**COMP205P Web Application Security Scenario Week Group Y Report**

by:

**Matt Policane**

**Yll Kelani**

**Fasbeer Eskander**

**Phoom Yenbamroong**

# The UCL Department of Computer Science

# December 6, 2016

## **Table of contents**

## 

[Table of contents](#_jtmkckvumu0c) 1

[1 Introduction](#_iptma0sf5bhe) 1

[1.1 Scenario week](#_9si1fzlz2ymj) 1

[1.2 Web security](#_i4k5tuorc9zl) 2

[2 Web application](#_kpyy48dqycg1) 2

[2.1 What does it do?](#_vc1xs5lg6mu) 2

[2.2 Architecture](#_cvqn35ygmka) 2

[3 Test environment](#_mj36u7nzhv9p) 2

[3.1 Test environment](#_fl0ggg9de8gt) 2

[3.2 Software configuration](#_dfr2kjbjus84) 2

[3.3 Vulnerabilities](#_5vyd7c4fol7z) 2

[4 Test plan](#_xw5nef470f4n) 3

[4.1 Password testing](#_ile8p48xp0i6) 3

[4.2 Injections testing](#_51k6hs5iklfc) 3

[4.3 Anything else?](#_s0abtv6a61df) 3

[4.4 How can these be repeated?](#_gfoqzeby1t5y) 3

[5 Vulnerabilities](#_mc5tzyqshu89) 3

[6 Tools](#_uve4rysvvofv) 3

[7 Points allocation](#_pctld0ttbibh) 4

[8 Conclusion](#_owi995biullp) 4

## 

## 

## 

## **1 Introduction**

### 1.1 Scenario week

As part of the COMP205P module, all students, in groups, are required to create two websites one which is secure and the other which is not. The purposes of this are to investigate a wide range of vulnerabilities that can come with websites and learn how to test for them as well as how to avoid them. Most of the vulnerabilities involved in the project are based from the OWASP top 10 list which clearly itemizes the most common vulnerabilities found in the websites on the Internet. After discovering some of these issues in our own system, we applied numerous strategies in order fix them before launching the second and more secure website.

### 1.2 Web security

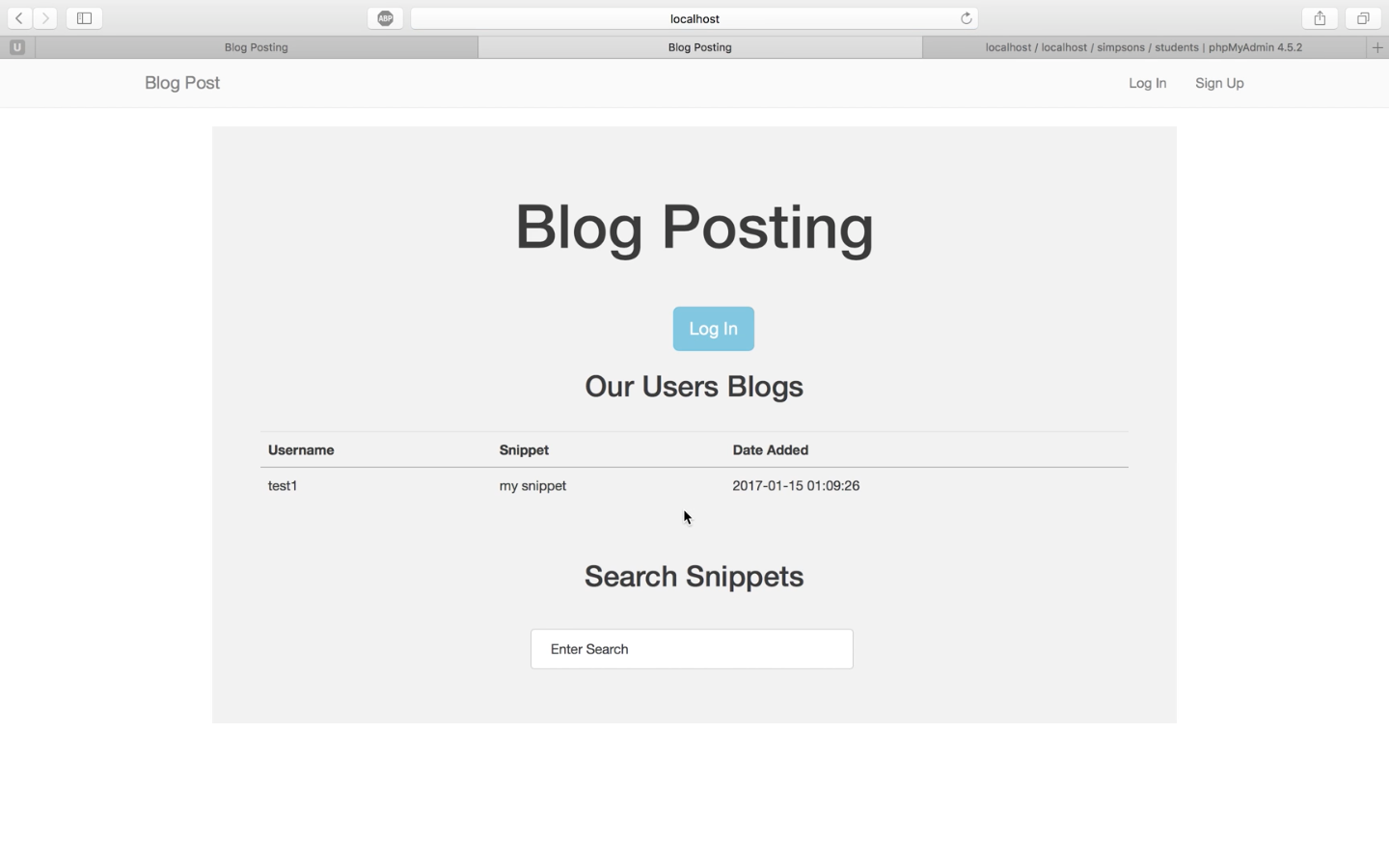
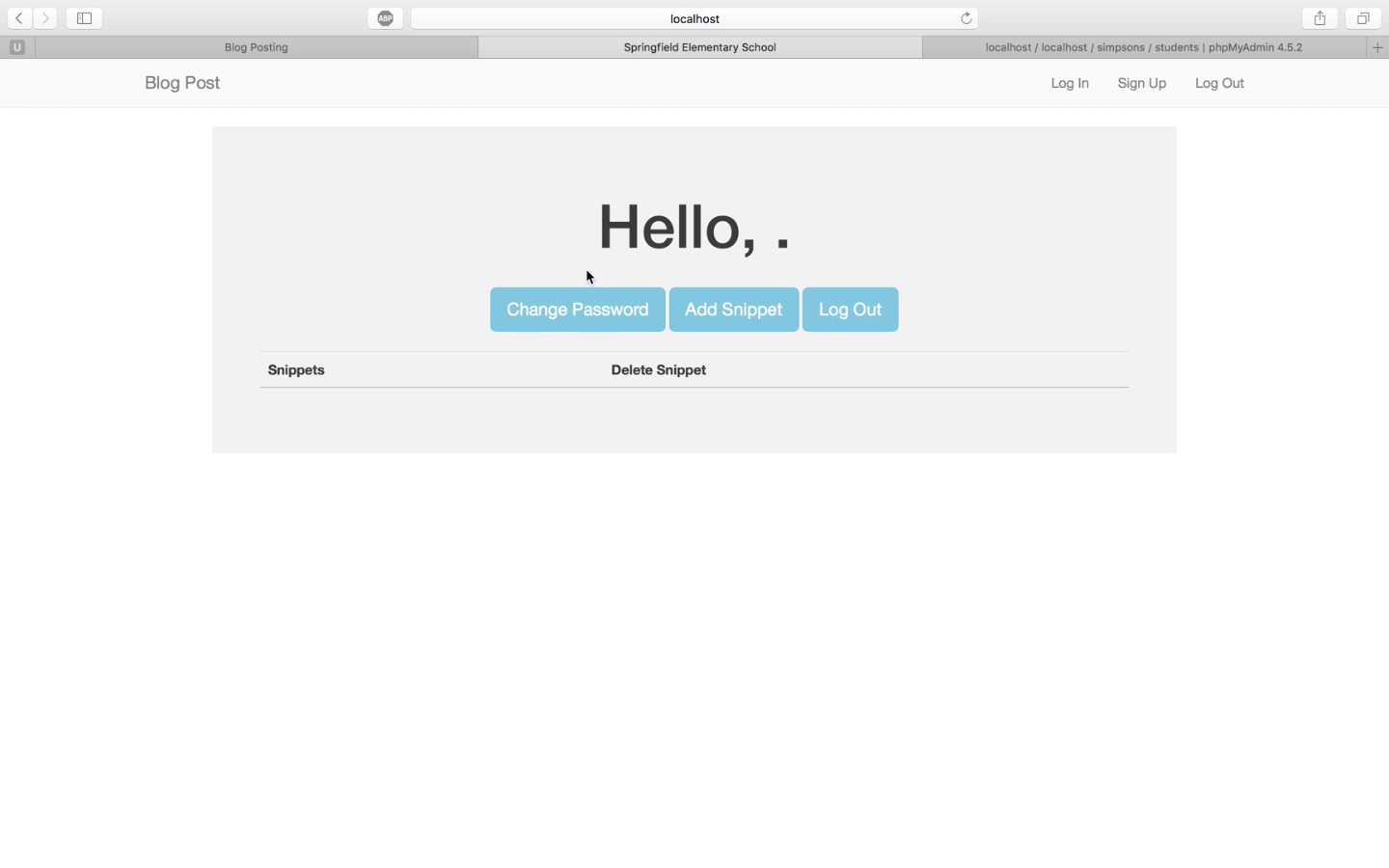
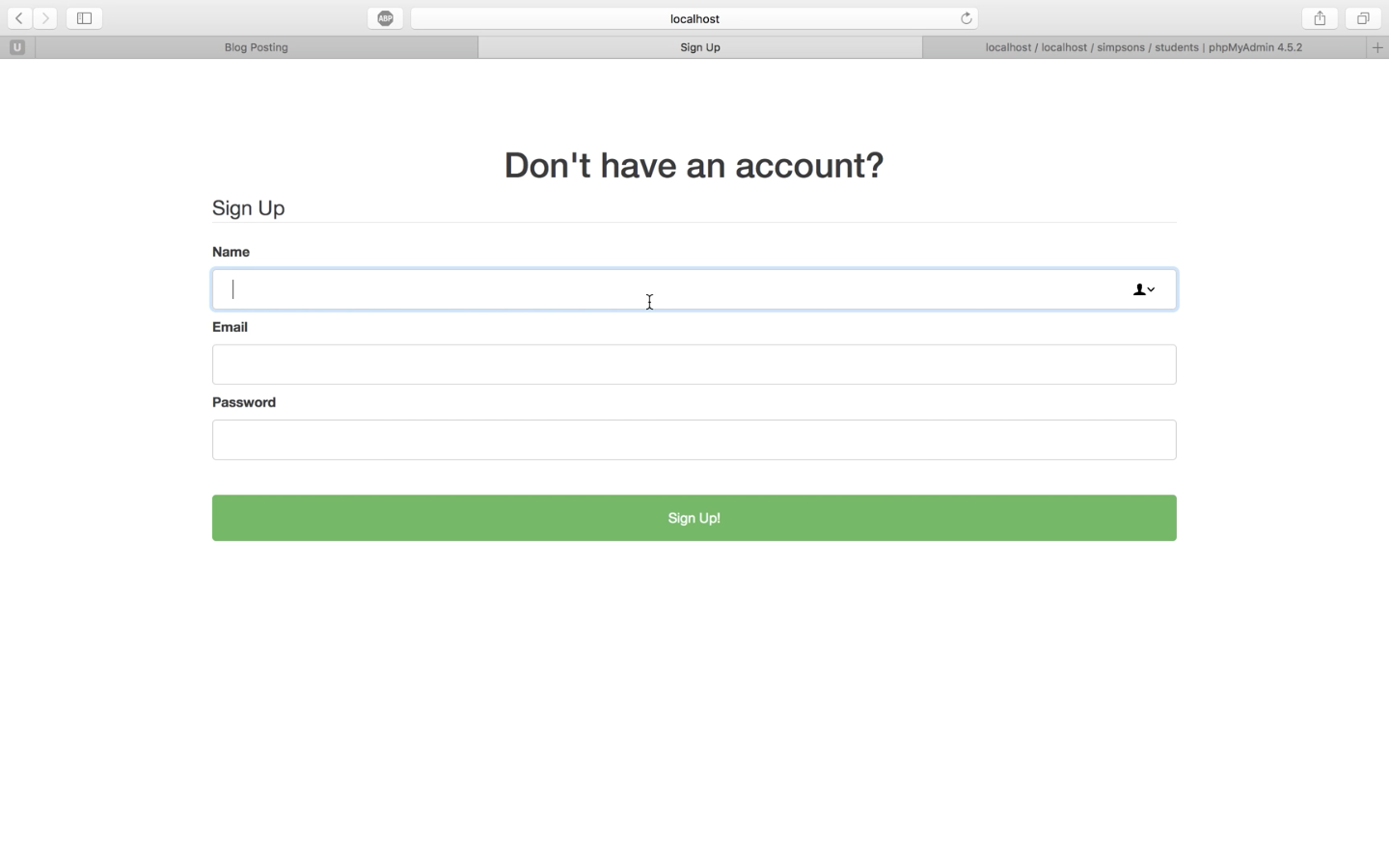
With the increased information sharing through social networking and higher business adoption of the Web as a means of doing business and delivering service, websites are often attacked directly. Attackers either seek to compromise the corporate network or the end-users accessing the website by subjecting them to drive-by downloading. As a result, businesses are paying increased attention to the security of their web applications. The majority of web application attacks occur through cross-site scripting (XSS) and SQL injection attacks which typically result from flawed coding, and failure to sanitize input to and output from the web application

In this report, all these common vulnerabilities will be covered and we will explain what we did to avoid them in the more secure web application. Moreover, a video has been made to highlight any important points further which can be found at <https://www.youtube.com/watch?v=wnio2dEOccw>.

**2 Web application**

### 2.1 Features

The website followed the template given on Moodle as a starting point. We then went on to further develop the application to allow our target users, students in this case, to log in and view their grades. Grades are stored in the databases which can be updated by users who have admin accounts. Teachers are assumed to have access to all information in the databases.



### 2.2 Architecture

We designed our app to follow the 3 part Model View Controller architecture. The model directly manages the data, logic, and rules of the application and in this case the data structure of our web app using backend system of phpmyadmin to control the sql database. The view is the use interface of our web app which shows the snippet information, the website CSS designs, and the table that is formed using HTML and JavaScript when adding and deleting snippets. And lastly the controller is the platform that controls and accepts user input such as logging in as a user or admin, signing up, adding and deleting snippet information from mouse click.

## **3 Test environment**

For testing our web app, a Kali Linux server was setup using Virtualbox by Oracle. The Debian operating system of 8.6.0 was loaded up using the i386 .iso file and Kali Linux version 2016.2 running on 64 bit was then installed with it. It used a base memory of 2GB running on core 2 processor booting from the Hard Drive.Then we installed Xampp version 3.2.2 and ran the apache and mysql servers to get the website up and running. OWASP Zap version 2.5.0 running on the VM was used to test the web app for vulnerabilities as well as a python password cracker to check for brute force password attacks. Some of the checks were automated using Burp Suite version 11.7.03 which checked for vulnerabilities while the website was browsed on the server. More tests were run manually to check if the vulnerability existed and was then corrected. This is the configuration that contained the site with the vulnerabilities and Xampp was then updated to a newer version to use PHP 7 instead of 5.4 .

The secure website was then uploaded using a web app service on Azure and then we integrated the MySQL database.

## **4 Test plan**

Security testing is a process intended to reveal flaws in the security mechanisms of an information system that protects data and maintains functionality as intended. Nowadays, businesses do not pay enough attention to application security which can often lead to disaster like loss of company’s sensitive data when vulnerabilities are exploited by a third party. Hence, security testing can greatly reduce the risk of this meaning that it is very important that all developers test for any potential vulnerabilities in their software before launching it to the public. We came up with a number of ways to test our website for any vulnerabilities and can be summarized as follows:

### 4.1 Penetration testing

We used penetration testing to help identify security vulnerabilities and possible threats within our web application. We focused on the area that the website was most at risk which was the authentication system. This was done through brute force attacks which used a password cracker developed in Python to try and crack the passwords in an attempt to evaluate how strong our password policy is. A different number of password dictionaries were used to optimise the chance of the success. Please note that this was done blind meaning no source codes or passwords were seen prior to the attempt. This could be easily repeated with other usernames by simply changing the username parameter passed to the cracker’s code to any username desired.

### 4.2 Injections testing

To test our system’s vulnerability to injections, we used OWASP Zap and manual testing with our own SQL statements. The website contains a search so that you can search for snippets by a specific user. In the vulnerable version, the search takes the input and appends it to an SQL query making the website vulnerable to injections. For example, if the user were to input “name or 1=1” , then they would get all the snippets, **including the ones that are private** - this is the issue.

We also used a tool developed by OWASP called OWASP Zap. We put in the url of our website and it “attacked” our website to expose vulnerabilities. After the “attack” was done, the tool provided a log of all the recognisable vulnerabilities. A vulnerability from SQL injections featured in the list and it specified the php file that processes the search input.

### 4.3 Other Tests

We did a lot of manual testing for vulnerabilities both within the OWASP top 10 and ones we’ve spotted ourselves. We did this testing during the development of the unsafe website and then after it was finished to aid us in creating the safe website. An example would be testing the flexibility of the sign up system. We discovered that a user was able to sign up to the website with blank details: username and password. This is a vulnerability as it would be easy for a hacker to guess that and then gain access to that user’s account. To amend this we enforce password policy that parses their input against a set of requirements.

We further did more tests using tools available on the Kali Linux server. We mainly used OWASP Zap to test for the vulnerabilities that arose and corrected them each time before testing again. Burp Suite was also used to check for other vulnerabilities and to automate the vulnerability checks while the website was being browsed on the server.

### 4.4 How can these be repeated?

The Testing can be repeated by setting up the test environment mentioned above and running the tests that we did both manually and using the tools available to us. For Brute Force attacks, the python password checker tool can be used with a dictionary of suggested words one by one to find and check for the password and if a password from the dictionary is valid and is given by the tool, it means that the database should be made more secure. For other tests, the OWASP Zap and Burp Suite tool available on Kali Linux can be used to check for the same vulnerabilities. Last but not least for manually checking the websites for vulnerabilities such as sql injection, the method written above to search 1=1 can be used. In this way, more web apps can be checked for vulnerabilities and then made more secure.

## 

## **5 Vulnerabilities**

After we carried out the testings mentioned above, many vulnerabilities were discovered as a result. They can be listed as follows.

### 5.1 Injection

The website has a search on the homepage so that you can search for snippets by specific users. This uses an SQL query to retrieve snippets from the database that were written by a username matching the search input. Each snippet has a column value “isPrivate” which states whether the user wants the snippet to remain private for only their viewing or to be available to all users on the website via the search. To ensure this, the query includes a clause in the statement that searches only snippets that have their “isPrivate” value set to 0. When I use the search normally the query works how we expected. But when I purposefully “inject” SQL into the search query with the right special characters, I can get the query to ignore the clauses specifying the “isPrivate” value and the search input so it returns a table full of every snippet in the database, including the ones that have been set to be private. To fix this we simply added single quotes around the parameter that is appended to the query. This makes it so the statement we use doesn’t work as the website itself recognises the hack attempt.

### 5.2 Broken authentication management

In the insecure website, user credentials can be easily guessed through weak account management functions. There were no restrictions to what the passwords could be meaning the password could be something as simple as “hello”. As a result, some of the passwords set in the website could be cracked by our Python password cracker. To tackle this, we imposed some password policies in the secure website. Each password is now required to be at the minimum length of 8 characters and has to contain at least one uppercase. This made the passwords less predictable and prevented our password cracker from cracking the passwords.

In the vulnerable website, the passwords were stored in plain text. This makes it easy for hackers to get usable information using MITM attacks. To guard against this, we hashed passwords when they’re created and store them in the database in that form.

### 5.3 Cross-site scripting (XSS)

The user supplied input was not validated before including them in the output page of the insecure website. Such input would then be treated as active content by the browser making the website use untrusted data in construction of HTML snippet without escaping. This way an attacker could send text-based attack scripts that exploit the interpreter in the browser. This would allow the attack to perform many acts such as executing scripts in a victim’s browser to steal sensitive data like user sessions or even redirect users to untrusted websites. We tested this by running scripts within the insert snippet input. We managed to get an alert to come up on the website and to redirect all users to google.co.uk when they added a snippet. HTML sanitisation was the solution to this vulnerability. We used the **sanitize\_text\_field** function in the secure website to validate all the user input. The function checks if the user supplied input contain any suspicious symbols that the field does not expect, if so the input will be denied immediately. This would prevent attackers from sending text-based and hence symbol-contained attacks via text fields.

### 5.4 Insecure direct object references

On the homepage there is a search that allows you to filter snippets in the database by username. To inject the search value into the php file that processes the search, we used a parameter in the URL. The parameter is then used in the SQL query to provide results from the search. This is a vulnerability as it allows the user to directly modify the parameter sent to the php file without using the intended input - the search. In OWASP terminology, the user can directly modify a parameter that refers to a system object, that refers to a system object (the SQL query) that the user is not authorized to modify. We fixed this by using POST to send the user input in the search bar to the php file. This means the only way the user has access to the parameter is via the search bar which is our intended means of input.

### 5.5 Security misconfiguration

In the insecure website, default accounts such as “user:test pass:test” remained in the database. This made the website vulnerable as hackers could guess obvious details like those and potentially gain admin privileges that they wouldn’t normally have. To fix this, we simply removed the default accounts from the database when we hosted the secure website.

We found it was possible to enter the specific path to a file on the server and you would be allowed to access that file regardless of authorisation. This is a form of force browsing where a hacker can just access the whole page via the directory listing. To fix this for the secure website, we added a variable that would change if the user was routed to the page from another page on the server. This made it so a hacker wouldn’t be able to access a page using the URL that wasn’t available access via interaction on the page.

### 5.6 Sensitive data exposure

In the vulnerable website, the passwords are stored in plain text in the database. This leaves the website very vulnerable to man-in-the-middle attacks as the password will be easily retrievable. Also, if a hacker were to get access to the database somehow, they would have free access to all the accounts on there as both the usernames and the passwords are stored as they are written when logging in on the website. To fix this, we hash and salt the passwords before they are stored. The passwords are hashed and stored so that they can’t be converted back to plain text. Salt is added to the passwords before they are hashed and then stored in the database under a separate column. Salt is a random set of bytes of any length. They make it harder for password crackers to guess passwords.

### 5.7 Missing function level access control

In the vulnerable website, if you type the exact url of the admin page ([oururl.com/admin.php](http://www.oururl.com/admin.php)), the admin page can be accessed regardless of if you are logged in or if you have an account with the required privileges to access this page. It gives you access to all the user accounts registered and gives you free reign to change all user details.

In the secure website, we overcame this problem by “killing” any php sessions that were accessed after having entered admin login credentials. We also stopped access via directory listing which mainly secured this vulnerability.

### 5.8 Cross-site request forgery (CSRF)

We used the Burp Suite tool and that showed us that our website was vulnerable to Cross-site request forgery. This was because we weren’t using any extra authentication to ensure the user is the one carrying out requests. We created a replica of our website which acted as a malicious website. If a user logs in and changes their password on the real website. The malicious website can then change the password without logging in again. This means that a hacker could potentially change a user’s password by getting them to interact with the malicious website and then steal their account. To protect against this, we added an authentication token that is sent with the change password request. If the token sent with the request matches the token given on login, the request is verified. If it doesn’t, the request is rejected.

### 5.9 Using Components with Known Vulnerabilities

By using PHP 7 instead of PHP 5.6 some of the past vulnerabilities were overcome. Attackers were able to remotely provoke memory errors in certain conditions and thereby execute malicious coding on servers on the previous versions of PHP and DoS attacks were also a result. These were improved and fixed to a large extent in PHP 7. According to the Common Vulnerabilities and Exposures (CVE) database, PHP averages between 20-25 vulnerabilities per year many of those impacting multiple PHP versions.

Sql also has some known vulnerabilities like Physical and network security; Attack surface, service accounts, and least privilege; Authentication, authorization, and SQL injection; and Disaster recovery and auditing. Among these we fixed SQL injection and authentication and improved network security. We added secure access to the database so attackers would not be able to access the information in it.

### 5.10 Unvalidated redirects and forwards

We avoided this vulnerability completely in that we rarely used URL to transfer information between PHP files. We generally used POST. This means that all redirects aren’t explicitly stated in the URL but are written in the code.

## **6 Tools**

### 6.1 Python password cracker

We wrote our own tool for password testing in python which simply works as follows. The program sends a request to the web server in order to gain access to the restricted webpages by checking a password of any inputted username in the dictionary one by one until it gets a response back from the server which grants access to it. It can then be concluded that the password which receives the response from the server is the correct one. Thus the security should be improved.

### 6.2 Kali Linux on VirtualBox.

As mentioned above, a test environment on Kali Linux was built to test for the vulnerabilities. OWASP Zap version 2.5.0 was used primarily to check for the errors on the webapp along with the manual checks. Vega was installed and used as well but the former software gave better results. Again, Burp Suite was used to automate checks for vulnerability on the website as it was browsed on the server with alerts of vulnerabilities being shown on the application. Burp Suite played a bit part in finding out CSRF attack threats. These informations were later used to make the web app more secure.

## **7 Points allocation**

### 7.1 Matt Policane

25 points

### 7.2 Yll Kelani

25 points

### 7.3 Fasbeer Eskander

25 points

### 7.4 Phoom Yenbamroong

25 points

## **8 Conclusion**

Many different skills were required throughout this project. For technologies, we had to use HTML, PHP, JavaScript and MySQL. This in collaboration with things like XAMPP and various testing tools. Not all of our team were familiar with all of them so we had to do a lot of work to learn them. Consequently, a substantial amount of research was necessary.

In this project, we learnt to develop a website and investigating it to identify any vulnerabilities. We learnt about a number of vulnerabilities that came up from the website, the different ways of using tools to check for vulnerabilities, to automate checks and also manually checking if they existed. The research we did then led us to learn how to fix those vulnerabilities.