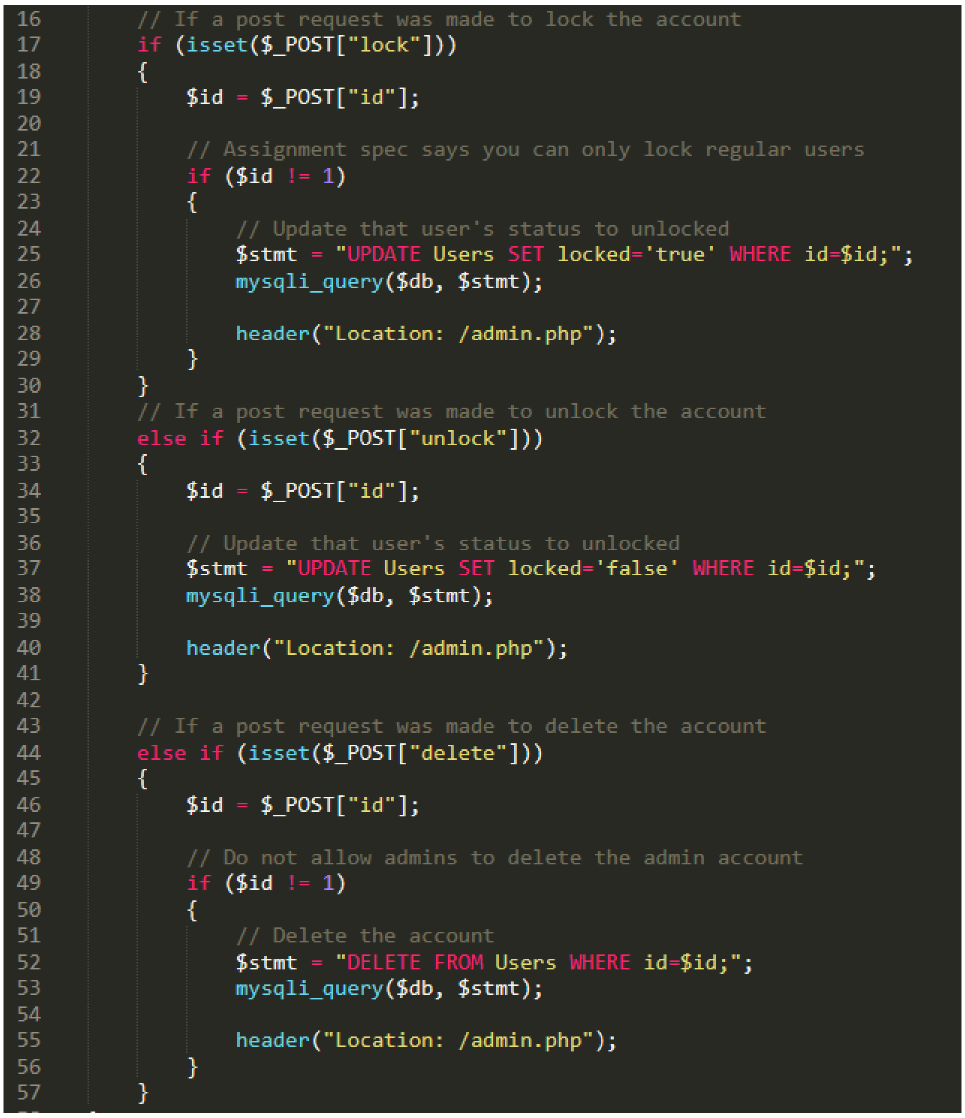


```php
// Check if a GET request was made to search for an item or a seller,
// and query the database accordingly
if(isset($_GET["itemName"]))
{
    $name = $_GET["itemName"];
    $sql = "SELECT user_id, name, description, price, id FROM Items WHERE name LIKE '%$name%'";
}
else if (isset($_GET["sellerName"]))
{
    $name = $_GET["sellerName"];
    $sql = "SELECT user_id, name, description, price, id FROM Items WHERE user_id =
        (SELECT id FROM Users WHERE username LIKE '%$name%')";
}
else
    $sql = "SELECT user_id, name, description, price, id FROM Items";

$result = mysqli_query($db, $sql);
echo mysqli_error($db);
```

The final three are in **handle_admin.php**, within the code that handles locking, unlocking, and deleting user accounts should an admin decide to do so. Whenever an admin wants to lock, unlock, or delete an account, a POST request is made with three possible payloads (for lock, unlock, and delete respectively):
id=2&lock=lock
id=2&unlock=unlock
id=2&delete=delete

However, as is evident from the code shown below, the **id** field is not sanitized, and thus an SQL injection is possible here which can allow the admin to lock, unlock, or delete all accounts at once. This is particularly dangerous because the admin should not be allowed to delete or lock any other admin accounts, which this vulnerability would allow:



```php
    // If a post request was made to lock the account
    if (isset($_POST["lock"]))
    {
        $id = $_POST["id"];

        // Assignment spec says you can only lock regular users
        if ($id != 1)
        {
            // Update that user's status to unlocked
            $stmt = "UPDATE Users SET locked='true' WHERE id=$id;";
            mysqli_query($db, $stmt);

            header("Location: /admin.php");
        }
    }
    // If a post request was made to unlock the account
    else if (isset($_POST["unlock"]))
    {
        $id = $_POST["id"];

        // Update that user's status to unlocked
        $stmt = "UPDATE Users SET locked='false' WHERE id=$id;";
        mysqli_query($db, $stmt);

        header("Location: /admin.php");
    }

    // If a post request was made to delete the account
    else if (isset($_POST["delete"]))
    {
        $id = $_POST["id"];

        // Do not allow admins to delete the admin account
        if ($id != 1)
        {
            // Delete the account
            $stmt = "DELETE FROM Users WHERE id=$id;";
            mysqli_query($db, $stmt);

            header("Location: /admin.php");
        }
    }
```

Trigger

For the first SQL injection vulnerability in **your_items.php**, simply have a bunch of items on the store, and perform a search with the following parameter:

' or 1=1 -- .

This will show every single item that is on the store, showcasing that an SQL injection attack is possible here.

For the next two SQL injection vulnerabilities in **store.php**, simply have a bunch of items on the store, and perform a search with the following parameter either for an item's name or for a seller's name:

' or 1=1 -- .

This will return every single item that is on the store, showcasing that an SQL injection attack is possible here.

For the final three SQL injection vulnerabilities, you will need to use a proxy such as Burpsuite or ZAP Proxy, or a REST API development client such as Postman to send POST requests. Using one of those tools, send any one of the following three POST requests to lock, unlock, or delete all accounts respectively:

id=2+or+1%3d1+--+.&lock=lock
id=2+or+1%3d1+--+.&unlock=unlock
id=2+or+1%3d1+--+.&delete=delete

Mitigation / Removal

PHP has a built in way to completely prevent SQL injection attacks from ever taking place. This is done through the use of a concept known as **PreparedStatements**. Essentially, the way this mitigation works is that instead of injecting the user input directly into the query, question marks (?) are used as placeholders in the query for the user input. Following that, each piece of user input is bound to each question mark, which essentially prevents SQL injections due to the fact that the query and the data is sent in separate requests. This effectively will completely mitigate all possibilities of SQL injection attacks.

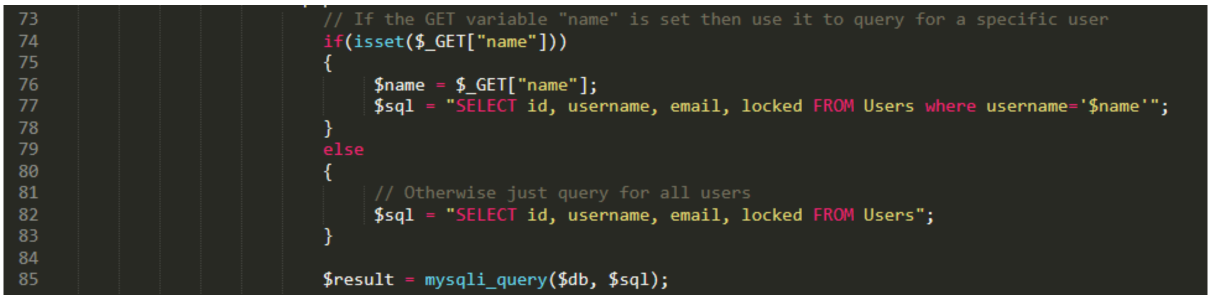
Blind SQL Injection

Description

SQL Injections have been explained in the above section. A blind SQL injection is where the effects of the query (its results) are not output onto the page directly, but can be inferred from a series of true and false questions. This could potentially allow a malicious actor to extract information from the database (such as user password hashes, or etc). An example is given below in the **Trigger** section.

Location in code

As far as I can see, there is one potential place where a blind SQL injection can be used to get data out of the database that a user should otherwise not have access to. In this case, the blind SQL injection is in the admin panel in **admin.php**, in the following lines of code:



```
// If the GET variable "name" is set then use it to query for a specific user
if(isset($_GET["name"]))
{
    $name = $_GET["name"];
    $sql = "SELECT id, username, email, locked FROM Users where username='$name'";
}
else
{
    // Otherwise just query for all users
    $sql = "SELECT id, username, email, locked FROM Users";
}

$result = mysqli_query($db, $sql);
```

As is evident, when an admin searches for a user by name, the name is passed in directly into the SELECT statement without any sanitization. This can allow an admin user to use a series of true and false questions to slowly but surely get a user's password hash out of the database (assuming of course that the admins won't have access to the actual database).

Trigger

To show an example of a trigger, I will use the fact that we already know that **normal_user**'s password is set to "password" without quotes. Since the passwords are stored as MD5 hashes in the database, we know that the MD5 hash of the word "password" is "5f4dcc3b5aa765d61d8327deb882cf99". We can then showcase that the following search query in the admin panel won't return any results since **normal_user**'s password does not start with the character 'a':

normal_user' AND 1=(SELECT 1 FROM Users WHERE username='normal_user' AND password LIKE 'a%') -- .

But the following search query WILL indeed return the result as if we just searched for "normal_user", since **normal_user**'s password has does indeed start with the character '5':

normal_user' AND 1=(SELECT 1 FROM Users WHERE username='normal_user' AND password LIKE '5%') -- .

This works because in MySQL, the character '%' is used as a placeholder in strings to denote "any number of any characters". So when we search for a password LIKE 'a%', we essentially mean "a password that starts with the character 'a' followed by any number of any characters". Same goes for password LIKE '5%'.

This way, a malicious admin can extract the password hash of every user one by one from the database, and use them to do any number of malicious things, since the passwords are hashed without any salt.

Mitigation / Removal

The mitigation here is the same as the one for normal SQL injections. If PHP's built in **PreparedStatement** functionality is used, it completely removes any chances of an SQL injection attack vector here. I have already explained how that works in the normal SQL injection section, so I will not repeat myself here.

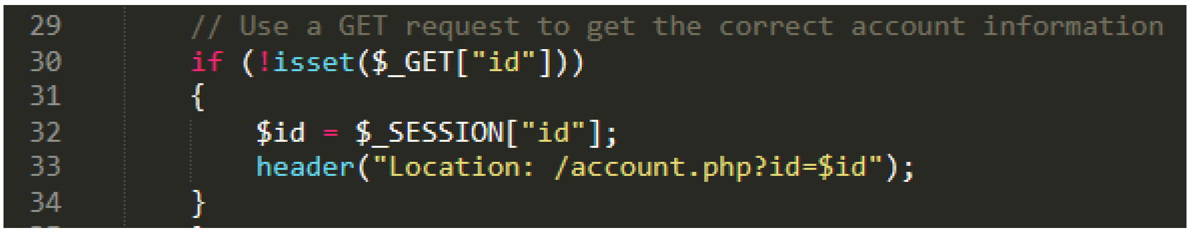
Broken Access Control

Description

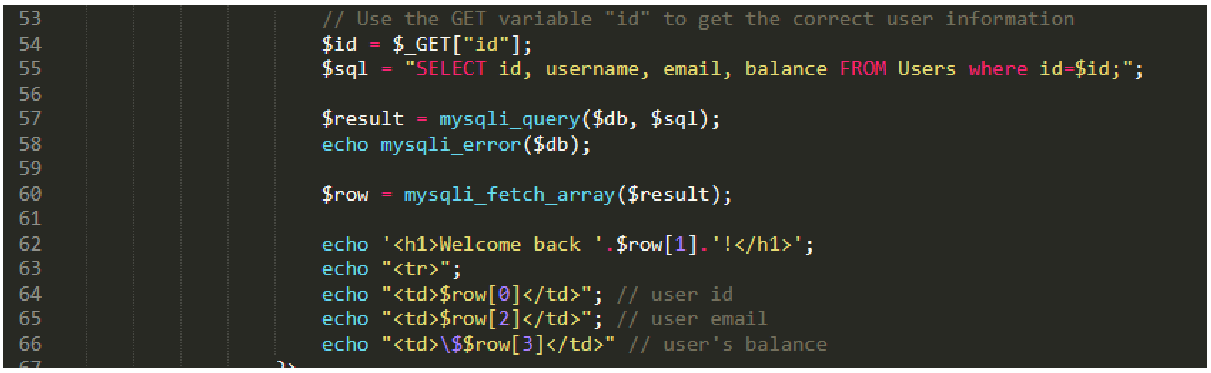
The term access control essentially just means that the amount of information or functionality a specific user should have access to should be controlled by some variable. As an example, if a web application has a "My account" page that shows the user all of their account details, then a specific user should only be able to access their own "My account" page. If they are able to access some other user's "My account" page, then this is an example of a broken access control vulnerability.

Location in code

There is only one instance of a broken access control vulnerability in this web application. It is in **account.php**, where a GET request is used to show to the user their own account details, as shown in the following two excerpts of code from **account.php**:



```
    // Use a GET request to get the correct account information
    if (!isset($_GET["id"]))
    {
        $id = $_SESSION["id"];
        header("Location: /account.php?id=$id");
    }
```



```
            // Use the GET variable "id" to get the correct user information
            $id = $_GET["id"];
            $sql = "SELECT id, username, email, balance FROM Users where id=$id;";

            $result = mysqli_query($db, $sql);
            echo mysqli_error($db);

            $row = mysqli_fetch_array($result);

            echo '<h1>Welcome back '.$row[1].'!</h1>';
            echo "<tr>";
            echo "<td>$row[0]</td>"; // user id
            echo "<td>$row[2]</td>"; // user email
            echo "<td>\$$row[3]</td>" // user's balance
```

However, if a user were to manually change this GET parameter to another user's id, then they would have broken access control and been able to view that user's details. This is showcased in the following section.

Trigger

In order to trigger this vulnerability, simply go to the **accounts.php** page, and change the GET parameter **id** to anything other than the current user's id. For example, logging in as **normal_user** (id=3) and changing the GET parameter **id** to a value of 1 will show you the account details for the **admin** user.

Mitigation / Removal

This type of vulnerability is very often overlooked, but it is actually very easy to mitigate. There are no specific methods to mitigating broken access control as the type of the mitigation will be on a case by case basis.

In this case, simply changing it so that the **id** is directly used from **$_SESSION["id"]** instead of a GET parameter (which the user can change) will solve the issue as PHP session ids are actually encrypted and cannot easily be tampered with to view a different user's account details. Of course, the user can use an XSS attack to get another user's session cookie and masquerade as them that way, but that is a different vulnerability altogether.

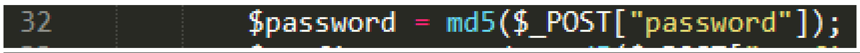
Broken Cryptographic Algorithm

Description

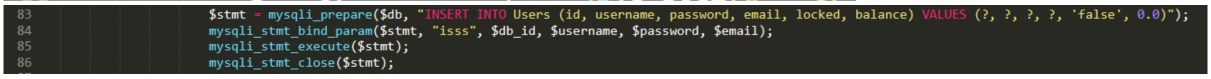
A broken cryptographic algorithm vulnerability is when a cryptographic algorithm that is known to be weak (easily susceptible to brute forcing or etc) is used to encrypt sensitive information before storage. It also applies to storing hashed passwords without a salt, as then if the password used is not ***extremely*** complex, rainbow tables can more than likely be used to figure out the password given the hash.

Location in code

In my web application, there is one instance of a broken cryptographic algorithm in use, and that is in the storage of hashed passwords. Passwords are simply stored as MD5 hashes. If a malicious actor somehow gains access to a password hash, they can either crack it or use a rainbow table to figure out the password. The hashing process is shown below in **handle_register.php**:



```
$password = md5($_POST["password"]);
```



```
$stmt = mysqli_prepare($db, "INSERT INTO Users (id, username, password, email, locked, balance) VALUES (?, ?, ?, ?, 'false', 0.0)");
mysqli_stmt_bind_param($stmt, "isss", $db_id, $username, $password, $email);
mysqli_stmt_execute($stmt);
mysqli_stmt_close($stmt);
```

Trigger

In order to showcase a trigger for this vulnerability, let us assume that a malicious actor has somehow gained access to the **admin** account's password hash, which is "5f4dcc3b5aa765d61d8327deb882cf99".

Simply plug the above hash into https://crackstation.net, solve the captcha, and click "Crack hashes". It will almost instantly return the password to you, which is just "password".

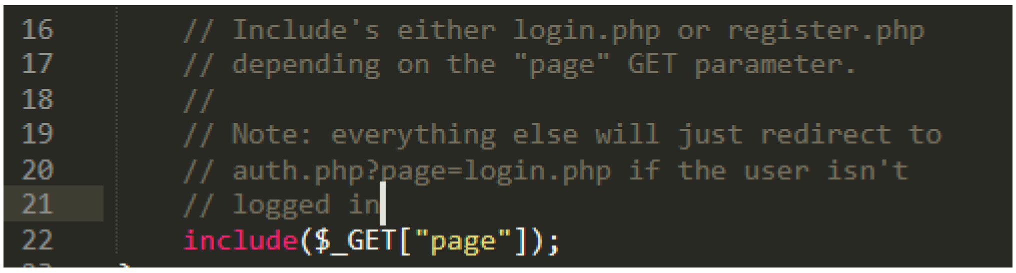
Mitigation / Removal
The simplest mitigation would be to append a salt to the password before hashing it. If using a salt, even MD5 will be pretty good as a hashing mechanism. However, for maximum mitigation potential, an algorithm such as SHA512 should be used along with a salt. SHA512 is known to be very cryptographically secure and extremely costly to calculate hashes for, so it fits our situation perfectly.

PHP File Include

Description
A PHP file include vulnerability is where a user is allowed to decide on what PHP file can be included onto a page. This is usually done through the use of GET parameters, but can also be done through other means.

Given a vulnerability such as this, a malicious actor can potentially use a path traversal attack to include any files they want, including important files such as /etc/passwd, or other configuration files to perform recon.

Location in code
There is one instance of a PHP file include vulnerability in my website, and that is in **auth.php**. When a user does not have a session established, they are redirected to **auth.php**, which has a GET parameter called **page** that is used to decide whether to show the user **login.php** or **register.php**, as shown below:



```
    // Include's either login.php or register.php
    // depending on the "page" GET parameter.
    //
    // Note: everything else will just redirect to
    // auth.php?page=login.php if the user isn't
    // logged in
    include($_GET["page"]);
```

A malicious actor can change this GET parameter and perform a path traversal attack easily. This is showcased below.

Trigger
In order to trigger this vulnerability, first logout to ensure no session is established. Then, simply change the **auth.php?page=login.php** GET parameter to something like **auth.php?page=/etc/passwd**. You will be given the output of the /etc/passwd file on the server, which shows that this vulnerability exists.

Mitigation / Removal
The simplest way to mitigate this vulnerability is to avoid using user controllable input in any PHP include statement. In this case, simply hard coding **login.php** or **register.php** without using a GET parameter would have sufficed, however that is not a mitigation that can be used in a larger web application. The safer way would be to remove **auth.php** completely and just show the user the **login.php** or **register.php** page directly.

Server Misconfiguration

Description

A server misconfiguration occurs when there is a security misconfiguration somewhere in the server. This can lead to information leakage as well as open up the potential for some attacks that would otherwise not be possible if the server was configured correctly.

A simple example of this is when directory listings are not disabled. When that happens, an attacker can enumerate directories and view all files that exist in that specific directory which can potentially lead to information leakage.

Another example would be when a server is configured to use HTTP only. In these days, HTTPS should be considered mandatory, and thus having a server using HTTP only is considered a security misconfiguration as it opens up the potential for man in the middle attacks.

Location in code

My web application has two instances of a server misconfiguration.

The first is that it doesn't use HTTPS (meaning it doesn't use TLS or SSL). There aren't really any places in my code where I can showcase this vulnerability as existing. The only way to showcase this is to open the website and see for yourself that it is only using HTTP.

The second instance is that directory listings are not disabled. This can be seen in the /etc/apache2/sites-available/000-default.conf file. There aren't really any specific lines that I can show, but in order to disable directory listings, the line "Options -Indexes" should exist under the options set for the root directory of the website.

Trigger

Again, there aren't really any ways to showcase "triggering" the first instance of the server misconfiguration. The only way to see it is to open the website and visually inspect it to see that it is indeed not using HTTPS.

In order to trigger the second instance though, simply visiting http://assignment.com/includes will show a directory listing of the /includes directory, which showcases a trigger for this vulnerability.

Mitigation / Removal

In order to mitigate the first instance of the server misconfiguration, I would need to generate my own certificates (by probably using openssl) and then configure Apache to use those certificates to communicate with anyone that visits the website. That is the simplest way to do it. The right way would be to get a certificate from somewhere like https://letsencrypt.org, and then the certificate would be signed by a root Certificate Authority.

To mitigate the second instance of the server misconfiguration, simply adding the following line to /etc/apache2/sites-available/000-default.conf below the lines "<Directory /var/www/html>" and "<Directory />" will fix the issue:
Options -Indexes

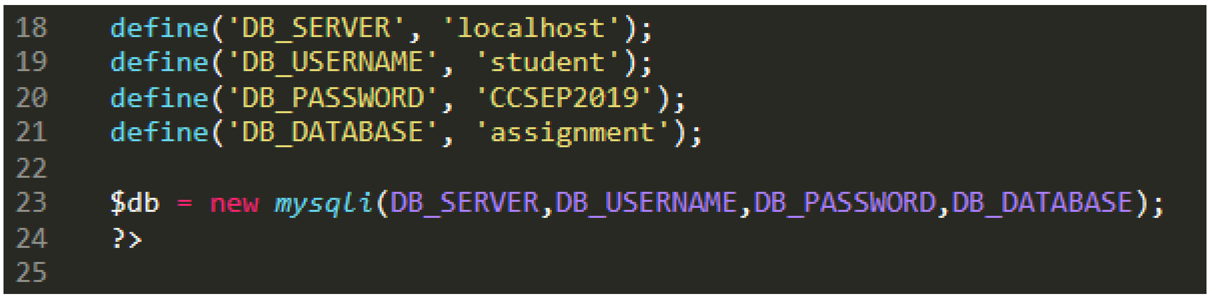
Use of hard-coded passwords

Description

This type of a vulnerability occurs when passwords are hard-coded either in the code itself or in config files that the code uses. If, for some reason, the code for the web application is leaked, then a malicious actor will have gained access to whatever the hard-coded password is meant to protect.

Location in code

There is one instance of a usage of a hard-coded password in my web application, and that is when a connection is established to the database in **db.php**, as shown below:



```
define('DB_SERVER', 'localhost');
define('DB_USERNAME', 'student');
define('DB_PASSWORD', 'CCSEP2019');
define('DB_DATABASE', 'assignment');

$db = new mysqli(DB_SERVER,DB_USERNAME,DB_PASSWORD,DB_DATABASE);
?>
```

As is evident, the password "CCSEP2019" is hard-coded into the code. If this file is somehow leaked, a malicious actor would have access to the entire database which is extremely dangerous.

Trigger

A quick nmap scan will show that port 3306 is open which is the port used by MySQL. A malicious actor, using this password, would then be able to connect to the database remotely, authenticate, and steal all the data that exists within it. In order to connect to the assignment database remotely for example, one would be able to run the following command:
mysql -u root -p -h assignment.com -P 3306 -D assignment

Mitigation / Removal

The way I would mitigate this vulnerability is to actually store the passwords in some configuration file that does ***not*** live in the root directory of the website. What I mean by that is, if for example the **index.php** file is in /var/www/html, then I would store the password in a (for example) **config.env** file within /var/www/config.env. This way, the web server can still access this file, but it is completely inaccessible from within the web application (barring a path traversal attack which is a different vulnerability).

Known Defects

I would say that there is one major defect in my web application, and that is the fact that it doesn't use **.htaccess** file. I considered putting it in after I'd finished all the functionality required for the web application, but I've been swamped with studying for IDS and PTD, so I couldn't be bothered changing the code around and fixing all the bugs that pop around in the process of adding a **.htaccess** file. It would just take up too much time.

Due to this, files that aren't really supposed to be viewable, such as **db.php**, or **/includes/*.php**, are actually openable and viewable in the browser. Of course, since they only contain PHP code (barring the **/includes/header.php** file), they show up as completely blank, but I would consider this a defect in my program, although I don't believe I should lose any marks for it as it wasn't a requirement in the assignment brief.

Conclusion

This assignment was quite fun to do! I've never used PHP before so it was an interesting challenge. It was also very useful since I've been thinking about getting better at web exploitation since forever, and this assignment forced me to learn about a lot of the different basic bug classes that exist within web applications.

I hope my assignment is worthy of a 100%!

THE END