University of Springfield

University Web Application Vulnerability Patching

By Andrew Lambert (16355706), Jem Cairns (16439124), Konstantinos Ntafloukas (19203806)

Github Repository: https://github.com/JemCairns/UNI_SYSTEM

Introduction

This report details the vulnerabilities the other team detected and the proposed ones by the coordinator, based on the OWASP Top-10, their location in the code, the measures we took to mitigate them and the results. These sections included Insufficient Monitoring and Logging, Security Misconfiguration, Broken Access Control, Sensitive Data Exposure and Broken Authentication.

The below steps are required to run the web application (also provided in the GitHub README):

- Open a MySQL terminal and run all SQL code within the sql_script.sql file in the Github repository.

- Enable Chrome to accept a Self-Signed Certificate on localhost
    - Open a new Chrome tab.
    - Enter the following into the address bar: chrome://flags/#allow-insecure-localhost and enable the highlighted option.
    - Click the three dots in the top-right hand corner of another tab.
    - Click on Settings.
    - Under the “Privacy and security” section, click on Manage certificates.
    - Under the “Your certificates” tab, click the Import button.
    - Find the uni_system.p12 under the keystore folder in the project.
    - Enter the password “MyBigHappyKeystore!” into the appropriate field.

- Run the web application and navigate to https://localhost:8443/login in your chrome browser.

Insufficient Logging and Monitoring – OWASP A10

Weaknesses

- The application did not log any events that happen within the system (login, payment of fees, modification of module data). As a result, if an attacker were to exploit this system their movements would go undetected and they could not be identified or tracked.

Mitigation

- Events that should have been logged include user logins and registrations (both failed and successful), the payment of fees, the editing of module information by a staff member (details and grades) and the enrolling and un-enrolling of a module by a student. These were fixed in the below location by implementing the "com.javacodegeeks.snippets.core" java logger. This was created using a Singleton pattern in the Logging.java class, and is referenced (used) in the below java classes:

  login.LoginController.java, register.RegisterController.java, grades.GradesService.java, edit_modules.EditModulesService.java, modules.ModulesController.java

Results:

- All loggings of this web application are both shown in the console and placed in a default.log file, providing the date and time, method, class and message of each logging along with the IP address of the current user for logins and registrations. For example, a failed user attempt is logged as per below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2020-05-15T12:15:28.992875Z</date>
  <millis>1589544928992</millis>
  <nanos>875000</nanos>
  <sequence>4</sequence>
  <logger>com.javacodegeeks.snippets.core</logger>
  <level>INFO</level>
  <class>uni.system.webapp.security.LoginAttemptService</class>
  <method>loginFail</method>
  <thread>49</thread>
  <message>User with IP=0:0:0:0:0:0:0:1 failed to log in.</message>
</record>
```

Security Misconfiguration – OWASP A6

Weaknesses

- Password Requirements

  The first security misconfiguration of the original web application involved allowing users to register to the website with weak passwords. This allowed users to register without a strong password, allowing users with popular passwords to have their accounts hacked quite easily.

- Error Handling

  In three instances of this application a command could be entered that produced a Whitelabel Error Page. These instances were simply the result of server-side error handling (null values in particular) and were fixed as per below.

Mitigation

- Password Requirements

  The allowing of weak passwords was mitigated by providing the user with a password strength-meter ranging from Weakest -> Weak -> Okay -> Good -> Strong. The password strength was then accessed from the REST registration controller, and if the password strength was less than "Good" the user's registration request was denied. These changes were applied to register.RegisterController.java along with the register.html file.

- Error Handling

  The three instances or error handling issues occurred when users entered a null amount for paying fees, a null amount for submitting grades and registered with a student id longer than seven digits long. Both null value errors were fixed by passing the amounts to the REST controllers as Strings instead of doubles (the HTML treated these null values as Strings) and parsing to doubles in the controller, while the student id error involved simply changing a < sign to a != sign for checking the number of digits in a student number. These errors were handled in the register.RegisterController.java, fees.FeesController.java and grades.GradesController.java classes.

Results:

- Password Requirements

  The enforcement of selecting a strong password in the web application not only ensures that an attack on user passwords will both be more difficult and take a longer period of time but also entices users to create a new password instead of re-using another password and avoiding their account being accessed due to a lack of security in other websites where this password was used.

- Error Handling

  As the above Whitelabel errors were fixed, the availability of the web application was increased and thus users are provided with a more cohesive experience along with zero website errors causing their login session to fail.

Broken Access Control – OWASP A5

Weaknesses

- Student Having Access to Staff Functionality

  This vulnerability was described within the vulnerability assessment of this web application. It involved a student gaining access to features meant only for staff members. This occurs when a staff user logs in and out again. Since the application used the same JSESSIONID, should a student then login and try to access a staff only page (E.G. /grades, /edit_module) they gain access to the page meant for the staff member who was just logged in. This vulnerability is brought on by incorrectly implementing session management and not authorising requests to ensure that only those with correct privileges can access restricted pages.

Mitigation

- Student Having Access to Staff Functionality - Session Management & Authorisation

  The main fix for this vulnerability was proper session management. Creating unique, random IDs or tokens for each user ensures that their activity is secure and cannot be controlled by an attacker. The implementation of proper session management is described in more detail in the Broken Authentication section.

  Another security control put in place was creating roles for each user and using them to dictate which requests users with a particular role could perform. A new table was created to hold these new User objects. From here, within the *SecurityConfig.java* file, certain URLs were configured to be only accessible by the correct users (E.G. .antMatchers("/grades").hasRole("STAFF")). This implementation also relies on the unique tokens created for session management. A user's data is stored within these tokens, including their role, and is retrieved and compared whenever a new request is made by that particular user. Using the configuration of the *JWTAuthorFilter* class, this allows for this comparison to be made for every request.

Results:

- Student Having Access to Staff Functionality - Session Management & Authorisation

  By including this functionality within the web application, it is protected against this Broken Access Control vulnerability. With proper session management, the unique token for a staff member's activity is invalidated. This means that it cannot be used by a student to escalate their privileges and gain access to staff functionality. Even if a student did try to access the restricted pages, their unique token includes their role of STUDENT. Should they try to access the /grades page, they will be sent a 401 Unauthorised response, disallowing them access to the page. This way, only students can access student functionality, and only staff can access staff functionality.

Sensitive Data Exposure – OWASP A3

Weaknesses

- Insecure Password Storage

  One key vulnerability produced in this web application was that user passwords were stored in plain text. This means that any administrator with viewing access to the database can record a user's id and password, log in to the application as the given user and edit their details. This is a trivial error and can simply be fixed by encrypting user passwords.

- Insecure Channel for Data in Transit

  One of the weaknesses that existed in the web application under Sensitive Data Exposure was that data was not secured in transit. An attacker had the ability to perform packet sniffing on data being sent between the app & the client. This would lead them to gain access to information like the user's ID, password, and user type (student/ staff). This vulnerability was present in the web app's server so affected the entire application.

Mitigation

- Insecure Password Storage - Salted Hash Encryption

  This vulnerability was mitigated by implementing salted hash encryption on each user's password when entered while storing the user's salt as an extra column in the student and staff tables. To verify users, their passwords are re-encrypted using the given user's stored salt and compared as encrypted. All encryption is completed server-side, while the appropriate responses are then sent to the given user. All encryption functionality is created in the security.PasswordUtils.java file while this code is implemented in the register.RegisterService.java and security.uniPasswordEncoder.java files.

- Insecure Channel for Data in Transit - HTTPS

  The way we prevented this vulnerability was by using end to end encrypted communication from the web application to a client. This was done by enforcing the web application to send data via HTTPS. This is an extension of HTTP in which the communication protocol is encrypted by TLS. TLS is a cryptographic protocol that provides security to data sent over a network.

  Within the *ServerConfig.java* file are a set of rules that ensures that HTTPS is used to send any data to/ from the client. There is also functionality that if the web application were to be accessed from its old address (localhost:8080) using HTTP, this traffic is forwarded to a new secure address (https://localhost:8443). For this to work, a self-signed certificate was created to tell the browser that the web application is indeed secure.

Results

- ### Insecure Password Storage - Salted Hash Encryption

  The encryption of password both increases the difficulty level for an attacker to retrieve a user's password along with ensuring that user accounts cannot be accessed or manipulated by administrators with database viewing privileges. This is also in compliance with GDPR regulations ensuring the secure storage of user data.

- ### Insecure Channel for Data in Transit - HTTPS

  With these new security measures in place, packet sniffing cannot be performed. When received the data is encrypted into a non-human readable format. While this data can be decrypted using attacks such as Man in The Middle attacks, these sort of attacks and decryption methods are very complicated and require a lot of time to result in anything of interest. SO the use of HTTPS to encrypt the communication channel both secures any sensitive data and deters threat actors from trying to decrypt the data due to its cryptographic complexity.

Broken Authentication – OWASP A2

Weaknesses

- #### Brute Force Attacks

  In the first iteration of the web application, there are no security measures in place that can help defend against a Brute Force Attack (BFA). A BFA is where an attacker will either create a script to try possible, common usernames and passwords or execute this manually. While these types of attacks are computationally expensive, they would work eventually given the correct guess of a user's login credentials. This vulnerability lies within the login service of the web application.

- #### Registration Vulnerabilities

  The initial registration functionality of this web application involved very few checks to ensure that users were entering valid details. This allowed users to register without a university email or a legitimate phone number. Not only did this make hacking into user accounts easier, but also increased the ability for attackers to create fake accounts themselves and by writing scripts to create multiple fake users.

- #### Session Hijacking

  This vulnerability arose from the fact that proper session management and communication channel encryption were not implemented. This meant that using the same packet sniffing attack as discussed in the Sensitive Data Exposure section, an attacker could read an HTTP request's Cookie. Here the JSESSIONID would be found that was created by Spring Boot's TomCat server by default. Through manipulation of a request to the web app & adding in the Cookie pair, a legitimate user's session could be taken over by the attacker. Again, since the web application never encrypted any data that was sent, these JESSIONIDs could be read in plaintext, allowing it to be easily obtained. The session management issue lies mostly within the functionality for logging in. Here, the user's entered details are merely checked against the database and no unique session token or cookie is created to prevent this identifier from being used elsewhere.

Mitigation

- #### Registration Vulnerabilities - Captcha & Input Checks

  Most registration vulnerability mitigations simply involved adding extra input checks while providing the users with very little feedback pointing to the error and thus not disclosing implementation aspects of the web application. These involved checking that the user's id was seven digits long, the email ended with "@springfield.com", the password was of good strength and phone number was ten digits long. A captcha was also added to the registration page to prevent attackers from registering using coding scripts. This captcha functionality involved creating a google captcha account while providing captcha functionality in the

captcha.ReCaptchaResponse.java class which is implemented in the register.RegisterController class along with the additional registration checks.

- Brute Force Attack - IP Blacklisting

To help prevent Brute Force Attacks, IP blacklisting was added to the web application's login functionality. This system records a user's IP address when they attempt to log in. If they succeed then nothing happens, but they fail, then their IP is given a strike. If they retrieve three strikes then the IP address is blocked for 24hrs. Since the IP address of a computer can change over time, this ensures that a legitimate user will not be blocked out of the service forever.

Using a Cache that maps IP addresses to the number of strikes, the web application can keep track of all attempts to log in. There are two methods within the *LoginAttemptService* class which implement the functions for a success or a fail. Depending on if the authentication is successful or not, either the loginSuccess(..) or the loginFail(..) method is called. If it fails, then the Cache mapping for the specific IP is incremented, and if it succeeds then the IP mapping is invalidated.

- Session Hijacking - Session Management

Through proper session management, vulnerabilities such as session hijacking and the web application not creating unique IDs could be removed. This was achieved with the use of JSON Web Tokens (JWTs) and the vast array of features that Spring Security provides. For this to be implemented, several files and configurations were created. The main file is *SecurityConfig.java*. The configures such as which URLs require a specific authorisation role, how the login and logout pages should be handled, and how authentication is performed.

JWTs were used as the unique identifiers for a user's session. They provided a way to create values that were unique, random, and long enough to ensure that even if a user cannot see the token because of HTTPS, it cannot be easily guessed either. Once a user attempted to log in, the *JWTAuthenFilter* class would control if the authentication was successful or not. If it was, a new JWT would be created with the user's details incorporated into it. Then this would be added as a cookie for the remainder of the user's time on the web app. This cookie would serve as the user's ID for every request they make. Since it housed their specific details, such as their role, it would forbid them from accessing URLs that they had no need for, E.G. A user should not access the /edit_module page.

Session timeout was also incorporated into these JWts too. The token will be invalid an hour after it is created. This is to reduce the time that an attacker might have to cause harm to the user, should their unique token be exposed or released. All of these implementations help protect a user's time while they are using the web application.

Results:

- ### Brute Force Attack - IP Blacklisting

  Since IP blacklisting was included in the authentication process, this prevents BFAs from being performed on the website. If a single source fails to log in three times in a row then they are blocked from the app for 24 hours. Defending against such a common and automated attack helps to protect the web application's users from low-level hackers who want to exploit the app. This kind of harsh punishment also ensures that users don't accidentally trigger this event too. Being locked out from a computer for 24 hours is a very long time for a legitimate user and helps to teach them to be mindful of their login attempts.

- ### Registration Vulnerabilities - Captcha & Input Checks

  The results of patching these vulnerabilities mainly involve properly authenticating users before registering for the web application, while also preventing attackers from creating dummy accounts using fuzzy-scripts and thus increasing the difficulty for users to gain access to the web application.

- ### Session Hijacking - Session Management

  The implemented security control is effective as it creates a unique identifier attached to a particular user and the activities they perform that cannot be tampered with or stolen. This allows a user to traverse the web app without having to log in or authenticate/ authorise themselves on every request. With these tokens, an attacker cannot easily guess or swap out their own token to gain access to another user's current session. This, along with tokens having an expiration date, a user's ID is made secure and cannot be used to assume their identity.