**EE4023 Distributed Systems – Group Project**

**Colm O’Brien**

**Jack Morrison**

**William O’Dwyer**

**Brian Kinane**

**Eoin O’Sullivan**

**Paul O’Sullivan**

**Report on used techniques to secure application:**

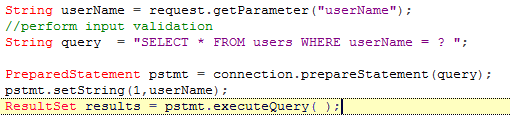
Injection:

Injection is tricking an application into including unintended commands in the data sent to an interpreter. Interpreters take strings and interpret them as commands. SQL injection is severe can usually results in the attacker reading and even modifying an entire Database.

The first step we took to avoid injection, was validating user inputted information to ensure they only input what we need/want them to input.

We then used Prepared Statements with Parameterised Queries. This allows the database to distinguish between code and data, regardless of user supplied input. Prepared statements make sure that the attacker is unable to change the intent of an SQL query. If the attacker entered a string like Colm’ or ‘1’ = 1’ the parameterised query would search for a username that matched the exact string instead of executing is an SQL.

Prepared Statement example:



Broken authentication and Session Management:

HTTP is a stateless protocol which means all log in credentials have to go with every request. To avoid this we used SSL (Secure Socket Layer) for authentication. Applications that have weak session management typically have the session ID clearly visible in the URL. This can easily be hijacked and the attacker can use the session ID to steal the innocent user’s identity.

To further protect our application with regard to our user authentication, we set pre-requisites for user information, such as:

* Passwords should have restrictions that require a minimum size and complexity.
* Users should be limited to a certain amount of log in attempts per unit of time.
* Password must be stored in encrypted or hashed form.

Cross Site Scripting (XSS):

Cross Site Scripting attacks are a type of injection, where malicious scripts are injected into trusted web sites. Flaws that make web applications susceptible to XSS attacks occur anywhere a web application directly outputs user input without validating or encoding it. The end users browser has no way of knowing whether a script should be trusted or not and so will execute it. The malicious script can access cookie information, session tokens or other sensitive information.

We made sure that we were not directly outputting and data that was inputted by the user without validating it first. We also used the Content Security Policy (CSP) to list our trusted sources so that even if the attacked managed to inject a malicious script, the browser would not execute the script if it did not come from a trusted source.

Insecure Direct Object Reference:

Insecure Direct Object Reference vulnerabilities are part of enforcing proper authorisation. Common mistakes include: only listing the “authorized” objects for the current user or hiding the object references in hidden fields and then not enforcing these restrictions on the server side.

We used the OWASP Enterprise Security API (ESAPI). The use of the ESAPI conveniently provides a mapping from a set of indirect references to the direct references, so convenient that you can even use a different mapping upon each request.

Missing function level access control:

For function, a site needs to: Restrict access to unauthorized users, enforce any user/role based permissions and completely disallow requests to unauthorized pages. Many applications hide direct access to certain functions but do not actually restrict access. Attackers can easily find other ways of forcing access. For example by looking at the URL:

http://example.com/app/getappInfo  
<http://example.com/app/admin_getappInfo>

The attacker can spot that by changing the URL he can potentially gain access to unauthorized functions or pages.

To prevent this is our application, we used the ESAPI function isAuthorized() to check if the user has access to page or function they are trying to access. We implemented this security check anywhere there are private pages or functions that require permissions such as any admin activity or editing other user’s information.

Cross Site Request Forgery:

Cross Site Request Forgery (CSRF) is an attack whereby a malicious website will send a request to a web application that a user is authenticated against, from a different website. CSRF attacks target functionality that causes a state change on the server, such as changing the victim’s user name or password or even purchasing something.

To prevent this risk in our application, we appended unpredictable challenge tokens to each request and associate them with the user’s session. Tokens are long cryptographic values that are difficult to guess. These tokens are unique per user session, and also unique per request. By including a challenge token with each request, it makes it impossible for an attacker to spoof the request.