Developing Secure Systems Individual Report

100203952 – Thomas Mcloughlin
CMP-6045B

1 Introduction

This report will present research of the top cyber threats and vulnerabilities of modern systems with the aim of identifying methods to mitigate them.

2 Part 1

2.1 Account Enumeration

Account enumeration is the manipulation of a service's login function to determine the existence of a user. Attackers would determine this through two actions, first by inputting a username and password into the login and if a message is received stating that the password is wrong but not the username, then the attacker now knows that the username exists. The way to mitigate this manipulation is to return a generic message for a login failure not specifying whether the username or password is wrong; however this mitigation can be nullified using the second method where the attacker will try to log in and then compare the time taken to resolve the failed login. If the response was quick then the username doesn't exist, if the response took slightly longer then the service recognised the username and took longer to try and match the password. This manipulation can also be mitigated by applying a delay to the response when the username is wrong so that the attacker could not tell the difference between the two responses. The mitigation methods described above fit into a secure by design approach because they are concerned with the back-end implementation details of the system and are obscured well from any attackers. These techniques have been represented as pseudocode in section A.1 of the appendix.

Threat actors likely to use account enumeration are attackers of any kind, they range from individual, autonomous hackers to well-resourced groups operating in a coordinated manner as part of a criminal enterprise or on behalf of a nation-state (NIST, 2016) A likely attack vector for using attack enumeration would be if the attacker had gained access to a list of passwords for a service and was trying to find users to match to so that they could break into the system.

The interaction between the end users and the service will not be greatly affected by the implementation of these mitigations with respect to the improved security they provide, these mitigations do not affect the process of a successful login. The only problem that can arise with usability will be that if a user forgets their password or isn't sure what username they used for the service, the generic message not specifying which is wrong can be frustrating and make logging in more difficult.

2.2 Session Hijacking

Session hijacking is the utilisation of a lack of security given to website sessions where a user that has logged in creates a session with the web server so that they can make requests to the server without having to send log in details for every request. These sessions have an attached ID so that the web server knows which user it is communicating with; session hijacking refers to the multiple methods used to gain access to a session, usually by accessing the ID.

When attempting to gain access using the ID, one possibility could be that the site uses existent session ID's rather than generating a new ID for every session. This opens the session for attack

from session fixation where the attacker uses a known ID in a phishing email link to have the user login and authenticate themselves then the attacker can hijack the session using the session ID (OWASP, 2020a).

There are multiple methods used to acquire session ID's to use for session fixation. Session sniffing, the use of packet sniffing software to intercept session packets and acquire the session ID attached to it, this can be done manually by the attacker or the attacker may use malware to automate the process. The attacker may also brute force the ID's by going through all possible permutations of the ID.

The threat actors likely to use session hijackers are the same as stated above in 2.1. The risk posed by these attacks are the attackers would be able to gain access to a user authenticated login and perform any actions that the user would be able to within that session such as a money transfer, the attacker would also have access to any personal information that the session allows the user to view which may lead to ID theft, the attacker may encrypt valuable/vital data for ransom which could include intellectual property.

The main methods to mitigate session hijacking attacks include making session ID's long and complex to avoid brute force access, make the site use a new session for each time a user logs in and give each new session a unique ID to stop access if a previously used ID has been compromised, ensure that a session is closed once a user logs out or if the session is not being used and times out and finally, all session data should be encrypted to prevent sniffing and malware attacks from accessing the session ID's. Another method of mitigation extraneous to any technical measures would include the training of end users to spot and avoid phishing emails.

Generating a unique ID with enough complexity to avoid ID guessing is very important. Therefore, an example ID generation algorithm has been included as an acceptable method shown in section A.2. This algorithm was created using recommended practices from OWASP (2020b) such as ensuring the entropy number used is 64 bits to give an acceptable level of complexity.

The addition of any of the mentioned mitigations would have very little effect on the end users as the mitigations proposed are mainly secure by design techniques that are more concerned with backend interaction. Usability will not be sacrificed to a noticeable degree, the only effect on the user would be the requirement to always log back into the system once they leave as the session they previously used would have been closed.

2.3 SQL Injection

SQL Injection is the manipulation of sql queries to interact with a database in a way not intended by design, allowing the attacker to view, modify and delete data from the database. These malicious queries are manipulated using input fields such as the username and password inputs on a login page. If the login input fields take any value inputted and inserts that input into an sql query meant to retrieve user data, an example of which can be seen in section A.3 of the appendix. If the attacker inputs part of a valid query that will always evaluate to true, such as "' OR 1=1;–", then the example query will return all user data from the table. Depending on how much sensitive data is stored in the user table this could be a very dangerous breach, for example if passwords were also stored in the user table.

There are multiple methods of mitigation for sql injection that will be discussed below. Firstly, is the use of prepared statements where instead of inserting input data into sql queries dynamically, the programmer defines all sql queries before- hand with inputs being parameterised. Any input then passed through to that query avoids executing an unintended query and will instead take the entire input as a string, for the example mentioned above using ' OR 1=1;– the query would merely search for a username matching the string "' OR 1=1;–" (OWASP, 2020c).

Secondly the use of stored procedures which act in a similar way to prepared statements to avoid

dynamic sql generation and instead parameterise and validate input. The difference between the two is that stored procedures are sql queries stored in the database that are then retrieved by the application when needed. If implemented safely, avoiding the use of dynamic sql generation, then stored procedures can provide nearly the same protection as prepared statements. However, a possible vulnerability of stored procedures comapared to prepared statements would be in the case of a database requiring certain access levels such as read and write permissions to prevent certain use of the database. In the case of stored procedures, access to execute queries on the database is required meaning that if an attacker were to gain access to the database then they would have full execute privileges (OWASP, 2020c).

Another method would be the implementation of whitelist validation for certain parts of sql queries that cannot use bind variables as placeholders such as the names of tables or columns. In this case if a user inputted parameter is used to alter a queries target table or column then whitelist validation can be used to avoid the risks of sql injection. By mapping input values to a preset whitelist of possible choices for the target, the executed query avoids using the actual input from the user, an example is described in section A.4 of the appendix.

The mitigation methods discussed above would all be necessary inclusions in a secure by design approach as it would take considerable effort to convert a system to use these methods without a near full re-write of the related code. When trying to combat sql injection attacks, mitigation should be considered from step 1 and not be a reactionary change to a system.

The threat actors likely to use sql injection are again the same as those stated in 2.1. The main risk of sql injection is that an attacker could be able to execute any query they want on a database, in the case of a catastrophic breach, the attacker could delete tables containing important data.

End-users are unlikely to experience any adverse effects of the mitigation methods as all methods used to prevent sql injection are backend related, these mitigations would not affect usability of the system a noticeable amount to the common user. The only effect on the system will be the minor performance concerns.

2.4 Cross-site Scripting

Cross-site scripting refers to the various methods used to execute malicious javascript to access or steal information from a victim. There are 3 types of XSS, the first being "Stored". Stored cross-site scripting is the most damaging type of XSS where an attacker uses a website form, commonly on a website that allows public posts such as a social media site, to insert a malicious string into the websites database. This malicious string is now stored in the site, hence the name, and when a victim requests the compromised post, the website includes the malicious string in the response and sends it to the victim. The victim's browser then executes the script inside the response. The script can be used for various functions but is commonly used to steal cookie data so that the attacker can then gain access to the site using the victim's access, this is session hijacking as explained in section 2.2. This form of XSS is particularly dangerous as one insertion of malicious script can affect multiple victims due to the script having been stored in the website database and multiple victims may request that data.

The second type of XSS is reflected cross-site scripting which differs from stored by having the malicious script be sent to the victim first in the form of a link to a site. The attacker tricks the victim into clicking the link which sends the malicious script to the vulnerable site, the site takes the data from the request and adds the malicious script to the webpage where it then executes the script which could be used for the same uses as described above, commonly session hijacking.

The third type of XSS is done through manipulation of a websites document object model (DOM). The attacker constructs a URL containing the malicious script and sends it to the

victim, the victim is tricked by the attacker into requesting the URL from a site that is vulnerable to attack. The website receives the request, but instead of responding to the victim with the malicious script, the DOM of the website is manipulated to insert the malicious script into the website and then the victims browser executes it. This sends the victims cookies and sensitive data to the attacker allowing them to hijack their session or use the sensitive data for other means such as identity theft.

The first important method of mitigation for XSS relating to stored and reflected XSS is encoding website element content also known as escaping. If the attacker has inserted some malicious script into a post on a social media site for example. The site should check the string used for the post and, character by character, encode any possibly malicious characters into a different format to avoid the script being seen and executed as javascript. For example, the "$<script>$" section of the malicious string may be encoded in ASCII, so the '<' and '>' characters will be converted to &#60 and &#62 respectively. To avoid loss of content in the actual post, the html element used to display the content should then be set to use ASCII encoding . This will result in the script being shown as part of the post and the user may read it, but it will not be recognised as script and executed (IBM, 2020). Escaping for different expected characters is used when needed in certain contexts, the example described above is an HTML escape context, but it important to do the same for Javascript and others when needed. An example of how this encoding could be designed is shown in section 5 of the appendix.

Another method of mitigation for XSS is validation of user input. Any data received that originates from outside the system should be untrusted until validated. This validation could be done using a whitelist of known acceptable inputs depending on the context. Along with validating input any untrusted input should also be sanitised to remove unwanted data, depending on the context, including html tags and unsafe characters.

Mitigation for DOM cross-site scripting is more complex than for stored and reflected XSS. When inserting untrusted data into sections of the DOM you should run HTML and Javascript escaping before-hand as described above. However, if the untrusted data is inserted into the Javascript section of the DOM then encoding it will not prevent it from being executed. A fundamental method of avoiding this issue is the use of a safe Javascript assignment property *textContent* and other safe content inserts, an example of which can be seen in section A.6 of the appendix. It is important to avoid the use of innerHTML as an output method and instead use innerText of textContent as explained above. This helps prevent DOM based XSS vulnerabilities (OWASP, 2020d).

The mitigation techniques explained above are all important to a secure by design approach, handling of untrusted data that is received by a site is incredibly important to prevent manipulation of users' access and personal information.

The likely threat actors to use cross-site scripting attacks are once again the same as those stated in 2.1, however cross-site scripting is likely to be used more by those with an advanced understanding of hacking and website vulnerabilities. The common attack vectors for cross-site scripting require the tricking of the user to accessing a site through an attacker provided link, phishing emails being the most common delivery system. The consequences of a successful attack can range from minor to catastrophic, from simple redirection of the browser to cookie theft used for session hijacking, keylogging to steal personal information and credentials, fake login forms also for stealing credentials and using XSS to steal CSRF tokens that will be explained in the next section.

The end users can be affected by some of the mitigation techniques mentioned if not properly implemented. Encoding of scripts being sent to a site can prevent certain actions the site could make that may help usability. However, the security benefits of avoiding XSS attacks are numerous.

2.5 Cross-site Request Forgery

Cross-site request forgery (CSRF) is a method used by attackers to execute requests on a trusted website by using an already open session of a victim. If the victim has a session open with a trusted site then visits a site controlled by the attacker and activates a link, by clicking a button for example, the link can make a request to the trusted site using the open session assigned to the victim, causing some action to be carried out by the site that the victim may not realise. An example would be if a victim is logged into their bank account on the bank's website, then accesses an attacker owned site through a phishing email, the attacker could create a link that makes a request to the bank website to make a money transfer to the attackers account and because there is an open trusted session with the victims account, the transfer would go through.

One mitigation technique for CSRF is the use of tokens for site requests, this attaches a server-side generated token to either the user session or a separate token for each field input, the latter being more secure, so that when the server receives a request it can check for the existence of a token in the request and check that the token matches to validate whether the request is legitimate or not. If the request is found to be illegitimate then the current user session should be closed and logged as a possible CSRF attack. CSRF tokens should be completely unique to avoid the capture and reuse of previous tokens, they should be secret, they should be unpredictable, generated with a suitable prng to prevent sequential brute forcing of token values. The tokens should not be transferred via cookies and should instead use hidden fields and headers, the tokens could also be encrypted to add another layer of security (OWASP, 2020e).

Another mitigation method is the use of captchas which help prevent spamming requests to a server as well as CSRF. By requesting an input that is random and unknown to the attacker you place a blockage between their malicious request and execution. Alongside captchas can be the use of re-authentication for certain actions, for example requiring the user to input their password again when trying to make a bank transfer. This can be taken another step further by requiring 2-factor authentication to avoid the attacker being able to re-authenticate the request if the victim's password has been compromised.

Also commonly used is the double submit cookie technique which involves sending a random value in both a cookie and as a request parameter, the server then verifies if the cookie value and the request value match. When a user visits, the site should generate a cryptographically strong pseudorandom value as a hidden form value and set that as a cookie in the user's browser, separate from the session ID. The site then requires that every request must include this random value as a hidden value. If both values match at server side then the server accepts the request, if not then the request is rejected (OWASP, 2020e).

These mitigation techniques are all good examples of secure by design, ensuring that requests made to the web server are via the user. The end user will only have a slight change to usability when interacting with sites using these techniques, for example the use of tokens can cause the "back" function in a browser to not work as intended due to the previously used token having changed or no longer being valid (OWASP, 2020e).

The most likely attack vector for CSRF is through phishing emails tricking victims into accessing the attacker owned site. The risks involved could include financial loss, stolen data, changing of credentials, ID fraud and much more.

3 Part 2

In this section a comparison and evaluation between three authentication methods will be carried out to determine which would be most suitable to implement in the group web application. The methods chosen are the standard username/password, biometric fingerprint scanner and graphical passwords.

First will be an exaplanation of each method. The standard username/password method is the most ubiquitous authentication method across all types of software. It uses a simple pairing of a unique and semi-public username (sometimes an email address is used) with a secret password. The semi-public nature of the username meaning that it is left uncensored when inputted, is used to refer to the user in communication and, in the case of an email address, used to communicate with the user directly. The username should be treated as secure information and withheld from unauthorised viewing where possible, but it is not considered as important and keeping the user's password safe.
There are many precautions put in place to keep a user's password secret to avoid having an attacker gain access to the user's account. The simplest of which is to censor the password when inputting to avoid shoulder surfing and having someone read your password as you type it in. Some methods of secrecy are placed in the responsibility of the user such as the use of special characters and character length requirements to make passwords harder to guess or brute force, these requirements are usually built into the login system to force users to create stronger passwords.
For back-end related security there's the discussion of storage methods when handling usernames and passwords. Once stored in a properly secured database you would decide what hashing algorithm to use for the password and possibly the username as well. Algorithms such as MD5 are not suitable as they are reversible, this site (NIST, 2020) by the U.S National Institute of Standards and Technology recommends that SHA-256 should be used, at minimum, to provide a secure and unique hash.

Biometric fingerprint scanners work by taking an image/scan of the user's fingerprint and storing it, then when the user tries to login another scan is done and compared to the saved scan to determine whether the user is authorised. Security needs to be considered with how the user's fingerprint is stored so that they cannot be stolen and used for identity theft, an issue with this being the difficulty to securely store an image for this purpose. The fingerprint scanner will not always receive exactly the same image of the fingerprint, software must find the similarities between the inputted fingerprint and the stored image to decide whether the user is authorised. This means that you cannot hash the stored fingerprint for comparison because the inputted fingerprint will be different and produce a completely different hash that cannot be compared. The hardware used for scanning also needs to be able to take high enough fidelity images to recognise the unique differences between each fingerprint to avoid false positives.

Graphical passwords work by having the user choose a varying number of points on an image, this image could be chosen by them, which are then stored. When the user wants to login they must click the same points on the image that they clicked before, allowing for some small margin of error because it will be too difficult to click the exact same pixel. A similar issue to that explained above for storing the fingerprint arises with a graphical password system. You must store the coordinates of the points inputted by the user but you cannot hash them as you will need to compare the new points that the user will click when trying to login.

The reason that fingerprint scanning and graphical passwords were chosen for comparison to the standard username/password method are because they show two directions that technology has taken when trying to improve on the standard method. Both have tried to take what works about username/password and improve where they can, one being successful and widespread (fingerprint scanning) and the other not moving into mainstream use (graphical passwords).

Some of the main reasons for trying to improve on the username/password standard is for im-

proving usability, using a fingerprint scanner is now extremely quick with modern scanners and graphical passwords make it easier for users to remember their login as the human brain is much better at remembering images than it is at remembering words (Grady et al., 1998). The usability of the standard username/password comes from it being so ubiquitous and familiar to the common user, every computer will have at least a keyboard allowing the input of a username and password. Username and password is slow however compared to fingerprint scanning and a graphical password. It also requires remembering a complex password compared to the easier to remember graphical password and not having to recall any information for a fingerprint scanner.
The security trade-offs that go along with these usability points are as follows. For a username/password approach the strong and ubiquitous security comes with the drawback that users are likely to forget their complex password. When given the requirement to have a complex and difficult to remember password, users are going to write down passwords to help them login, therefore negating a main advantage of the security.
While the fingerprint scanner is fast and easy to use, it can be easily obscured if there is dirt on the fingerprint or if there is damage to the fingerprint the user may be unable to login at all (this should be and is avoided using multiple fingerprints), the insecure nature of storing the fingerprint image, mentioned above, is also an important factor. For the benefit of security, the fingerprint for logging in is usually stored on the device rather than on a database server to prevent an attacker from stealing biometric data if the database is compromised; however this comes with the drawback that you must have a different stored fingerprint on each device.
With a graphical password, the issue of complexity when choosing an image is important, with a very complex image, users are likely to forget where they clicked before and users are also likely to pick the same points if they are using the same image. Even if an attacker had never seen the image the user chose before gaining access to it, they would likely be able to brute force guess some of the chosen points.

For the group web application assignment, I believe that using the standard username/ password method will be most suitable. It provides a high level of security when using complex requirements, doesn't require special hardware like a fingerprint scanner, allows storing passwords securely using hashes unlike a fingerprint or graphical password image and will be familiar to all users.

4 Conclusion

In this document I have discussed the common vulnerabilities of web applications in Part 1 with methods of mitigation for each, the threat actors likely to use these vulnerabilities, the attack vectors that they would try to exploit, the risk implications of each vulnerability if they were successfully attacked, how the mitigation methods relate to secure by design techniques and the effect on end-users with usability vs security concerns.
In Part 2 I discussed username/password, biometric fingerprint scanning and graphical passwords as various authorisation techniques, how each of them works, their effectiveness, the trade-offs with usability and security and which I recommend for implementation in the group web application and why.

References

Grady, C. L., McIntosh, A. R., Rajah, M. N., and Craik, F. I. M. (1998). Neural correlates of the episodic encoding of pictures and words. *Proceedings of the National Academy of Sciences*, 95(5):2703–2708.

IBM (2020). https://www.ibm.com/garage/method/practices/code/protect-from-cross-site-scripting/.

NIST (2016). Guide to cyber threat information sharing. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-150.pdf.

NIST (2020). https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions.

OWASP (2020a). https://owasp.org/www-community/attacks/Session_fixation.

OWASP (2020b). https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html.

OWASP (2020c). https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

OWASP (2020d). https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html.

OWASP (2020e). https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

A Appendix A

A.1 Account Enumeration Mitigation

Algorithm 1 login(*username*, *password*) **return** *response*

Require: *username*, the username for login
Require: *password*, the password for login
Require: *users*, the set of users and passwords that the system will compare the login against
Ensure: *response*, either a successful login or a response message

```
1:  Failure ← "The username/password is incorrect"
2:  for all user in users do
3:      if username = user.username then
4:          if password = user.password then
5:              response ← Success                    ▷ Username and password correct
6:          else
7:              response ← Failure                    ▷ Username correct, password incorrect
8:          end if
9:      else
10:         delay                                     ▷ Wait however long the check for the password would take
11:         response ← Failure                        ▷ Username and password incorrect
12:     end if
13: end for
14: return response
```

Figure 1: Example of a login with account enumeration mitigation included

A.2 Session ID Generation

Algorithm 2 generateUniqueSessionId() **return** *sessionId*

Require: *IdList*, the set of existing session Id's
Require: *prng*, pseudo random number generator
Ensure: *sessionId*, a unique session Id

```
1: entropy ← prng(64)          ▷ Pseudo random number generation of length 64 bits
2: Id ← generateId()                                          ▷ Generate complex Id
3: sessionId ← concatenate(Id, entropy)
4: if sessionId ∈ IdList then
5:     generateUniqueSessionId()          ▷ Session Id isn't unique, generate a new Id
6: end if
7: return sessionId
```

Figure 2: Example of a random session Id generator using recommended practices from OWASP (2020b)

A.3 Basic User Retrieval SQL Query

```
SELECT * FROM Users WHERE username='' AND password='';
```

Figure 3: Example of a query that could be used to retrieve user data

A.4 Whitelist Validation Check

Algorithm 3 getTableName(*userInputValue*) **return** *outputTableName*

Require: *userInputValue*, the value for the table name inputted by the user
Require: *tableNameList*, list of valid table names
Ensure: *outputTableName*, the whitelisted chosen table name

```
1: for all name in tableNameList do
2:     if userInputValue = name then
3:         outputTableName = name
4:     end if
5: end for
6: return outputTableName          ▷ if outputTableName returns empty then we
                                     know that the user input was invalid
```

Figure 4: Example of table name validation, an adapted example from OWASP (2020c)

A.5 External Script Character Escaping

Algorithm 4 stringCharEscape(*inputString*) **return** *encodedString*

Require: *inputString*, the inputted string (potentially malicious)
Require: *charEncodingList*, list of characters encoded values
Ensure: *encodedString*, the final encoded string

```
for all char in inputString do
    if char.encode ∈ charEncodingList then
        encodedString ← concatenate(encodedString, char.encode)
    else
        encodedString ← concatenate(encodedString, char)
    end if
end for
return encodedString
```

Figure 5: Example of character escaping to encode a possibly malicious string

A.6 Use of textContent for Inserting Untrusted Data

```
<script>
element.textContent = untrustedData;  //does not execute code
</script>
```

Figure 6: Example of using textContent to insert untrusted data safely into the DOM (OWASP, 2020d)