Introduction
This project is a lottery website. A user may register an account and login. They may pick a set of 6 different numbers between 0 and 60. The user may add any number of sets, and check if their numbers match the winning numbers. A user may also view the sets of numbers they've chosen in a formatted table. Security features such as input validation, role base access control and others have been implemented within this project.

Approach
Data Input
User's input either login data or registration data via a html form. Client side input validation via Java Script occurs upon submission of creating an account. Embedded functions ensure user information is of correct format before submission. Client side checks are advantageous as incorrect data does not have to be sent back from the server. Reducing the amount of burden on the server[1].
Secondly both the login and register form are linked to a servlet filter. Information inputted by the user are searched for certain keywords. If a certain keyword is found the user is redirected to the error page. This is used to prevent and protect the database from SQL injections. Which prevents private user information from being stolen or modified and the database from being compromised.

Data Creation & Storage
User information is stored in a database as plaintext, except for the password. The user's password is mathematically converted into an unreadable string using a one-way function, a hash. For security reasons this is stored as the password. Storing the hash instead of plaintext is advantageous. Because if the database was compromised their password would not be in a useable format.
To log in, a user's inputted password is hashed and compared against the hash stored in the database.
Other user information, particularly the numbers they've chosen. As a security approach these are encrypted and stored within a text file. By storing specific user information as encrypted text this prevents unauthorised access to each of the user's numbers. I chose asymmetric encryption as this is considered more secure as 2 keys are used[2].

Data Access
Failing to log in after 3 attempts disables the login button. Each time the user fails the login form, a counter is increased. Once this counter is equal to 3, the html button gains a new attribute 'disabled'. The button can no longer be clicked to log in. This prevents a brute force attack via the index page. Where someone repeatedly inputs a different password until eventually guessing the correct password.
An admin user is permitted to view the accounts table, lottery numbers table and roll new lottery numbers. However it is important that these 3 features are only available to admins. Each of these admin features are mapped to a servlet filter. This filter performs a check for an admin attribute in the session object. If the user is an admin the request chain is continued on as normal executing the admin feature. Otherwise the user is redirected to the error page, which only has a link to the home page. This form of role base access control prevents unauthorised access to certain features. Which also protects personal information as this is viewable by an admin.

Problem Solving
To solve the problem of input validation using Java Script. I initially researched this topic that was new to me. Leading me to various blog posts, tutorials and websites. I decided to decompose this problem into various subproblems. Which were creating check functions for each necessary field. Researching relevant syntax and how to acquire data from the form solved these problems. Another subproblem was executing the validation to affect submission. More research into this specific topic and testing implementations led me to a solution. Which was to link the validation function at the top of the form.

Public and private keys are created and stored as objects. These are used to decrypt and display the current user's numbers. My problem was, sometimes the decryption would error, resulting in a table of unreadable text. However, other times the user's number were decrypted correctly. Firstly I displayed information about the keys in the console. I noticed upon creation of the private key it's type was 'SunRSASign CRT'. Note the private key object data is also stored in a text file for later extraction. Upon extraction the key is casted to 'Sun RSA'. These types are not the same resulting in completely different keys which was causing the error. I solved this by casting the private key during creation to the same type when extracted (SunRSASign CRT to Sun RSA). This no longer resulted in any decryption errors.

I configured the project to run using only docker commands. However after executing the project from the command line. The webpage returned an error stating the war file was not found. Ensuring all references to the war file were correct did not fix my problem. I proceeded to build the war file multiple times under many different conditions. Such as modifying the contents of the 'src' folder. This also did not work. I then consulted another student who gave me some advice. To download a new copy of the coursework, place all of my files in that copy, and then build an artifact. To my surprise this actually worked, opening the webpage no longer resulted in an error.

Testing

Testing Input

My testing strategy for input into the html fields was to input values that were valid, extreme, and invalid, where possible. This was to ensure fields for registering an account accepted the input stated in the requirement. But also rejected any input that did not meet the specified requirement or specified to be rejected. This also included inputting the keywords prevented by the ServletFilter to prevent SQL injections. To again, ensure the servlet filter was performing as expected.

Testing Hash Creation and Storage

To ensure a hash was being created from an imported function I displayed the return value. From observation I was able to confirm a hash was created. A requirement was to use the hashed password for login verification. To ensure this was possible, two identical strings of length 12 were hashed and compared to determine whether the hashes were equal. The resultant hashes were equal, which ensured the hashing algorithm could be used for login verification.
Confirming the hash was being stored in the database as opposed to the password. I verified this from observation of the database password columns. As all strings under that column began with '$2a$10' which is a part of the hashing algorithm I used.

Testing Encryption and Decryption

Numerous strings were encrypted, written to a file, then read back out into an array. Each string in the array was then decrypted and outputted original values. This ensured the public and private keys were being used correctly. To reduce waiting time and to simplify the testing process, I first ran these tests in a separate class. To also ensure that the user logged in, could only modify their own numbers. I logged into separate accounts to add and view numbers. Different accounts yielded different tables of sets. Ensuring the encryption and decryption requirements were met.

Recommendations

As stated in the requirements users are limited to 3 logins before disabling the login button. However, it is also noted 'that it is enough to just disable the form which can be reactivated'. For real world applications this may not suffice. As this may not deter brute force attacks. A solution could be to lock down the account that someone has failed multiple times to log into. However, this may cause a denial of service attack. Someone may purposely fail to log into many accounts to lock them. Therefore I would recommend to implement a security question. Along with a CAPTCHA. The user must answer a personal question and complete a test to determine if they are human.

As stated in the requirements a servlet filter is used to prevent SQL injections. However, this only extends to the create account section. I would recommend extending this to the login section. As an SQL statement is executed to verify logging in. Users could potentially bypass, modify or expose the contents of the database. By performing an SQL injection on the login form. I chose to implement my recommendation by mapping the SQL injection filter to the user login. Furthermore the filter works by searching for certain keywords. However, the list of keywords does not include 'drop' and 'table'. This means that potentially a user could delete the contents of the database. As these keywords were not required to be filtered. I decided to also include these keywords to be filtered.

Preventing data leakage was not mentioned in the requirements. This means to prevent error messages presenting program information to the user. This program information can possibly be sensitive data about the structure of the program itself. Leading to more focused attacks. This can be prevented by thoroughly testing interactable components to ensure they do not error. Another method could be to configure the application's environment that prevents errors from being displayed[3]. I partially implemented this by preventing any errors I encountered. For any null pointer exceptions I implemented a check for null before further using the data or object.

I would recommend implementing a security measure against path traversal attacks. This type of attack entails accessing files outside the web root folder without authorization. By manipulating the URL to include special sequences such as '../'[4]. An attacker may acquire personal data. I personally found, that a non-admin user may add '/admin/admin_home.jsp' to the URL. Which grants unauthorised access to the admin page. I would recommend using web server access control capabilities to prevent attackers from directly requesting files[5].

Reflection

What Worked

Indicating whether a user is an admin by setting and storing a bit in the database. I found this straightforward to implement. Upon account creation a user simply ticks a box. A series of simple conversions take place to store this in the database as a bit. Upon logging in, the value of the bit is measured to determine whether a user is an admin or not.

Expanding the database structure to feature another table I found simple to implement. Quick research into MySQL syntax for creating tables and choosing appropriate data types worked as intended. Also creating my own SQL statements to query the new table was quite simple. As I recalled my experience with databases from my first year modules.

What Didn't Work

My first implementation for, disabling the login button after 3 failed login attempts. I stored a boolean indicating the user failed to login within the session object in Java. I then expected that the webpage's session storage stored the same attribute. Which it did not, the disabling of the login button would not occur. As I was checking for an attribute that did not exist within the webpage session storage. My initial assumption was incorrect.

What I Learned

I learnt that a servlet is a class used to extend the capabilities of servers that host applications accessed by means of a request-response model[6]. Moreover a servlet filter can be used to intercept and manipulate requests to perform security measures. Such as preventing certain keywords that could trigger an SQL injection.

Another interesting topic I learnt was hashing passwords. Hashes are stored instead of plaintext passwords. Because if the database was compromised the passwords would not be readable or useable. Someone would have to implement a valid rainbow table. A precomputed dictionary of plaintext passwords and their corresponding hash values[7]. This research led me to implement BCrypt for hashing passwords as it includes a 'salt' which prevents rainbow table attacks. Although BCrypt was not necessary for this project, I still found this really interesting to learn and develop with.

What I Found Hard

Prior to this coursework I had no experience developing with Java Server Pages and Servlets. I found overall research the most challenging. As I did not know exactly what to research. Such as how to link a JavaScript function to a html form. Another example, was mapping a servlet filter to a servlet class. I also had no experience with tomcat and docker. Initially setting these tools up and then attempting to use them was difficult. However, understanding the purpose of these tools within the project. By researching them, helped me become more familiar with them.

References

1. Tutorialspoint.com. n.d. *Javascript - Form Validation - Tutorialspoint*. [online] Available at: https://www.tutorialspoint.com/javascript/javascript_form_validations.htm [Accessed 28 November 2020]

2. Yedakula, K., 2019. *Exploring The Differences Between Symmetric And Asymmetric Encryption | Cyware Hacker News*. [online] cyware-social-nuxt. Available at: https://cyware.com/news/exploring-the-differences-between-symmetric-and-asymmetric-encryption-8de86e8a [Accessed 28 November 2020].

3. Cwe.mitre.org. 2020. *CWE - CWE-211: Externally-Generated Error Message Containing Sensitive Information (4.2)*. [online] Available at: https://cwe.mitre.org/data/definitions/211.html [Accessed 28 November 2020].

4. Owasp.org. 2019. *Path Traversal | OWASP*. [online] Available at: https://owasp.org/www-community/attacks/Path_Traversal [Accessed 29 November 2020].

5. Cwe.mitre.org. 2006. *CWE - CWE-22: Improper Limitation Of A Pathname To A Restricted Directory ('Path Traversal') (4.2)*. [online] Available at: https://cwe.mitre.org/data/definitions/22.html [Accessed 29 November 2020].

6. Docs.oracle.com. 2010. *What Is A Servlet? - The Java EE 5 Tutorial*. [online] Available at: https://docs.oracle.com/javaee/5/tutorial/doc/bnafe.html#:~:text=A%20servlet%20is%20a%20Java,applications%20hosted%20by%20web%20servers [Accessed 27 November 2020].

7. GeeksforGeeks. 2018. *Understanding Rainbow Table Attack - Geeksforgeeks*. [online] Available at: https://www.geeksforgeeks.org/understanding-rainbow-table-attack/ [Accessed 27 November 2020].