CMP-6045B - Individual Research Report

100225776 – Martin Siddons

1 Introduction

In the first section of this report, I will detail how a selection of vulnerabilities work and how they are mitigated. In the second section I will detail some authorisation techniques and explain which one I would choose for implementation on a modern web application.

2 Analysis of Common Vulnerabilities

2.1 Account Enumeration

Description: Account enumeration is the process of identifying if user exists in an application. There are two types of account enumeration, message based, and timing based. It is a popular and common vulnerability, with thirty two CVEs logged by Mitre in 2020 (Mitre, 2020).

Threat Actors: This is an unsophisticated vulnerability usable by most threat actors, even those with little programming skill such as script kiddies using brute force.

Attack Vectors: Message-based account enumeration looks for hints in the response message from trying an incorrect username or password. For example, if an attacker targets a page of an existing user and receives the reply "Incorrect login details", they will not know if the user exists. The message "Password is incorrect" could imply the account exists. Timing-based account enumeration looks at how long a reply takes to come back from the server, where longer times indicate an account was found due to the time it takes for the server to hash and compare the given password. HTTP response codes may show if an account exists (Hacksplaining, n.d.a), therefore developers must ensure the headers for existing and non-existing accounts match.

Risk implications: Hacksplaining (n.d.a) identifies account enumeration as a common and easily exploitable vulnerability which, on its own is not considered too impactful. Impact can be increased significantly if this vulnerability is combined with others such as brute force, as it can lead to compromise and a loss of data integrity on an account, which is dangerous for Admin accounts where there could be the potential for data breach or destruction, or loss of service.

Secure-by-design Techniques: Developers must ensure that errors for incorrect password and non-existent usernames are the same so as not to hint if a username exists. They must also ensure that the time taken for the system to reply to a login is consistent for existing and non-existing users to guarantee an attacker does not receive a hint through time taken. OWASP (2020b) recommends testing to ensure there are no hints given in URLs, page titles or HTML packets too. Hacksplaining (n.d.a) mentions having the service send a password reset email if a user tries to sign up with an existing username, rather than say the account exists. For secure services, it would be advisable to not allow user pages to be discoverable via the URL and instead use a pseudo-random designation, necessitating a brute force attack.

End-user Usability vs Security: As Taylor (2020) explains, obscuring if the username exists can be frustrating to users, but our shared opinion is that the trade-off for security is worth the inconvenience. For some sites where the username is visible to others, such as social media, the service should require different login and display names. This is unlikely to impact usability enough to be a concern, especially if the login form asks for an email rather than username. For admin accounts, forcing them to log in through a more difficult route would discourage using the admin account unless absolutely needed.

Algorithm For Mitigation: See Appendix A.1

2.2 Session Hijacking

Description: OWASP (2019) describes session hijacking as "the exploitation of the web session control mechanism, which is normally managed for a session token". A session token or ID is stored as a cookie on the user's device, as token in the URL, or in the HTTP request header or body. An attacker can take this ID and fool a server into thinking they have been authenticated.

Threat Actors: Since the attacker needs to know which type of token is being used and the victim, this attack is most suited to cybercriminals looking for financial gain.

Attack Vectors: Attackers can use one of the following methods to hijack a session from a potential victim. **Session Fixation**: Used when session ID is part of the URL. Attacker sends victim a link to the site to log in but appends a random session ID to the end of the link. The user logs in to their account as normal and the attacker can then visit the same link to have their session authenticated as the victim. **Session Sniffing**: When the attacker has access to the same network as the victim, they can use a packet sniffer to steal session cookies from them to impersonate the victim's session. **Malware**: An automated version of packet sniffing can intercept cookies being received and relay them to the attacker. **Brute Force**: When the site being attacked has predictable session IDs, the attacker can try guessing the ID and hope they get a random valid ID. **Cross-Site Scripting**: Attacker either tricks the user into clicking a link with malicious code embedded within, or the attacker embeds malicious code into a page that the victim visits. This then returns the currently active cookies to the attacker.

Risk implications: OWASP (2020a) lists Broken Authentication (including Session Hijacking) as currently the number two biggest application security risk. Hacksplaining (n.d.g) describes session hijacking is a relatively rare occurrence that is moderately exploitable and reasonably harmful to the target. Through this attack it is possible that admin accounts could be compromised, leading to the loss of data confidentiality, integrity and availability for the entire service.

Secure-by-design Techniques: Authentication should always use SSL/TLS. Since 2016, the Electronic Frontier Foundation (EFF) have recommended HTTPS for use across an entire site specifically to prevent content hijacking (EFF, 2016). Schneider (n.d.) mentions that session renewal is key to defeating these attacks. This requires a new ID to be fetched while the user is logged in, which invalidates the current ID and prevents an attacker from further access. Session renewal also defeats sessions stolen through brute force and malware, as the session ids found would only be useful on a couple pages before being rejected. Using longer, more complex URL session IDs that are routinely renewed would prevent attacks that target session cookies. Users can also be logged out of their session automatically once they are finished with them. The site could also deny access to users of older, more vulnerable browsers.

End-user Usability vs Security: Using SSL/TLS for authentication would be invisible to users, as would utilising HTTP headers for session IDs. Logging users out would be would be a visible action only suitable for financial services, as users frequently visiting lower profile sites may be annoyed by being logged out. For many sites, simply renewing the session ID would be enough. Denying access from older or outdated browsers would be recommended as the benefit outweighs the small number of annoyed visitors in my opinion.

Algorithm For Mitigation: See Appendix A.2

2.3 SQL Injection

Description: SQLi involves an attacker tricking a service into executing SQL code through a form input, allowing modifying existing data, creating new data, and/or deleting anything held on the database, including the database itself.

Threat Actors: Cyber Terrorists wish to destroy critical systems, Cybercriminals and State-Sponsored Actors can blackmail, commit further attacks or sell information to a third party and

Hacktivists can expose internal data or disrupt business. An Insider would value this approach if they were looking to cause damage to their employer's systems.

Attack Vectors: Executing an injection on a target system can be done via: **POST request**: A statement such as *'OR 1=1–* is entered into a form on the page and submitted. If the page returns data or an SQL error, that input is open to SQLi. **GET request**: If the page encodes GET requests in the URL, the attacker can try changing a query from *products.php?id=123* to *products.php?id=%27*. If the server gives an error, it is open to SQLi. **Via Cookies**: Aboukir (2012a) explains that changing a cookie field like *lang=en* to *lang=en'* can execute a SQLi when the site reads the cookie. **Via Server Variables**: An attacker can modify a request header to *User-Agent: 'OR 1=1–* to test for SQLi (Aboukir, 2012b).

There are then multiple techniques that can be used by to expose the database: **Piggybacked queries**: The attacker adds their request on the end of the expected query, such as *id=123* being modified to *?id=123+OR+id%3D+124* allowing for additional execution (Borland, 2002). **Tautologies**: The attacker appends something that is always true, such as *'OR 1=1–* to a request to receive all items. **Abusing alternate encodings**: Using Hexadecimal, ASCII or Unicode to get around a filter that might be in place. For example, using *0x27* instead of *'*, or encoding an entire query into hex (Alwan, 2017). **Illegal Queries**: Alwan (2017) mentions logically incorrect queries are those which are known to produce error messages that include the database metadata. For example, *AND CONVERT (char, no)*. **UNION**: Allows the attacker to append their own SELECT statement on to the end of a valid statement if both return the same number of columns. **Inference**: Clarke (2012, p. 240) explains *IF* or *CASE* statements such as *IF SYSTEM_ USER='sa' SELECT 1/0 ELSE SELECT 5* can infer if specific data exists, this example tells if the database is run by the system administrator if an SQL error is returned. **DBMS specific**: Many exploits will work differently depending on the Database Management System (DBMS) being used. In Clarke (2012, p. 204-210), the author details how to expose password hashes in Microsoft SQL Server, MySQL, Postgres and Oracle DBMSs.

Risk implications: The risk posed by an SQLi attack is high. OWASP (2020b) lists Injection flaws such as SQLi as the number one web application security risk. Hacksplaining (n.d.d) comments that SQLi is the one attack everyone should be looking out for. They consider it a reasonably prevalent vulnerability, easily exploitable with potentially devastating consequences. SQLi can expose data to the attacker, enabling them to break data confidentiality, integrity, and availability. This exploit can also lead to privilege escalation allowing the attacker full control over the database and any services running on the server (Prodromou, 2019).

Secure-by-design Techniques: SQLi can be defeated using a combination of: **Input validation** Regular Expressions are good for ensuring data coming in matches a certain pattern. Fixed values can be provided via a dropdown list to ensure only certain values can be entered. **Parametrised Queries**: OWASP (n.d.d) recommends the statement is prepared first then parameters are passed into the statement as text only, which prevents the database executing data as code. **Stored procedures** are not safe by default, as demonstrated by Goyal (2008). The key is to parametrise the input as shown in the above example. **Use Least Privilege**: The Principle of Least Privilege says the database account should have the least amount of access to the system needed to work. The account being used should not have access to admin commands or the ability to query the database metadata, and multiple database accounts can be used to accomplish different tasks. **Web Application Firewall (WAF)**: PT (2019) recommends using a WAF to prevent most attacks we are looking at in this report, including SQLi. A WAF monitors traffic to and from a web server and looks for threat patterns.

End-user Usability vs Security: Using drop-down lists or spinners would need to be done in a way to ensure security without making it hard for the user to input what they need. Developers should be mindful of the length of space they give users to enter data such as names, as some are very short (Ng), long (Balasubramanian) or include non-standard characters (Kozłowski, 王)

that must work with validation so as not to restrict what users can input. In addition, validation must correctly identify input to avoid errors such as with True (2021), where Apple's iCloud service identified a user's surname as a Boolean value, causing it to crash.

Algorithm For Mitigation: See Appendix A.3

2.4 Cross-site Scripting

Description: In a Stored XSS attack, an attacker inserts JavaScript into a database which the browser executes on load. In a Reflected XSS attack, the victim receives a link containing JavaScript that is executed when loaded. DOM-based XSS is similar but only runs in the browser.

Threat Actors: Reflected XSS is likely to be used by Cybercriminals and State-Sponsored Actors to target individuals for money or sensitive information. Stored XSS (and Reflected XSS from email bursts) attacks target users or sites randomly and would come from Cybercriminals.

Attack Vectors: XSS relies on getting a browser to execute JavaScript, which leaves hundreds of known attack vectors to exploit (Kurobeats, 2017). **Stored XSS** can be tested by entering *<script>alert(1);</script>* in a form saved to the database on the target site. The attack is successful if a popup box appears when loading this data. In **Reflected XSS** the attacker provides a link such as *www.legitimatesite.com/products.php?id= <script>alert(1);</script>* to the victim which brings up a JavaScript alert if successful. This script does not get stored by the site but is returned and executed by the victim's browser. **DOM-Based XSS** relies on a site to be running JavaScript that displays something provided by the user which is executed in the browser itself. A good example of this is provided by Graceful (2016) where a site such as *www.legitimatesite.com/#input* could use some JavaScript to display "input". If the link is changed to contain a script and it runs on the page, the site is vulnerable.

Risk implications: OWASP (2020a) currently lists XSS as the seventh security risk. Hacksplaining (n.d.e) rate both Stored and Reflected XSS (Hacksplaining, n.d.f) as being quite common, easily exploitable with harmful impact. XSS attacks do not generally target the site itself, though if the victim is an admin the site itself could be compromised. Developers should be concerned of reputational impact as victims see attacks coming from their domain. This attack can compromise the integrity of the victim's data and easily lead to further vulnerabilities.

Secure-by-design Techniques: All attack vectors can be defeated by following some simple rules laid out by OWASP (n.d.b), the primary one being to not allow untrusted data to be displayed on your page to limit the opportunities attackers have to exploit.

Five characters must be encoded to prevent a script from executing inside a HTML tag (such as *<div>here</div>*): & (encoded to &amp;), < (encoded to &lt;), > (&gt;), " (&quot;) and ' (&#x27;). When passing data into a HTML attribute, JavaScript that runs on the page, inside dynamic JSON or CSS, as well as with data coming in from a GET request, those above must be encoded as well as % * + , - / ; = ^and |. OWASP (n.d.b) recommends similar techniques when using JavaScript attributes such as element.innerHTML() and methods such as document.write() to prevent DOM-Based XSS.

End-user Usability vs Security: When implemented correctly, character escaping or encoding should not be visible to the end user. Issues may arise for users using older alternative encodings such as Shift-JIS (Japan). As WHATWG (2021) notes, encoding for web pages should be in UTF-8 for maximum compatibility rather than ASCII.

Algorithm For Mitigation: See Appendix A.4

2.5 Cross-site Request Forgery

Description: CSRF happens when a victim logs into a target site and clicks a link to a malicious site the attacker controls, whereby the malicious site uses the victim's active session to send a request in order to steal data from the victim, as well as create more links to the malicious site.

Threat Actors: State-Sponsored Actors and Cybercriminals use CSRF to target individuals for financial gain or to take sensitive information, or use a spam campaign to trick random victims into clicking a malicious link. Hacktivists would use CSRF when targeting the admin of a target site. If used to destroy data, Cyber Terrorists could also use this vulnerability.

Attack Vectors: As KirstenS (n.d.) describes, this attack requires building a URL or script that exploits the site to do what the attacker wants it to do, then trick the target into executing the exploit. This attack can be done via: **GET request**: A site submits requests as *transfer.do?acct=[ACCOUNT_NAME]&amount=[AMOUNT]*. If the attacker uses their account name and has the victim with an active session click the link, the site will submit a request from the victim for the amount given. **POST request**: A form with hidden inputs matching the above can be submitted using JavaScript when the victim with an active session visits the page.

Risk implications: KirstenS (n.d.) considers CSRF as a common vulnerability which is easy to exploit with harmful impact. Though CSRF does not target the site directly, it could still suffer in lost revenue, damages and impacted reputation from affected users. Users can lose the confidentiality and integrity of their data and attacked administrators could risk site integrity.

Secure-by-design Techniques: OWASP (n.d.a) gives the primary prevention technique as using CSRF tokens, either with state or stateless, held in the DOM directly or in a JavaScript variable on the DOM. A CSRF token is a unique, unpredictable string generated server-side per session or per request. **Synchroniser Token Pattern**: When the user issues a request to the server, the server compares the token given in the request with the token in the current user session. If the tokens do not match, the request is dropped before being processed. **Encryption-based Token Pattern**: A token comprised of the user's session ID and a timestamp is given to the user in a hidden field on a form. When the form is submitted, the server checks the token has not expired and that the session ID matches.

Changes can be made to cookies to mitigate CSRF too. **SameSite Attribute**: Set to either Lax or Strict within a cookie to control if the session is sent to the site when linked from an external source. Strict would never send the cookie on, so the user would have to re authenticate whenever they click onto the site. Lax allows most external clicks through as long as the user is not accessing a restricted area, so a products page would work but linking to a checkout would be denied. All chromium-based browsers have this value set to Lax by default as of Chrome 80 (August 2020) (Chromium, 2021). **Double Submit Cookie**: Stateless technique where the same random value is set in a cookie and as a request parameter. Requests to the server are bundled with the cookie and the server verifies that both submitted values match. **Re-authentication**: Can be forced before sensitive events such as entering a checkout, changing a password etc. **Custom Request Headers**: Can be used to secure an API endpoint (OWASP, n.d.a). OWASP (n.d.a) advises not to use GET requests for state changing operations since these techniques only work for POST.

End-user Usability vs Security: Per-session based CSRF tokens are invisible to the end-user, but if generated per request the user will have issues using the back button in the browser as there would be a mismatch between tokens. For security, this ensures forms are not tampered with after initial submission though, which could cause some processes like checkouts to otherwise fail. The Strict SameSite attribute forcing re-authentication would be frustrating when viewing posts on a social media site. Where re-authentication is necessary, such as a checkout, a message should be displayed explaining that they need to re-authenticate for security reasons for their protection. This should help to alleviate annoyance felt by most reasonable users.

Algorithm For Mitigation: See Appendix A.5

2.6 Web Application Specific Additional Mitigation: Unvalidated Redirects

Description: For an additional mitigation, I consulted both OWASP (2020a) and Hacksplaining (n.d.b), choosing to cover **unvalidated or malicious redirects** as it is different from preceding vulnerabilities and rarer, so many sites that may not have considered it. This attack can occur on sites that redirect users when clicking a link before they can access a page. An attacker can modify the redirected page to point to a site they control to attack the victim.

Threat Actors: This would be a targeted attack that could be exploited by Cybercriminals and State-sponsored Actors for financial gain or to steal site credentials. Script Kiddies may also prefer this type of attack as they only need a URL that points to a site to harvest data.

Attack Vectors: If a site uses a redirect parameter in the URL, the attacker could verify the redirect is not being validated by changing where it leads. The target site could then be set up identical to the original site to trick people into giving up their login details.

Risk implications: Hacksplaining (n.d.c) describes this vulnerability as being common and easily exploitable but of limited impact, with which I would agree. As it is not too dangerous to the target site, there are likely many developers and admin that are unaware of it. However, there is a potential for reputational damage to companies, as victims think they are clicking on a legitimate page and end being attacked. For users, an attack using unvalidated redirects could expose their account to loss of confidentiality and integrity. If an admin were to be caught by this attack, there could potentially be issues with loss of availability to the site too.

Secure-by-design Techniques: OWASP (n.d.c) lays out the prevention to this attack. **Avoid redirecting**: Not always suitable, but the most effective way to stop this attack happening. **Do not allow user input**: The site could redirect users using a check on the server rather than an attribute within the URL itself, also preventing this attack. **Provide a token**: In the URL which the server can map to an address or disregard when the request is received. **Validate the redirect**: Ensure the redirect is within the same domain or subdomain as the site itself, though this could be bypassed if the attacker registers a URL that matches the validation.

End-user Usability vs Security: Outside of the redirecting itself, which would likely bother some people, this should not be noticeable. I would personally opt to avoid redirecting users in the first place as that is guaranteed to work and not annoy users at the same time. When redirects are necessary however, such as when the site being visited is only being available with an account, the redirect should be held in secret on the server and never visible to the user.

Algorithm For Mitigation: See Appendix A.6

2.7 Extra Considerations

2.7.1 Risk Matrix

I have considered the risks that a vulnerable, high reward web application would be targeted with the above vulnerabilities and plotted them to a Risk Matrix, in Appendix B figure 1. We can see from this that the maintainers of the site should focus primarily on ensuring the site is not vulnerable to SQL Injection, with Session Hijacking, XSS and Account Enumeration following.

2.7.2 Threat Actors

Cyber Terrorists look to cause damage against those they perceive as their enemies, with political or ideological motivation. They usually aim for maximum destruction, targeting critical infrastructure for businesses. **State-Sponsored Actors** are funded by governments to further the interests of their nation by stealing or destroying information or machinery. They target private sector organisations or governments in enemy countries. **Cybercriminals** are individuals or organised crime syndicates looking to earn money illegitimately online by stealing it from others or selling information on the black market. They target cash or data-rich businesses. **Hacktivists** target groups and businesses opposed to their goals to ruin their reputation, or high-profile sites for publicity or to further their political, social, or ideological goals. **Script Kiddies** have little to no coding experience and rely on tools developed by others to exploit vulnerabilities to cause as much damage as possible, earn money or for entertainment. **Intentional Insiders** are current or former employees of a company who want to cause damage to said company's systems or reputation. Their insider knowledge of the company make them hard to defend against. **Unintentional Insiders** are current employees or users that unintentionally cause damage to a company's systems through user error or poor system design.

3 Authorisation and Authentication Techniques

Authentication can be classed into four groups: Knowledge factors, or what you **know** (Password, PIN or Security Question), ownership factors, or what you **have** (Access Token, Dongle, Key, or Phone with Software Authentication), inference factors, or what you **are** (Fingerprint, Retinal Pattern, Location) and behavioural factors, or something you **do** (Keystroke Dynamics, GUI interaction, Signature, or Speech Pattern), (Shepherd, 2019). For this section of the report, I have chosen two alternatives to Username & Password - **Swipe Pattern** and **OAuth**.

3.1 Description of Chosen Authentication Methods

Username-Password Combination: This is the ubiquitous default for almost any web application in use today. The user provides their identification as either their email address or a moniker they choose. The authentication comes in the form of a string of characters, generally a mix of numbers, symbols, lower-case and upper-case letters (though limitations can vary between implementations). The username and password are supplied by the user at signup, and must adhere to certain rules concerning length and characters used. When the user wishes to log in to the system, they supply these same details. The server then compares the given password to the one stored in the database with the username and if they match, the user is authorised.

Storing passwords in plain text is highly not recommended due to how easy it would be for an attacker to discover them. Hashing must be implemented on passwords, which involves using an algorithm to transform the password into a string of letters and numbers which cannot be reversed. Older hashing algorithms such as SHA and MD5 are not recommended for storing passwords too due to how easily they can be broken by modern computers. For example, Pound (2016) demonstrated that it is possible to brute force 3.8×10^{10} (38 thousand million) MD5 hashes per second. In addition to hashing, salting should also be used to ensure the hash remains secure. This is a random series of bytes usually stored with the password which is appended to the password before it is hashed to increase the complexity and security of the hash. Peppering can also be used to increase security further, where a secret salt held in a separate area of the server (such as in a system variable) and added to the password and salt before hashing.

The password, should be able to be changed and reset at will, and admin often put in rules to ensure that passwords are changed on a regular basis. For security, it is recommended Hacksplaining (n.d.h) that the old password is entered along with a new one if the user wants to change it to prevent an attacker locally changing it and locking the victim out.

Swipe Pattern: A swipe pattern is an authentication method where a user draws a single line across a grid of points which, when input in the correct order, allows access. Generally, swipe patterns use a grid of nine points arranged in a square and the user must connect at least four points in one movement. The points are stored on the device or application as an ordered list of numbers representing the points, generally encrypted in some way (Tahiri, 2013). Upon attempting to log in, this list is checked against a list generated by the user and if they match, authorisation is granted. Swipe patterns do not allow the user to re-use points once they have been swiped over once, likely to allow them to be more easily remembered.

OAuth: OAuth is an open access standard for authentication which allows a developer to verify a user's identity via another existing login that user has on another service, such as Facebook or Google. The developer can embed the OAuth API from the service they would like to have the user authenticate with on their app and it is up to the third-party site carry out the authentication. When a user wishes to log in to the given web app, the server sends the authenticating service, such as Google, the type of access to be granted (Google, 2020). The authenticating service then verifies the user details entered and generates a session and user consent, and passes back an authorisation code to the original web app, which that app then exchanges back with the third-party site to receive an access token and a refresh token. The host web app can store the refresh token and use the access token to access an account API. If this access expires, the web app can use the refresh token to obtain a new one. This way, the app can authenticate and maintain the session of a user without storing the user's login details on the server.

3.2 Reasons for Choosing the Alternative Methods

Swipe Pattern: As I was already looking at a knowledge factor (username & password), I wished to look at one inference or behavioural factor and one ownership factor. For inference factors, swipe pattern does not require any special equipment beyond what is available to a web app user in the first place. Fingerprint and Iris scanners require specialised hardware and software to decode, which will not be available to all potential users, and retinal scanners would not be feasible for almost any application beyond physical locations that need the highest security.

Compared to other behavioural factors, keystroke dynamics and GUI interaction would differ depending on the device being used, which would be hard to work around for a web app. Signature input would be difficult for most devices and signatures would be too varied between tries to be a reliable authentication technique when compared to other choices. Speech pattern recognition would not be suitable for a web app due to its complexity and the amount of storage needed.

The closest technique to swipe pattern here would be graphical password. I chose the former as it is easier to understand (many smartphone users will be aware of it already) and graphical passwords do not offer much additional security (as they are still susceptible to shoulder-surfing attacks) compared to their complexity. Additionally, accuracy would be a factor if the app were to be used on a smartphone, meaning there would need to be far more leeway in where the hot spots were on the image, negating any additional security we would gain from using it.

OAuth: Initially I was going to look at One-Time Password (OTP) via software authentication on a smartphone as an ownership factor technique, however this is not a service that can exist as sole authentication. This is due to most OTP apps requiring a sync with an existing account before being used, where it acts as a two-factor authentication (2FA) procedure. OTP also requires the user to not only have a smartphone but also know how to register and use a separate authentication app. This difficulty was highlighted by Milka (2018) where the author stated that less than ten percent of Gmail users have enabled 2FA on their account, with ten percent of users having issues understanding how to correctly enter a validation code into the prompt.

A dongle would not be suitable here as it would incur a large cost and setup time per user to the company requiring the service and is suited more for verifying users of expensive hardware or software. A token device would suffer from the same issue as OTP as they are both similar approaches, with the token device requiring more cost due to it being a physical object. I chose not to look at pin codes or shared secrets as these are knowledge factors which are covered more suitably by username & password. I decided on OAuth as it is becoming a popular modern authentication technique with what I felt were many upsides. We are beginning to see sites that have OAuth as their only method of authentication too, which proved to me that this would be a suitable method to cover in this section, as I ideally wanted a method that did not rely on another form of authentication to work, such as a username.

3.3 Usability vs Security

Username-Password: Passwords have been standard for a long time in computing, which means their usability has been tested by many different groups to find the best trade-off between usability and security which will differ depending on the risk tolerance of the developers and company. For passwords, we can limit the length and usable characters.

Regarding length, a site require longer passwords, which while safer may cause some people to simply add repeating numbers to pad the end of the password they use on other sites, which does little to prevent an attack using more modern brute-force techniques (Hitaj, 2019). Some sites will limit the maximum number of characters to something too small to hold passwords that are very secure, such as four random words joined together. This can frustrate users who are used to using longer passwords and force them to reduce its complexity to remain compliant.

Regarding usable characters, some web apps do not have any minimum requirement, allowing users to use a very unsecure lower-case only password. Others may require a minimum of an upper case letter, or that and a number, or both and a symbol. The more complex the requirement, the harder it is for users to produce a password and the more frustrated a user will get. Adding more characters to a password can be shown to have a greater effect on password entropy compared to additional options per character (Poza, 2021). Indeed, the NIST no longer recommend developers impose composition rules on passwords for this reason (NIST, 2017).

Some groups feel it is necessary to force users to change passwords regularly, however this can result in password overload for users. Because of this, the NCSC no longer recommend forcing a password update if the account has not been breached (NCSC, 2018). The web app could use a service to test known bad passwords against a list of compromised passwords or hashes and reject those that match. This may be frustrating to users that can't use their usual password, but if we use a message explaining why the password was rejected (and urging the user to change their other passwords), they would be better informed and likely less annoyed.

For usernames, it would be a good idea to have a username separate from the name displayed on the app since this removes a possible attack vector. Generally, apps will use the email address for this purpose, since the email is required for confirming a password reset. It might be more secure to require a third name for identification, but this would likely be vary confusing to users who already need to remember if the site is requiring their screen name or email on the login form, so I feel this trade off is not worth making.

Swipe Pattern: Swipe Patterns are a relatively unsecure method of authentication compared to other methods but are preferred by many due to their ease of use, especially in situations where the authentication must happen regularly, such as a smartphone lockout. However, we could choose to implement rules on the choice of pattern which would pay off in terms of security later. We could enforce a high minimum number of points. Since most swipe patterns are on a three-by-three grid, the user would only have nine points to choose from. Enforcing a minimum of six of these nine will make it harder for some users to produce the pattern they want but the pay-off in security would be worth it.

Disallowing certain arrangements such as letters is a possibility too. This would increase security even further as 44% of users start on number one, and in one survey (Løge, 2015), 42% of patterns created were in the one hundred most used patterns in a possible set of over 389 thousand.

Persuading the user to not use these could allow for more variation between users, increasing security. The downside would be that some users will quickly get frustrated if all the patterns they try were on the banned list, which is quite likely considering how popular these patterns tend to be. Therefore I feel this trade off not beneficial for the end-user.

We could allow users to cross over points without activating them, only detecting a direction change as an activated point. This may annoy some people if they want to use a point they can't activate, but I feel the trade-off in extra security outweighs this as it raises the total number of patterns to almost one million. We can also allow the option of removing the grid from the screen when users authenticate to increase security and reduce the risk of shoulder surfing attacks.

OAuth: The largest factor when it comes to the usability of OAuth is how new it is, in my opinion. If a site relies only on OAuth, it might drive away more casual users that have not seen it before. There might be users who know a little about security but not enough to understand what OAuth is, who would become very wary that the app was "asking for my Facebook", causing them to leave the app at the risk of losing access to their third-party account.

There will also be users that will either not have access to the OAuth accounts offered, or for privacy reasons would not want the given third-party sites logging their details. Indeed, Eran Hammer, the original lead author for OAuth 2.0 resigned from the role before it was completed, citing that the enterprises concerned with designing the standard were looking for "a new source of revenue through customisations" (Hammer, 2012). Many users will rightly question where the revenue is coming from when the user and site are not directly paying for the service.

Despite OAuth requires the user to go through additional screens to log in, this ends up generally being quicker than other methods since the login is usually just a list of accounts to select. This does trade security for usability, however, as a local attacker can easily log into a web app in a couple seconds if they have access to an open device.

3.4 Recommended Method for Implementation

3.4.1 Username-Password

Pros: They are common enough that almost every user should be expected to understand how they work. They allow users to re-use the same identification across multiple applications making it easy to remember and easy to recover the password using that identification. If passwords are sufficiently hashed, salted, and peppered (which is easy to implement), they can not be broken in a reasonable amount of time. Usernames and passwords are easily created and stored in a database. Third-party services exist which can manage password storage and generation of separate, secure passwords for every application and service.

Cons: If the password is a common word and the web app does not limit logins, that password could be easily broken with a dictionary attack. Users are likely to reuse both usernames and passwords between applications, so one breach puts all accounts at risk. NCSC (2018) mentions that users can suffer from password overload where they are expected to remember a different password for every service, which encourages users to write down all their passwords where an attacker could easily get hold of them. If the username and password systems are not implemented correctly, authentication details could be leaked to attackers.

3.4.2 Swipe Pattern

Pros: Swipe Patterns are easy to learn and remember. They are quick and easy to enter on a touch device. If it is limited to securing a device like a smartphone and are encrypted, they are more complex to brute force compared to passwords as the user would need physical access and be able repeatedly try without being blocked.

Cons: A swipe pattern works well for device access, but for web applications it would need to be paired with a username, negating the advantage of being a quick and easy source of authentication. They are susceptible to shoulder-surfing attacks (Aviv et al., 2017), where 64.2% of attacks were successful with one observation. Another attack vector is the smudge attack, Aviv et al. (2010) found that 68% of all patterns were fully identifiable just from the smudge on the device, and 92% of patterns were at least partially identifiable. If brute force is used on a web app swipe pattern, there are few possible combinations for the pattern (389,112 patterns between four and nine points in length (Adrián S, 2016)), compared to $\sum_{i=4}^{9}(26+26+10+32)^i = 5.79\times10^{17}$ for passwords in the same range. Using a swipe pattern on a PC or laptop would be much more difficult than with a touchscreen due to accuracy issues, especially for older persons.

3.4.3 OAuth

Pros: OAuth simplifies authorisation by offloading it to a third-party. Security for the web app is improved by not requiring the app to track users. OAuth transfers data over SSL and has been thoroughly tested to ensure there are no gaps in the system that could leak sensitive details. OAuth provides a more secure experience for the user as they do not need to worry if the app will leak their details one hundred percent secure. OAuth allows the user to see what clearance the app wants, via the user consent screen. Users can also revoke access easily at any time.

Cons: The user can't see how their data is being used by the third-party service. If the third-party has an outage, it could affect OAuth login services. Similarly, if the third-party permanently goes out of service, the login system for all implementing apps could be broken if that is the only service they implement. If an attacker can breach the third-party site, they could target and gain access to the same account on every service using OAuth.

3.4.4 Deciding on a Method

The Swipe pattern method was removed from recommendation quickly due to its inherent security flaws. Between Password and OAuth, OAuth has more positives and fewer negatives going for it but has privacy concerns due to not knowing what data is being passed to the third-party. Password can be the most secure and the best authentication technique for privacy, but only if it is implemented securely enough. Due to its ease of implementation, therefore, I would recommend OAuth for my group's web application authorisation.

3.5 Suitability of the Recommended Method

OAuth can be more complex to implement compared to a standard Username and Password, as the developer needs to interact with an API rather than simply querying a database, but using OAuth ensures it is almost impossible to expose details of the process outside the current session. Attacks targeting the site, such as dictionary attacks, cannot be used on a site only protected with OAuth. Users only will need to remember their login to the third-party service, making it easier for them to log in - additionally the details for some services such as Google are fetched automatically from the current active session with that service, meaning login can be done seamlessly in just two clicks. OAuth does not require a separate piece of hardware or app to be installed to work, only another account that the user almost certainly already uses. OAuth would not be suitable for systems that require a high degree of security, such as a bank account, but for a blog or forum it is acceptable. Further to that, there are already sites such as Quora which offer OAuth as a login option alongside username and password, which for me proves the suitability of OAuth in this field. Finally, although it is a newer authentication compared to others I have examined in this report, OAuth is easy enough to use that it should not put off new users from registering their account as the process for doing so is almost as easy as possible.

References

Aboukir, Y. (2012a). Cookie-based sql injection. **URL:** https://resources.infosecinstitute.com/topic/cookie-based-sql-injection/.

Aboukir, Y. (2012b). Sql injection through http headers. **URL:** https://resources.infosecinstitute.com/topic/sql-injection-http-headers/.

Adrián S (2016). All combination of the android pattern. **URL:** https://www.youtube.com/watch?v=TyCAialVM2M.

Alwan, Z. (2017). Detection and prevention of sql injection attack: A survey. *International Journal of Computer Science and Mobile Computing*, 6(8):5–17.

Aviv, A. J., Davin, J. T., Wolf, F., and Kuber, R. (2017). Towards baselines for shoulder surfing on mobile authentication. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 486–498.

Aviv, A. J., Gibson, K. L., Mossop, E., Blaze, M., and Smith, J. M. (2010). Smudge attacks on smartphone touch screens. *Woot*, 10:1–7.

Borland, M. (2002). Advanced sql command injection: Applying defense-in-depth practices in web-enabled database applications.

Chromium (2021). Samesite updates. **URL:** https://www.chromium.org/updates/same-site.

Clarke, J. (2012). *SQL Injection Attacks and Defense*. Syngress.

EFF (2016). Encrypting the web. **URL:** https://www.eff.org/encrypt-the-web.

Google (2020). Using oauth 2.0 to access google apis. **URL:** https://developers.google.com/identity/protocols/oauth2.

Goyal, A. (2008). Are stored procedures safe against sql injection? **URL:** https://anubhavg.wordpress.com/2008/02/01/are-stored-procedures-safe-against-sql-injection/.

Graceful, H. (2016). An introduction to dom xss. **URL:** https://gracefulsecurity.com/an-introduction-to-dom-xss/.

Hacksplaining (n.d.a). Avoiding user enumeration. **URL:** https://www.hacksplaining.com/prevention/user-enumeration.

Hacksplaining (n.d.b). Hacksplaining: Lessons. **URL:** https://www.hacksplaining.com/lessons.

Hacksplaining (n.d.c). Preventing malicious redirects. **URL:** https://www.hacksplaining.com/prevention/open-redirects.

Hacksplaining (n.d.d). Protecting against sql injection. **URL:** https://www.hacksplaining.com/prevention/sql-injection.

Hacksplaining (n.d.e). Protecting your users against cross-site scripting. **URL:** https://www.hacksplaining.com/prevention/xss-stored.

Hacksplaining (n.d.f). Protecting your users against reflected xss. **URL:** https://www.hacksplaining.com/prevention/xss-reflected.

Hacksplaining (n.d.g). Protecting your users against session fixation. **URL:** https://www.hacksplaining.com/prevention/session-fixation.

Hacksplaining (n.d.h). Secure treatment of passwords. **URL:** https://www.hacksplaining.com/prevention/password-mismanagement.

Hammer, E. (2012). Oauth 2.0 and the road to hell. **URL:** https://web.archive.org/web/20130325140509/http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/.

Hitaj, B. (2019). Passgan: A deep learning approach for password guessing. *Applied Cryptography and Network Security Lecture Notes in Computer Science*, page 217–237.

KirstenS (n.d.). Cross site request forgery (csrf). **URL:** https://owasp.org/www-community/attacks/csrf.

Kurobeats (2017). Xss vectors cheat sheet. **URL:** https://gist.github.com/kurobeats/9a613c9ab68914312cbb415134795b45.

Løge, M. (2015). Tell me who you are and i will tell you your unlock pattern. Master's thesis, Norwegian University of Science and Technology.

Milka, G. (2018). The anatomy of account takeover. **URL:** https://www.usenix.org/sites/default/files/conference/protected-files/enigma18_milka.pdf.

Mitre (2020). Mitre cve search - enumeration. **URL:** http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=enumeration.

NCSC (2018). Password administration for system owners. **URL:** https://www.ncsc.gov.uk/collection/passwords/updating-your-approach.

NIST (2017). Nist special publication 800-63b - digital identity guidelines. **URL:** https://pages.nist.gov/800-63-3/sp800-63b.html.

OWASP (2019). Session hijacking attack. **URL:** https://owasp.org/www-community/attacks/Session_hijacking_attack.

OWASP (2020a). Owasp top ten. **URL:** https://owasp.org/www-project-top-ten/.

OWASP (2020b). Testing for account enumeration and guessable user account. **URL:** https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account.

OWASP (n.d.a). Cross-site request forgery prevention cheat sheet. **URL:** https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

OWASP (n.d.b). Cross site scripting prevention cheat sheet. **URL:** https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

OWASP (n.d.c). Preventing unvalidated redirects and forwards. **URL:** https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html#preventing-unvalidated-redirects-and-forwards.

OWASP (n.d.d). Sql injection prevention cheat sheet. **URL:** https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.

Pound, M. (2016). Password cracking - computerphile. **URL:** https://youtu.be/7U-RbOKanYs?t=561.

Poza, D. (2021). Nist password guidelines and best practices for 2020. **URL:** https://auth0.com/blog/dont-pass-on-the-new-nist-password-guidelines.

Prodromou, A. (2019). Exploiting sql injection: a hands-on example. **URL:** https://www.acunetix.com/blog/articles/exploiting-sql-injection-example/.

PT (2019). How to prevent sql injection attacks. **URL:** https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks/#8.

Schneider, C. (n.d.). Unauthenticated session fixation attacks. **URL:** https://christian-schneider.net/UnauthenticatedSessionFixationAttacks.html.

Shepherd, J. (2019). The ultimate authentication playbook. **URL:** https://www.okta.com/blog/2019/02/the-ultimate-authentication-playbook/.

Tahiri, S. (2013). Android forensics: Cracking the pattern lock protection. **URL:** https://resources.infosecinstitute.com/topic/android-forensics-cracking-the-pattern-lock-protection/.

Taylor, S. (2020). Balancing ux and security at auth0. **URL:** https://auth0.com/blog/balancing-ux-and-security-at-auth0/.

True, R. (2021). Tweet by @racheltrue: 12:39 am · feb 27, 2021. **URL:** https://twitter.com/RachelTrue/status/1365461618977476610.

WHATWG (2021). Html living standard. **URL:** https://html.spec.whatwg.org/#charset.

A Appendix A

A.1 Account Enumeration Algorithm

Algorithm 1 authenticateUser(u, p) **return** s

Require: strings u and p, the username and password.
Ensure: s, a boolean indicating if the user is authenticated.

```
 1: $s \leftarrow$ false
 2: $t \leftarrow$ currentTime()
 3: $s \leftarrow$ getSalt($u$)        ▷ return the user's salt from the database or null if not found
 4: if $s :=$ not null then
 5:     $p \leftarrow p + s$
 6:     $p \leftarrow$ applyHash($p$)
 7:     $q \leftarrow$ getPassword($u$)        ▷ return the user's (hashed) password from the database
 8:     if $p := q$ then
 9:         $s \leftarrow$ true
10: $t \leftarrow$ currentTime() $-t$        ▷ get the difference in time
11: system.pause($1000 - t$)        ▷ assuming hash + lookup take less than a second
12: return $s$
```

A.2 Session Hijacking Algorithms

Algorithm 2 createSessionToken(u, i) **return C**

Require: strings u, the current user id and ip address.
Ensure: s, a cookie of the current session token.

```
1: $t \leftarrow$ getCurrentTime()
2: $k \leftarrow$ rand(64)        ▷ random 64-bit long hex secret
3: $h \leftarrow u + i + t + k$
4: $h \leftarrow$ applyHash($h$)
5: $\mathbf{C} \leftarrow$ createCookie($h$, $t$)
6: storeKey($u$, $i$, $k$)
7: return $\mathbf{C}$
```

Algorithm 3 checkSessionToken(**C**, u, i) **return** v

Require: cookie **C** containing the current session and strings u and i, the user id and ip.
Ensure: v, a boolean indicating if the token is valid.

```
 1: $v \leftarrow$ false
 2: $t \leftarrow \mathbf{C}$.time        ▷ the time variable from the cookie
 3: if $t + 1800 <$ getCurrentTime() then        ▷ check if the 30 minute token has expired
 4:     $k \leftarrow$ getKey($u$, $i$)
 5:     if $k :=$ not null then
 6:         $h \leftarrow u + i + t + k$
 7:         $h \leftarrow$ applyHash($h$)
 8:         if $h := \mathbf{C}$.hash then
 9:             $v \leftarrow$ true
10: return $v$
```

Algorithm 4 refreshSessionToken($\mathbf{C}$, u, i) **return D**

```
Require: cookie $\mathbf{C}$ containing the current session and strings $u$ and $i$, the user id and ip.
Ensure: $\mathbf{D}$, a cookie containing the new session token.
 1: $t \leftarrow \mathbf{C}$.time
 2: $k \leftarrow$ getKey($u$, $i$)
 3: $h \leftarrow u + i + t + k$
 4: $h \leftarrow$ applyHash($h$)
 5: if $h$ := $\mathbf{C}$.hash then                ▷ if we verify the old session was valid
 6:     $t \leftarrow$ getCurrentTime()
 7:     $k \leftarrow$ rand(64)                     ▷ random 64-bit long hex secret
 8:     $h \leftarrow u + i + t + k$
 9:     $h \leftarrow$ applyHash($h$)
10:     $\mathbf{C} \leftarrow$ createCookie($h$, $t$)
11:     storeKey($u$, $i$, $k$)                     ▷ replace the old key with a new one
12:     return C
13: return null
```

A.3 SQL Injection Algorithm

Algorithm 5 getUser(u) **return R**

```
Require: string $u$, the user to be found in the database.
Ensure: $\mathbf{R}$, the record of user $u$ or null.
 1: $\mathbf{R} \leftarrow$ null
 2: $u$ := removeChars($u$, "'-;")          ▷ remove some characters that can cause an SQLi
 3: $\mathbf{C}$ := connect(dbLookup)       ▷ db account that can only parse SELECT statements
 4: $\mathbf{C}$.execute('SELECT * FROM users WHERE username=?', $u$)
 5: return $\mathbf{C}$.fetchone()
```

A.4 Cross-site Scripting Algorithm

Algorithm 6 htmlValidateInput(i) **return** o

```
Require: string $i$, given input from the user.
Ensure: $o$, string of the output going back to HTML.
 1: $\mathbf{B} \leftarrow$ [&, <, >, ", ', %, *, +, „ -, /, ;, =, ^, |]
 2: $o \leftarrow$ null
 3: for each char $c \in i$ do
 4:     if $c \in \mathbf{B}$ then
 5:         $c$ := encodeChar($c$)          ▷ replace the given character with its HTML counterpart
 6:     $o$.add($c$)
    return $o$
```

A.5 Cross-site Request Forgery Algorithms

Algorithm 7 createCSRFToken(i) **return C**

Require: string i, the user's session ID.
Ensure: **C**, a valid encrypted CSRF Token (to be embedded in a form)

```
1: $t \leftarrow$ getCurrentTime()
2: $h \leftarrow i + t$
3: $h \leftarrow$ applyHash($h$)
4: $s \leftarrow$ rand(64)                    ▷ random 64-bit long hex secret
5: $\mathbf{C} \leftarrow$ signToken($h$, $s$)          ▷ sign the payload and secret
6: storeKey($h$, $s$)
7: $\mathbf{C} \leftarrow \mathbf{C} + t$        ▷ store the current time with the signed token
8: return C
```

Algorithm 8 verifyCSRFToken(i, **C**) **return** v

Require: string i, the user's session ID **C**, the user's signed CSRF Token.
Ensure: v, a boolean value indicating if the token is valid.

```
1: $v \leftarrow$ false
2: $t \leftarrow \mathbf{C}$.time                    ▷ the time variable from the token
3: if $t + 600 <$ getCurrentTime() then   ▷ check if the 10 minute token has expired
4:     $s \leftarrow$ getKey($h$)
5:     if $s :=$ not null then
6:         $d \leftarrow$ decryptToken(C, $s$)          ▷ decrypt token using secret
7:         $t \leftarrow$ getCurrentTime()
8:         $h \leftarrow i + t$
9:         $h \leftarrow$ applyHash($h$)
10:        if $h := d$ then
11:            $v \leftarrow$ true
12: return $v$
```

A.6 Unvalidated Redirects Algorithm

Algorithm 9 mapRedirect(i) **return** a

Require: string i, given redirect value from the URL
Ensure: a, actual page address the user will be redirected to

```
1: R ← null      ▷ map of values to addresses (these would be held as EnvVars on the system)
2: R.add("4Q1GQZpCfGR9x2Ut", "login.php")
3: R.add("cOOpwgkFjDCtSMHE", "subsite.thissite.com")
4: if $i \in \mathbf{R}$ then
5:     $a \leftarrow \mathbf{R}$.value($a$)
6: else
7:     $a \leftarrow$ "errorpage.html"
   return $a$
```

B Appendix B

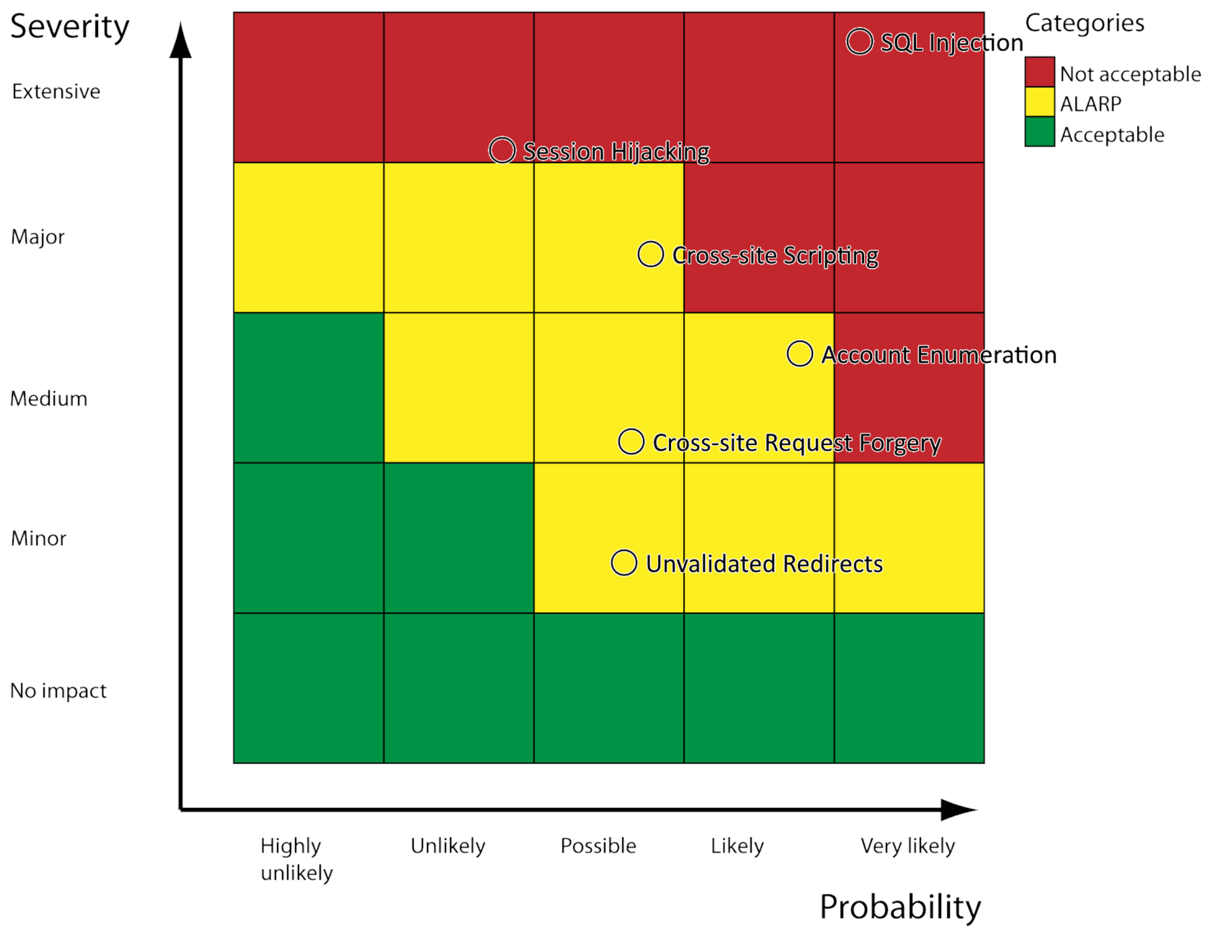

Figure 1: Risk matrix of probability of attack vs severity for different vulnerabilities