Group: Secure the Bag
Members: Daniel Alvarez, Sean Connolly, Fernando Franco Crespo, Daniel Luckman

The landscape of security vulnerabilities has changed dramatically in the last several years. As the prominence of web-based applications has continued to increase, so has the amount of vulnerabilities, such as SQL Injection, IDOR/URL manipulation, CSRF/Session attacks, and XSS. The goal of this report is to discuss how to initiate exploits, types of vulnerabilities, how the mitigations were applied, differences between password hashing and encrypting passwords, and what salting and salt + peppering passwords enable—whilst comparatively evaluating them.

Our group was tasked with first creating a functioning webapp, then to exploit the four common software security vulnerabilities within that webapp. The first is SQL injection, which is labeled as the #1 security threat by OWASP. SQL Injection (SQLI) is when untrusted data is used to construct an SQL query. The data is inserted (or "injected") into the SQL query string. The most typical form of SQLI is based around the attacker inputting an edited string on a search form within a web-app that is linked to a SQL database. If an attacker can figure out what the application code for the form is, then they can modify their submission to be able to access elements of the database they shouldn't have access to.

Due to the original code of our website, our website was left vulnerable to SQL injection attacks. The queries sent to the database from SQL did not contain any form of filtering to prevent the syntax from being manipulated.This is proved most, in our websites login page. The only knowledge that the attacker needed was a single or multiple usernames. Upon writing in the username, if the attacker entered " ' or '1=1-- " in the password field, they would be able to log into that account.

Here is the query being sent to the database:

```
sql = "SELECT * FROM user WHERE username = '{}' AND password =
'{}'".format(username, password)
```

The above query is vulnerable and it was fixed by including bind variables. After doing so, it was not easily exploitable.

This is the fix:

```
user = db.execute('SELECT * FROM user WHERE username = ? AND password = ?'
```

The next abuse case is IDOR/URL manipulation which refers to manipulating the URL to get access to confidential data. If we use the example of a website where the userID is visible on the URL, then an attacker could manipulate the URL to change the userID to a different number and then get access to whatever the website GET function is. There are still waves of maintaining unique userIDs for GET requests, whilst preventing them from being exploited by an attacker. For example, access control verifies that only the correct user can get access to more privileged data (such as something like a changePassword function). For our application, the original vulnerable website allowed an unauthenticated user to access the create page by changing the url to the ending /create. This gave an attacker the ability to make a new post without logging in or registering, potentially containing a XSS payload.

Another major webapp security risk is with Cross-site request forgery (CSRF)/session attacks. An attacker can look at how the GET and POST requests for a website work and try to manipulate it to get access to requests that they shouldn't be able to. At its worst, an attacker could input something like changePassword?userID=793, allowing them to change the password of user 793. However, CSRF attacks are very difficult to execute if the developers are aware of it. The three most typical preventative measures include unpredictable session tokens, tying access of most requests to a user's session, and validating that session before every request. Within our webapp we ensured that for any potential malicious requests, it validates that the user session is active. The CSRF attack that we used as an example involved created a forgery of a post request for the creation of a post on our site. This would

allow a hacker to post whatever they want from a the users account if they are still logged in and happened to click this button on the malicious site.

Finally, we also looked at cross site scripting (xss) which is an attack that injects scripts onto a webapp. An example would be if someone on a forum made a post with a malicious script where that script is activated whenever someone goes to that specific forum page. An attacker could use this script to gain access to user data, session id, cookies, and many other sensitive information. The most common way to prevent xss is through escaping, which refers to validating all areas of user input that are valid before they are rendered to the end-user. The tools we used for our project allowed us to escape all HTML tags from posted content, protecting the site and it's users from XSS attacks.

For this report, we were also tasked with addressing the difference between password hashing & password encryption. A common analogy in describing hashing is by looking at the hashed function as a loaf of bread that was created from a recipe. Someone can not take apart the bread to get access to the egg, or the flour that was used in creating it. There are different types of loaves of bread, and the data inside each one relates to the ratios that were used in the bread-making-process. Alternatively, encryption refers to a safety deposit box—in order to get access to the items inside, you need a key. So, what separates bread from a safety deposit box? The safety deposit box is not edible. Hashing is considered a one-way street, whereas encryption is two way, and once something is encrypted it can also become decrypted. Both have their own respective applications, and are used for completely different things.

It is also important to identify what salting, and salting + peppering passwords do. Salting, at its most fundamental level, is adding a random string of characters to the front of each password before it is hashed, and peppering refers to adding a random string to the end of each password before it is hashed. The biggest benefit pepper has is that it's not kept directly within the database. However, it is more often a hardcoded set of data—it can't be changed. If a user just salts their password and a data breach occurs, then any two users who hypothetically have the same password would be at risk, whereas peppering adds another layer of security.