Assignment Report

Cyber Crime & Security Enhanced Programming

Samantha Taylor - 18863512

Curtin University
Science and Engineering
Perth, Australia
Oct 2019

System Overview

Specifications

This system is set up in a virtual machine that is configured with a LAMP *(Linux, Apache, MYSQL, PHP)* stack setup. The web application hosted on this machone allows users to buy and sell items. Hidden beneath the surface are exploitable vulnerabilities such as, *XSS, SQL injection, Broken Access Control* and more.

The overall aim is to explore the importance of understanding how various flaws and vulnerabilities work, how to discover and prevent them.

Usage

The machine is configured with the static IP address of 192.168.56.150 for a host only adapter. The credentials for the VM is student with the password CCSEP2019. Which can be run from VirtualBox.

Functionality

1. Create accounts
2. Additions of funds for users
3. Putting items up for purchase
4. Purchasing items using funds
5. Searching capability
   (a) Search for items by name
   (b) Search for items by seller
6. Admin Features - note an Admin user has already been created with the credentials of Admin:Admin.
   (a) Disable/lock regular users
   (b) Remove items for sale

Files Breakdown

Screen shots of these files can be found at the bottom of this report.

SQL Files:

1. Init_scehma - creates the database and three tables (test, users, and items).
2. Seed_data - seed data test all three tables.

   NOTE: A user with the username 'Admin' has been created with 'Admin' as the password.

Text Files:

1. robots.txt - basic example that shows admin.php file exist.

PHP & Corresponding CSS Files

1. admin.php - displays all users and items. Provides a way for admin to enable/disable users. As well as delete users and items posted.
2. db.php - database connection file.
3. index.php - login and create user page.
4. itemList.php - component used to display all items found on the database.
5. login.php - functions for creating and logging in a user. This is where the user session is started.
6. logout.php - simple function for destroying a users session when they click to log out.
7. navbar.php - provides a navbar to all pages that includes it. Only displays the admin page link if the user has the permissions.
8. searchPage.php - displays all items for sales. Also provides searching capabilities by item title and owners name.
9. settings.php - provides a place for the user to change their password.
10. utils.php - contains functions for creating/deleting items, handling money transactions, and locking/deleting accounts.
11. validateSession.php - is included on every page to ensure that the user is correctly logged in. If not they are sent to the login screen.
12. welcomePage.php - is the home page for the user. It displays the users money balance and provides the user with the opportunity of selling an item. It also displays the users account status and user level.

Vulnerabilities

XSS - Reflected

1. Description:

   When a user inputs text that happens to be scripting code, it gets interpreted and executed on the fly as genuine JavaScript (which is capable of generating HTML). This is possible when you display input that was sent from the user eg. a search form as shown below.

Figure 1: Reflecting Back User Input

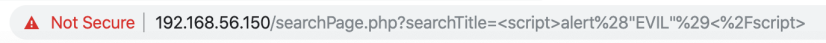

Figure 2: Search Page URL containing GET query string

2. Location in Code:

   (a) On the search page you can type what you want to search for in either search box. Since there is no proper sanitization being done, the user is free to type whatever they want, which will be reflected back to the user (which can embedded javascript into the original page). This is because we are reflecting back what the user inputed verbatim (no sanitization).

3. Triggering It:

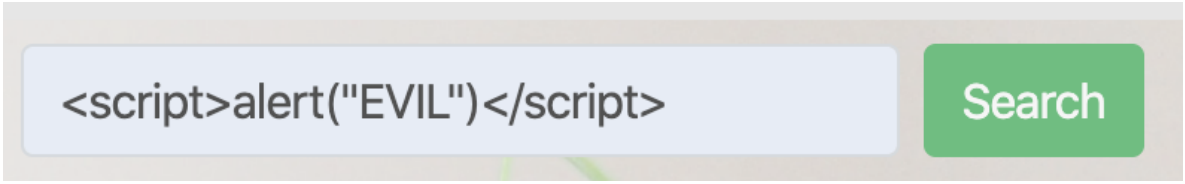

Figure 3: Search page XSS Reflected

4. Mitigation or Removal:

   For easy removal of this vulnerability, simply don't reflect any user input back to the user. Note this is not always easy to obtain. Normally it provides a less user friendly interface. Security often takes a backseat when it comes to usability.

   If it cannot be removed, we can try to limit the possibilities of a successful attack. To do this requires deep sanitization on user input. When it comes to sanitizing user input, take the *'Guilty-unless-proven-innocent'* approach. If user input seems suspicious, simply reject it. For example, if a user inputs '<' or special characters unneeded like '&' then reject it to minimize the liklihood of a successful injection.

XSS - Stored

1. Description: Stored XSS is very similar to Reflected, however it is more dangerous as it is persistent. This is because it affects all users, not just users who may have been sent a link containing injected (Reflected/Non-persistent) XSS. Persistent XSS occurs when malicious JavaScript is saved on a trusted sit's page and displayed to **all visitors**.

username	password	money	userEnabled
Sammi	42f749ade7f9e195bf475f37a44cafcb	0.00	1
John Doe	42f749ade7f9e195bf475f37a44cafcb	0.00	0
Test User	42f749ade7f9e195bf475f37a44cafcb	0.00	1
<script>alert("EVIL")</script>	8b1a9953c4611296a827abf8c47804d7	0.00	1

Figure 4: Example of Stored XSS in the Database

2. Location in Code:
   (a) Usernames gets displayed to all users searching items
   (b) Also appears on the users welcome screen (less major as it only affects that user)
   (c) You can also do this with item titles and descriptions.

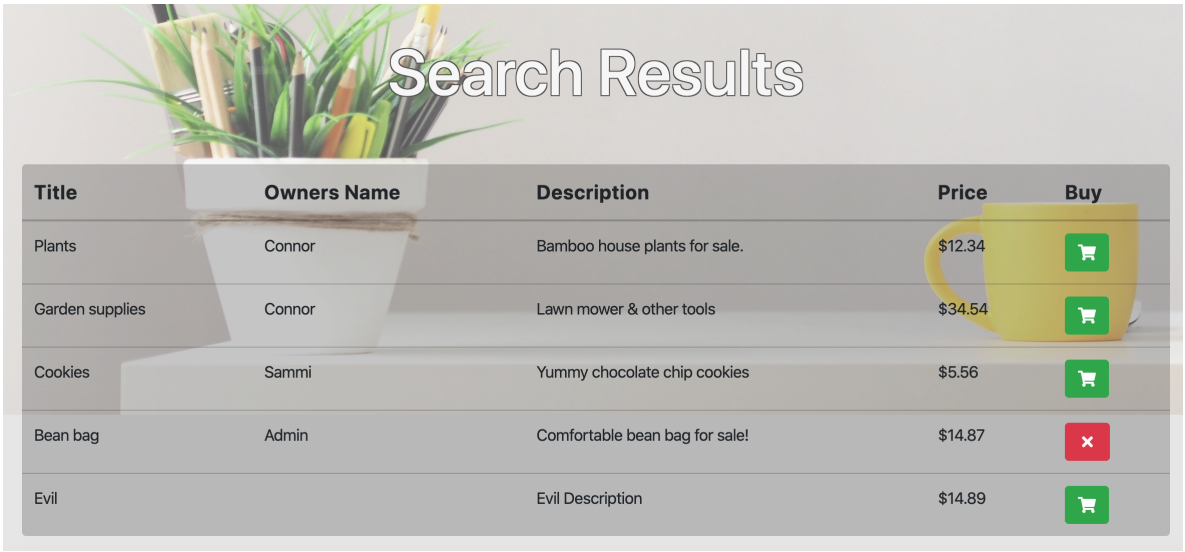

Figure 5: *(a)* Affects all users who goes to the search results page when the evil user has created an item to sell.

```
<td>Evil</td>
▼<td>
    <script>alert("EVIL")</script> == $0
</td>
<td>Evil Description</td>
<td>$14.89</td>
```

Figure 8: If we inspect the search page on the row with the item Title is Evil we can see the embedded script tag.

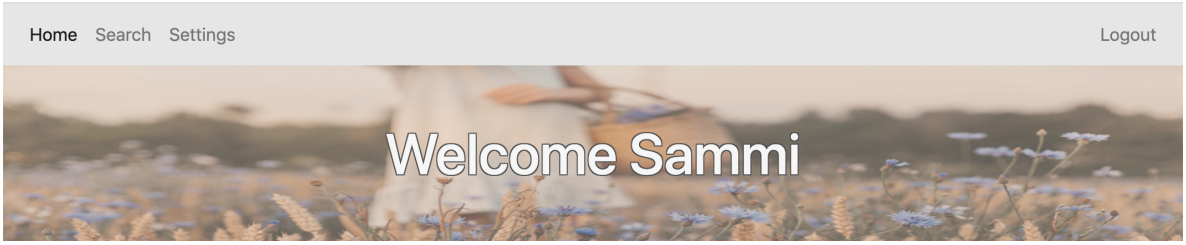

Figure 6: *(b)* Only affects the user who created this account.

3. Triggering It:

   When creating a user, create a username containing malicious code. This code will get displayed (and automatically executed) on the Search page if that user has an item up for sale.

Figure 7: When we go to the search page the JavaScript automatically gets executed. In this case it is a harmless alert. However, it could be malicious and could execute without any knowledge of the victim.

4. Mitigation or Removal:

   As before with the Reflected XSS the only way to fully prevent this is to not reflect any user input to users. Otherwise sanitization is key. Reject it if it is not trust worthy or if it seems suspicious. For more details please reference to the Reflected XSS section above.

SQL Injection

1. Description:

   Occurs when a hacker is able to send and execute their own SQL instructions into your relational database. This can happen when user input is passed to the database without any input validation, not checking for it being malicious.

2. Location in Code:

```php
// Creates a user with a unique username into the db table users
function createUser($db, $username, $password) {
  $sql = "INSERT INTO users (username, password) VALUES ('$username', '$password')";
  $resultStr = "";

  if (mysqli_multi_query($db, $sql)) {
    $resultStr = "Successfully created: " . $username;
  } else {
    $resultStr = "Could not create a new user: " . mysqli_error($db);
  }

  return $resultStr;
}
```

Figure 9: login.php - unsafe SQL practices

3. Triggering It:

   On the login/create user page, if you enter *'; UPDATE users SET isAdmin = 1 #* as the username and anything random for the password value in the form. The SQL string that is constructured is *INSERT INTO users (username, password) VALUES ("; **UPDATE users SET isAdmin = 1 #** ,'somePassword')"*. The string actually contains more than one SQL query. In bold you can see the input actually changes all the users in the database to have their *isAdmin* flag set to *true*.

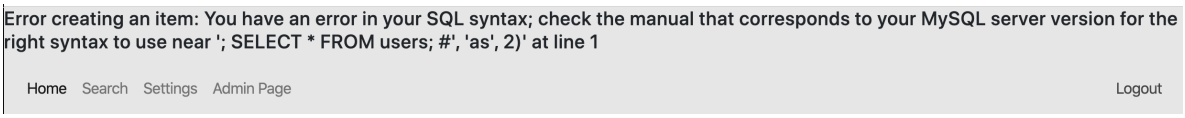

Figure 10: Detailed error message shown to any user who enters an SQL statement that causes a SQL syntax error.

4. Mitigation or Removal:

   Don't use multi-statements for SQL. They are rarely actually needed and are very dangerous tool that enables malicious users to cause great damage. Instead, use prepared statements with variable binding (parameterized queries). This separates the basic SQL statement from the user-input parameters. This allows the database to distinguish between code (SQL) and data (user input). This ensures the attacker is not able to change the intent of a query. They also make code cleaner, easiler to read, and more secure.

   Sanitization is also important. Once again, if the user is entering suspicious input you should reject it to reduce the risk of a successful attack. Otherwise you can try and convert their input

to something less harmful by escaping those characters. Another solution, suggested by OWASP is to whitelist possible valid input. Then only allow input that matched the list, making for easy validation.

An additional defense is to enforce the least privileges policy. Don't have the application have full write/delete access of the database if it only needs to read from it. Limit the damage an attacker can do by limiting all privileges as much as possible.

```php
// stmt - statement
$stmt = $dbConnection->prepare('SELECT * FROM users WHERE name = ?');
$stmt->bind_param('s', $username); // type s for string
$stmt->execute();
$result = $stmt->get_result();
while ($row = $result->fetch_assoc()) {
    // do something with $row
}
```

Figure 11: OWASP: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL_Injection_Prevention_

SQL Injection - Blind

1. Description:

   The difference between normal SQL injection verse blind SQL injection is the way the data is retrieved from the database. When the database doesn't output data to the web page or doesn't any errors to you. For example, in the example above if you'd have a SQL syntax error an error message appears stating detailed information of the database. This normally happens when the application is configured to show generic error messages (but is still vulnerable to SQL injection).

   The only way to steal data is by asking the database series or true or false questions and determine the answers based on the applications response.

$$IF(condition, truepart, falsepart)$$

$$username \ = \ 'Admin' \ OR \ IF(condition, SLEEP(1), SLEEP(0))$$

2. Location in Code: loginUser function found in the login.php file.

3. Triggering It:

   On the sign in section on the (*index.php*) page, go to the username and enter *Admin' and if(1=1, sleep(5), sleep(0)); #*. Also enter something in the password part so the form can be submitted. The page will take 5 seconds to load which allows us to know that our statement is true, there is a user with the name **Admin**. We know our statement isn't true if the page is loaded immediately (eg. sleep(0)). For example, we type *lauren' and if(1=1, sleep(5), sleep(0)); #*, it will load straight away as there is no user named lauren in the database.

4. Mitigation or Removal:

   Same as normal SQL injection, sanitize user input or use prepared statements. Please reference SQL injection above for more mitigation techniques.

Broken Access Control

1. Description:

   Access control is how a web application grant access to content and functions to some users and not others.

   These checks are performed after authentication, and govern what 'authorized' users are allowed to do.

   One specific type of access control problem is administrative interfaces that allow site admins to manage a site over the internet. (manage users, data, and content)

2. Location in Code:

   The *admin.php* page checks if the user is logged in. However, it doesn't check if the users *isAdmin* flag is set to true like the *navbar.php* page does.

3. Triggering It:

   Any user can actually go to http://192.168.56.150/admin.php. There is no checks on this page to prevent any user just manually accessing the page. Only the users with the isAdmin flag set to true (1) in the database will see the direct option on the navbar to navigate to that page. To access the page you must be logged in a user.

   If a user navigates to the *robots.txt* file they will actually see that /admin.php exists and are probably very curious on what they can do with it.

   Since the insecure admin flag stored in the users database, if a user does any sql injection that changse the isAdmin flag to true (1). Then that user will get access to the navbar with the extra admin features directly through the navbar.

```
+----+------------+----------------------------------+-------+-------------+---------+
| id | username   | password                         | money | userEnabled | isAdmin |
+----+------------+----------------------------------+-------+-------------+---------+
|  1 | Sam Taylor | 42f749ade7f9e195bf475f37a44cafcb | 74.10 |           1 |       0 |
+----+------------+----------------------------------+-------+-------------+---------+
```

Figure 12: User Database Table

4. Mitigation or Removal:

   To fix this ensure the admin page checks the isAdmin flag and only allows admins to continue to this page. Having a policy and lists of checks to investigate before pushing code to production or to the client to check the access controls work as expected.

Broken Cryptographic Algorithm

1. Description:

   MD5 is considered broken with regards to its *hash collisions and it's quickness* compared to other algorithms. At first glace, a quick algorithm seems appealing. However, slowness in this regard is important. Computers are becoming faster and can try more input possibilities. This means an attacker can try password guessing billions of potential passwords per second.

   Overall, MD5 shouldn't be implemented in applications where a collision-resistant hash function is required or where the hash proves identity, such as passwords (along with the username).

2. Location in Code: LoginUser() inside login.php file.

```
function loginUser($db, $username, $pwd) {
  $password = md5($pwd);
  $sql = "SELECT * FROM users WHERE username='$username' AND password='$password'";
```

Figure 13: MD5 & Insecure SQL

3. Triggering It:

   MD5 can be found is use when creating or logging in a user. The main problem is concerned with logging in. Where an attacker can try password cracking another users account. Anyone creating a user can actually determine if a username exists and has already been taken. If someone tries to create a user with the same name they will get an error message. With this information an attacker can target that user account and try to crack their password and break in.

4. Mitigation or Removal:

   One solution is to completely remove the use of MD5. As a replacement, it is recommended to use deliberately slow hashing algorithms like SHA-512 or bcrypt which is the industries standard. Another solution is to put a limit on the number of attempts of incorrect password input. For example, put a 5 minute lock on an account if a login attempt has failed 5 times and increase the time if they continue. It is a similar approach to before where we are trying to increase the time.

PHP File Include

1. Description:

   Can allow an attacker to view files on a remote host they shouldn't be able to see. It can even allow the attacker to run code on a target.

2. Location in Code:

```php
<?php
  if (isset($_GET['page'])){
    include($_GET['page']);
  }
?>
```

Figure 14: welcomePage.php when a user selects change password it sets *page="settings.php"*

3. Triggering It:

   On the Home page click on the *Change Password* button. It will update the GET query to have the key-value pair of *page="settings.php"*. The settings page will now appear below the three cards on the home page.

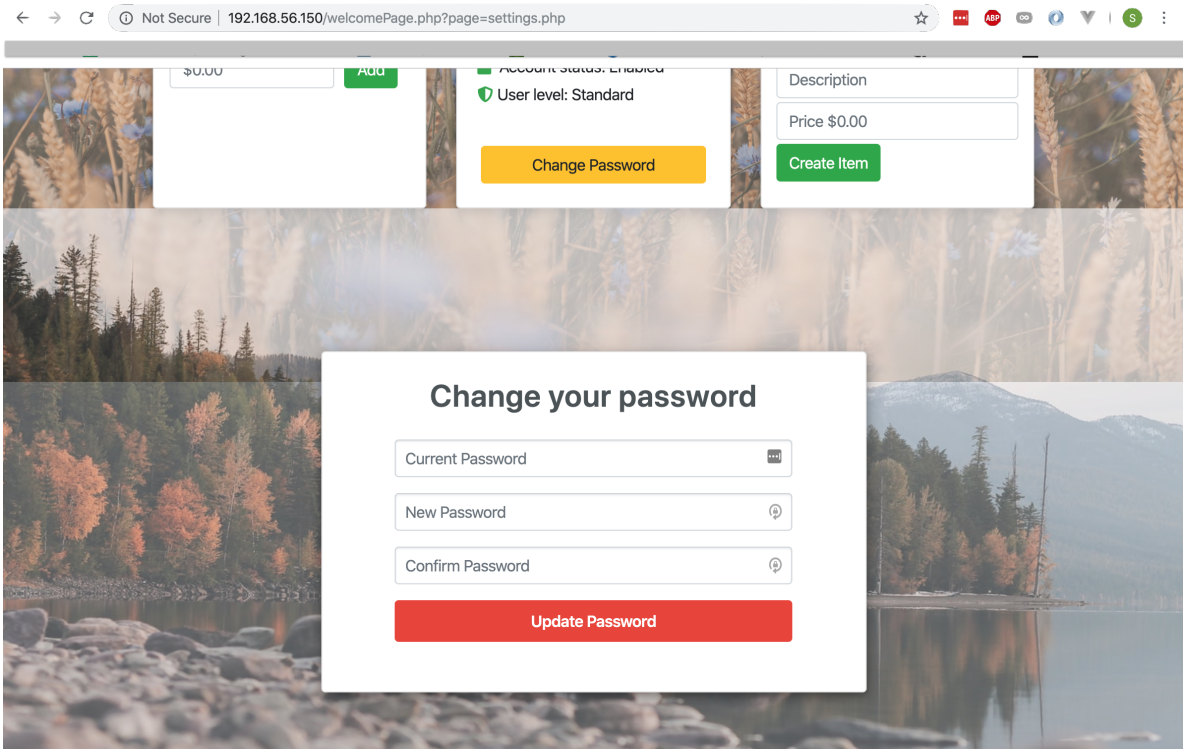

Figure 15: Home page now displays settings page without having to completely take the user over to the settings page.

What if we go the URL and change *"settings.php"* to *"../../../etc/passwd"*? Wow now the password hashes from the /etc/ folder are displayed!!

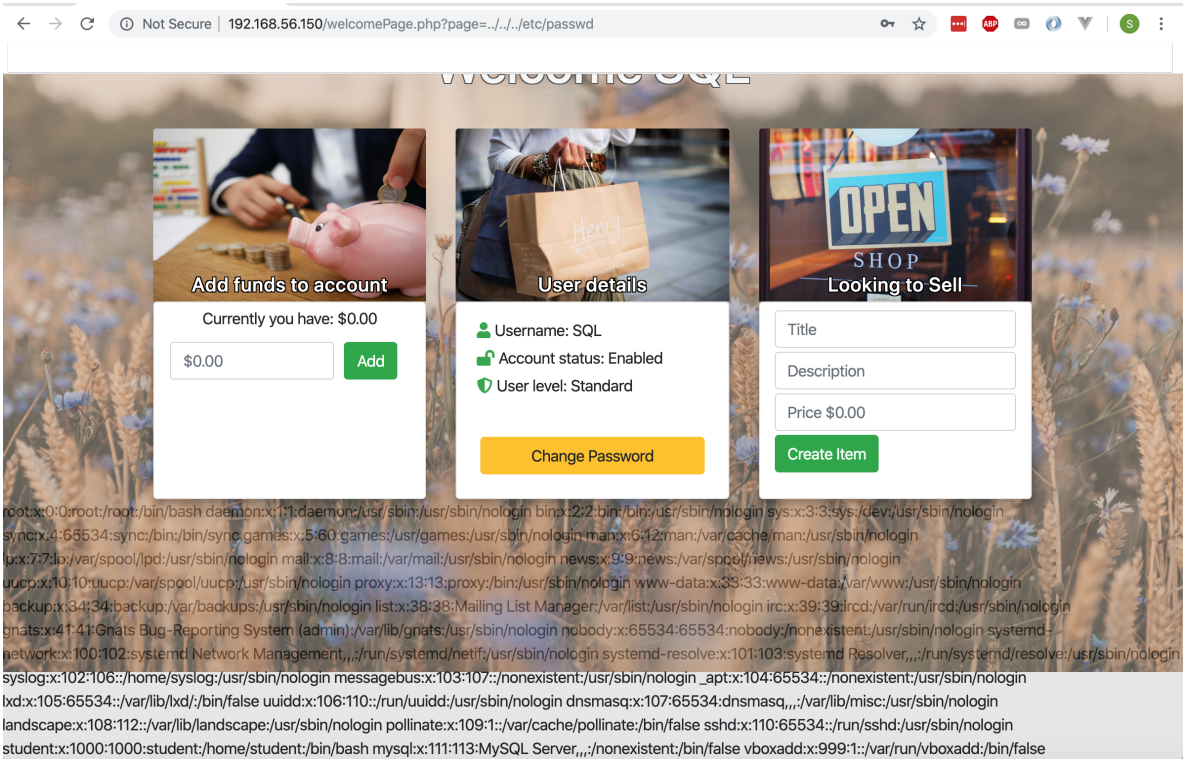

Figure 16: Free range over what file is displayed below the normal home/welcome page.

The attack could happen if you allow user input to determine the file path without sanitization or checks

4. Mitigation or Removal:

   The most effective solution is to not use user input to determine the file inclusion.

   If it is not possible for application to work without user input then validate their selection. Validate the user input and check against a whitelist of files if their selection is valid file for them to access. Again this comes down to input sanitization as many of the vulnerabilities. Other than that, set the value of *allow_url_include* and *allow_url_fopen* to *Off* in the PHP configuration file.

Server Misconfiguration

1. Description:

   The server misconfiguration is that it runs over HTTP rather than HTTPS. This means that our HTTP request and responses are sent in plaintext. Furthermore, this allows anyone who captures our traffic to know exactly what is being sent back and forth. The level of information could range from non-sensitive to highly-sensitive information, including passwords, bank details, or company secrets.

2. Location in Code: the whole website is insecure as the site does not have a SSL certificate or corresponding key.

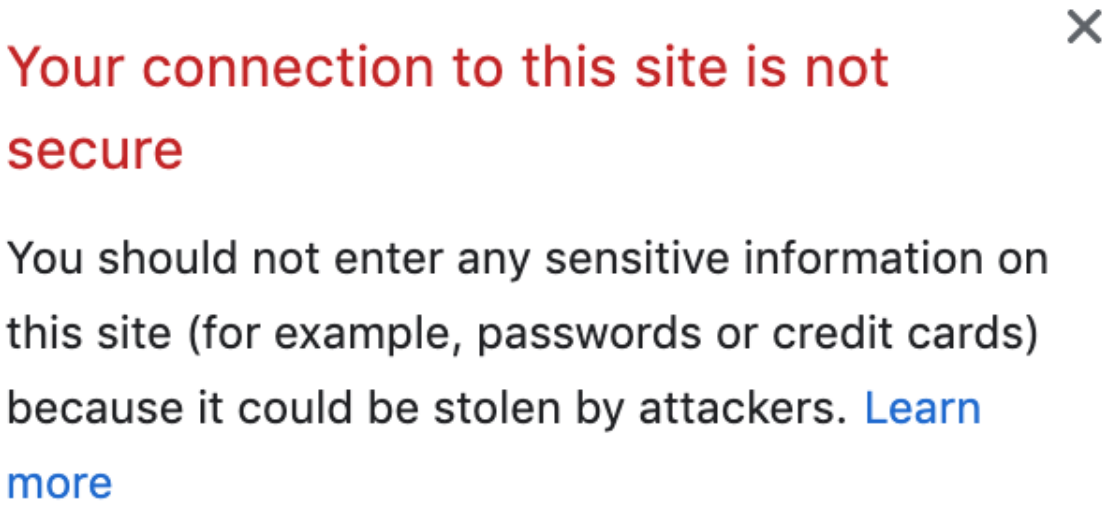

Figure 17: Chrome HTTP Popup

3. Triggering It: Certain browsers, like Chrome have warning popups and tagging the URL 'Not Secure' to notify the user of the dangerous security flaw on websites. Using the website with a *http* protocol in the URL is how you can determine in is indeed using HTTP.

4. Mitigation or Removal:

   HTTPS corrects this problem by using TLS/SSL encryption. These security protocols enables CIA (confidentiality, integrity, availability) to our internet communications. The encrypted data can still be intercepted by malicious parties. Now however, they will only see the encrypted data that cannot be decrypted to its original form without the correct key.

   This type of system uses two different keys to encrypt communications between two parties. A *public key* and a *private key* are used. The SSL/TLS also prevents impersonations by confirming a website server is who it advertises to be using these keys. To prove your identify, you *sign* a part of some initial data with your private key (which only you have access to). The other user can truly sent you sent it by decrypting this information with your public key. When it comes to sending real data, the sender will use the public key of the receiver, as only they can decrypt it with their corresponding private key.

Hard-coded Passwords

1. Description:

   Hard-coded passwords are when credentials, secrets, or plain text passwords are found directly in source code. They are considered bad as it poses as an easy target for password guessing exploits. The problem with this is if malicious users find this information, they can:

   (a) Gain access to sensitive information (confidentiality),

   (b) Change valueable information (integrity),

   (c) Completely destory data (availablity),

   (d) Take over the device and use it as a botnet.

2. Location in Code: *db.php*

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

// Connect to the database
$db = mysqli_connect(DB_SERVER,DB_USERNAME,DB_PASSWORD,DB_DATABASE);
```

Figure 18: Hard-coded Passwords

3. Triggering It:

   This is a problem if someone can find your source code (published code on GitHub or other software sharing sites). They can easily identify the passwords being used. It is also inconvenient if the developer needs to change the password. For this to occur they must patch/update the code.

4. Mitigation or Removal:

   If possible, remove the need for these files. For example, when handling *inbound authentication* have a 'first login' mode that gets the user to create a unique, strong password. This should replace the need for hard-coded values. However, this is not always possible.

   For *outbound authentication*, store credentials outside of the code in a strongly-protected encrypted file, apply strong one-way hashes, or a database that is protected from access by all outsiders (including other local users). If you cannot encrypt to protect the file, then ensure the permissions are restrictive as possible

   If the code is going to be shared, ensure there is a policy on source release. This should define the proper procedures to minimize the likelihood of freely releasing sensitive information. For example, using git ignore on sensitive files. On another note, this is still a problem if the code is compiled and release. It can possibly be decompiled so you should still avoid using hard-coded passwords wherever possible.

Image Breakdown of Website

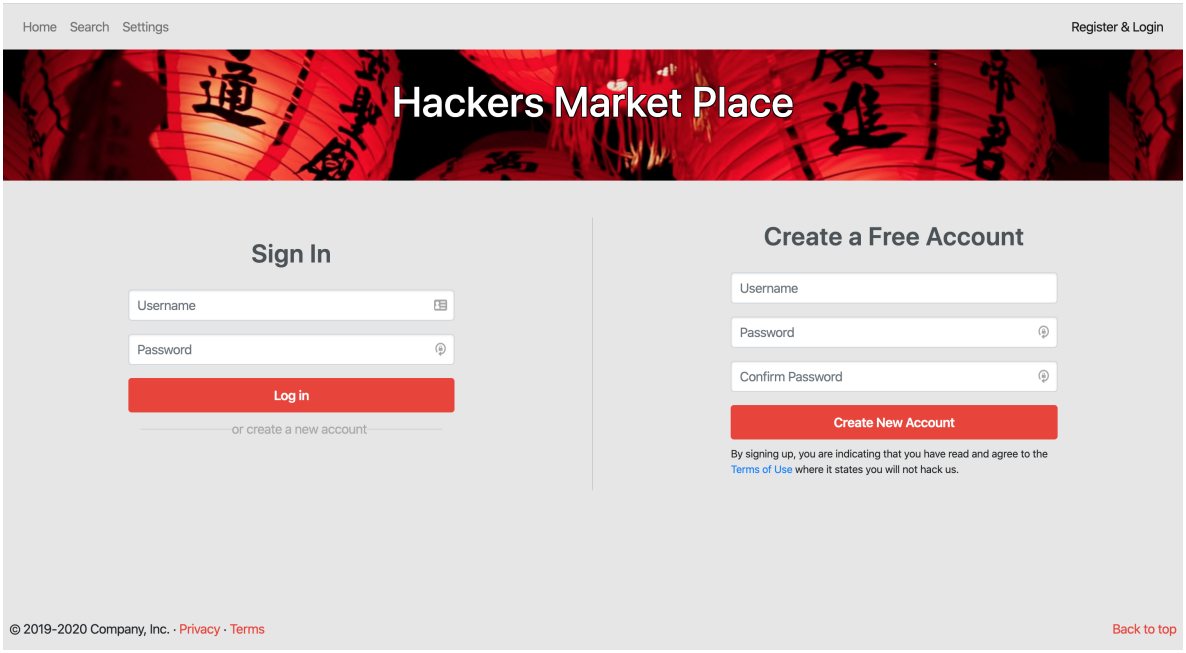

Figure 19: Login Page - index.php

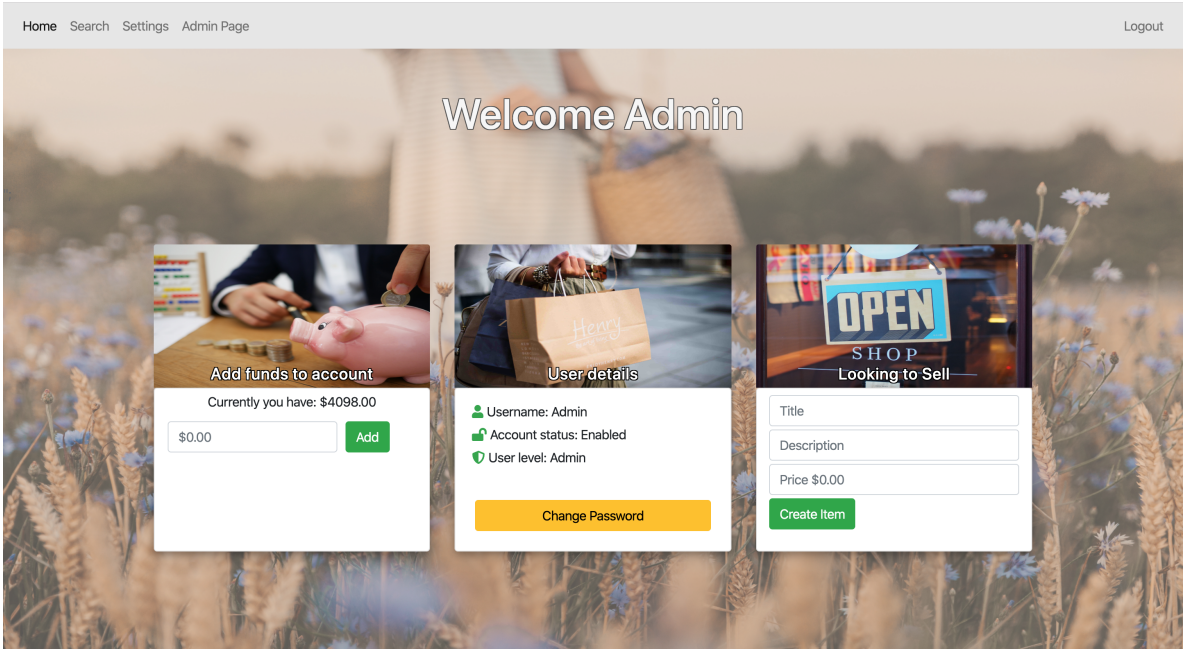

Figure 20: Home Page - welcomePage.php

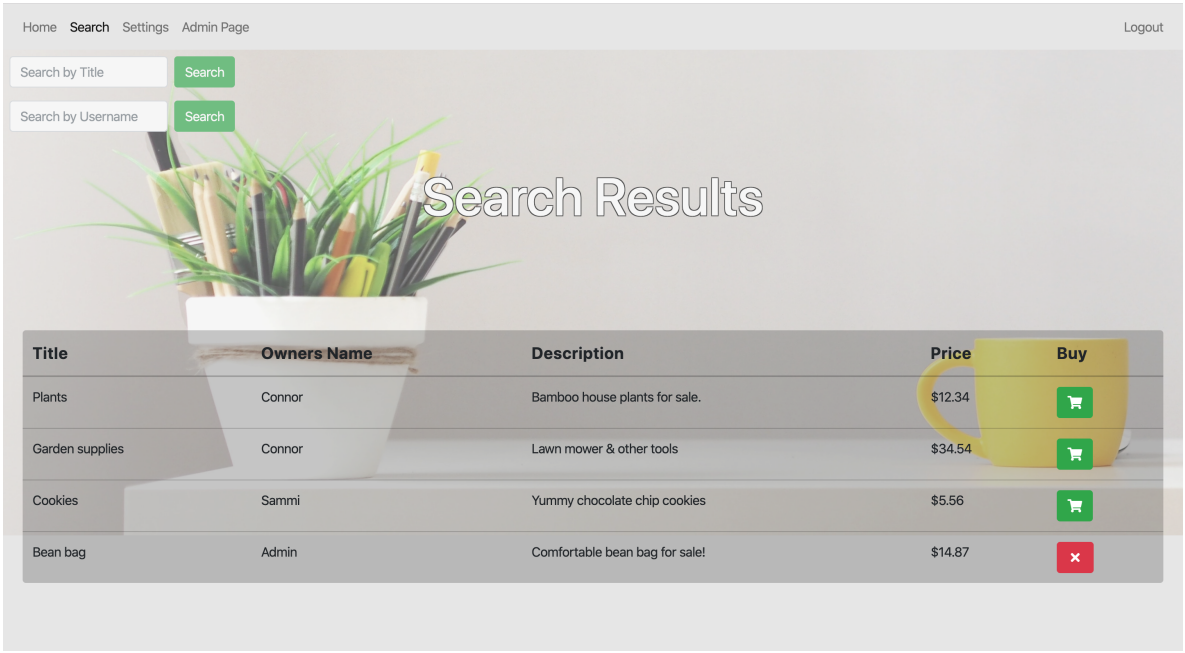

Figure 21: Search Page - searchPage.php

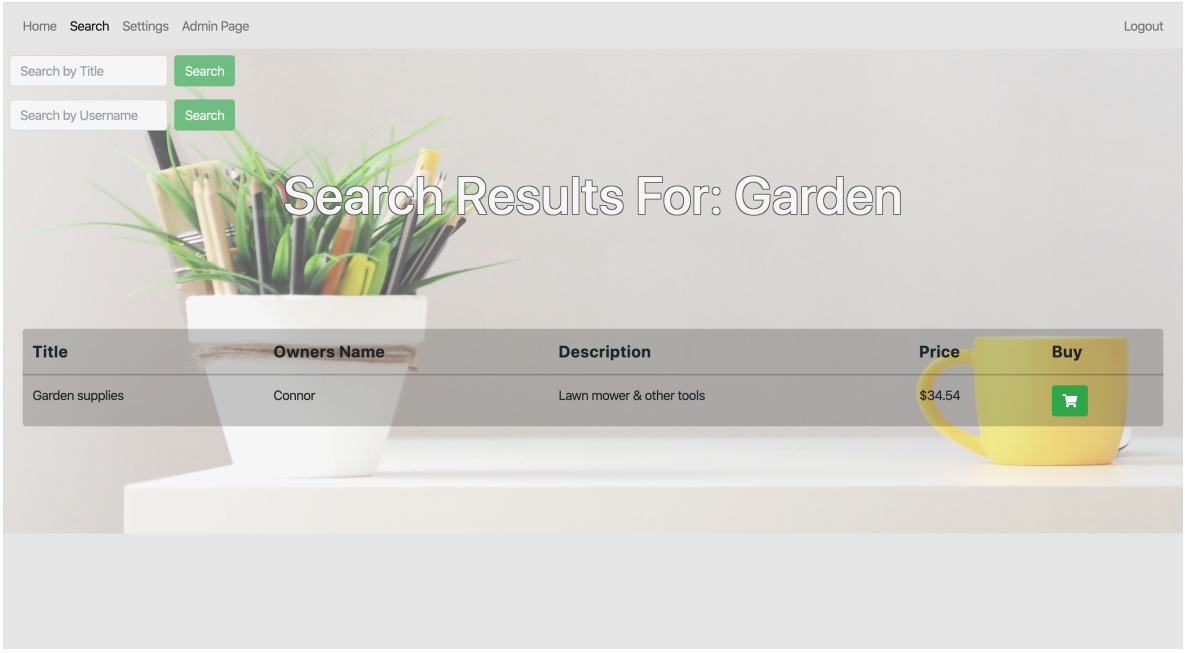

Figure 22: Search Page - searching by title

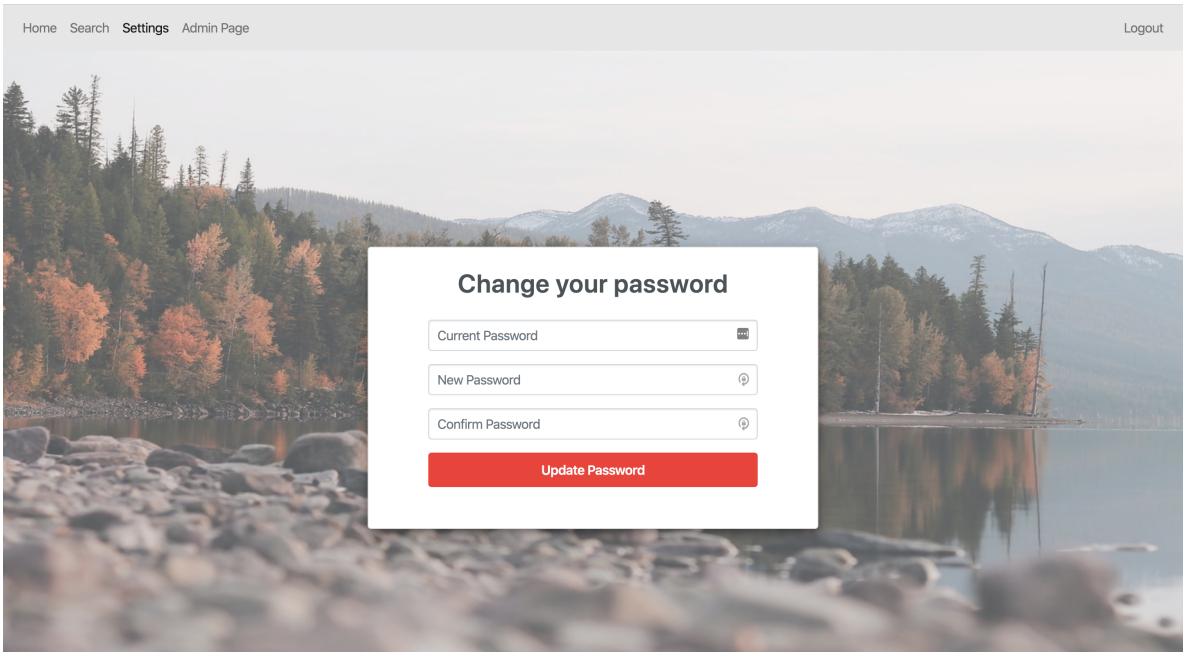

Figure 23: Settings Page - settings.php

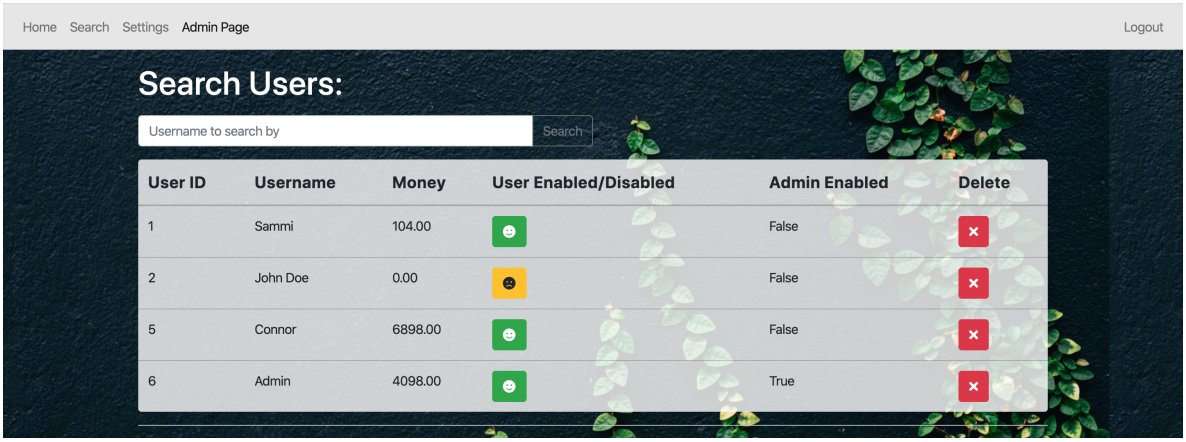

Figure 24: Admin Page - admin.php

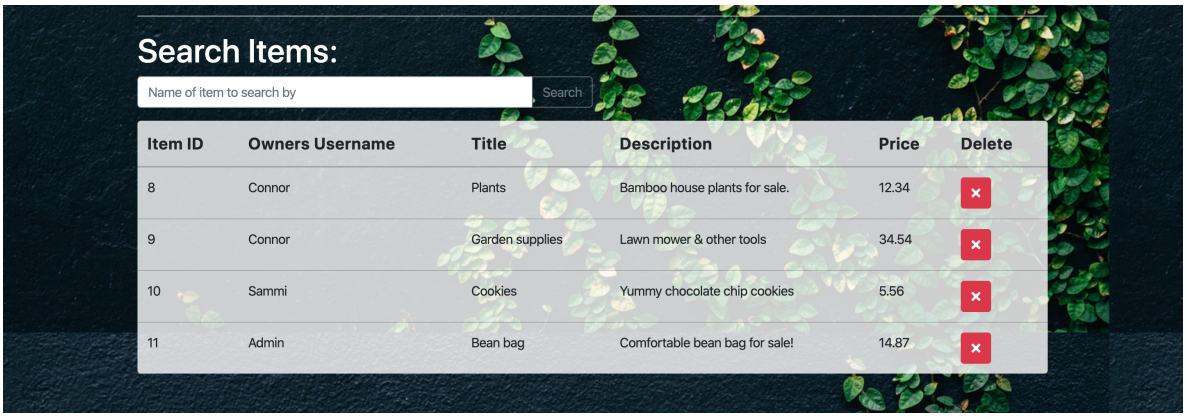

Figure 25: Admin Page Continue