When building a web application such as this one it is imperative to consider the security techniques and aspects that will be used to deter any users for using it for their own malicious purposes, as well as to preclude any attempts to do so. To this end, it has been deemed necessary to investigate what is considered the 3 most potent threats to the security of a web application that has been created in PHP, as well as techniques that can be implemented to reduce the chance of these threats being utilised and even remove them entirely.

The first, and debatably the most potent, of these potential threats is something called SQL Injection (David Shirey, 2012), which has been used to gain access to the servers of multiple major organisations such as the Wall Street Journal in 2014 (Jeremy Kirk, 2014) and even the World Trade Organisation in 2015 (Pierluigi Paganini, 2015). An SQL Injection attack occurs when the user enters SQL commands, such as the “SELECT” command, into an input box on the web application in an attempt to manipulate the database that is associated with the web application to give themselves access to it, bypassing any security and authentication that may have been put in place on the database. Naturally, the user gaining unauthorised access to the web application’s associated database via SQL injection is exceptionally bad and is definitely something that will want to be avoided.

There are a number of security techniques that can be utilised in the prevention of SQL injection attacks, each with varying degrees of difficulty and effectiveness. However, it is widely agreed that the most effective countermeasure to these attacks is the usage of a “prepare” statement. This is due to the fact that using a prepare statement will separate out the query that is being sent to the SQL server, and the data that the user is attempting to enter into the server. This will ensure that, if the user starts entering SQL commands into the input box, these commands won’t be sent to the server as a part of the query and thus executed. Instead, they will be sent to the server as the data that will be used by the query which will have already been sent to the server, meaning any commands the user has entered will not be carried out.

The second of the potential threats that could be utilised against the web application for malicious purposes is known as Cross Site Scripting (XXS) (WordFence, 2018), a famous, albeit harmless example of this being the Samy worm that was used on Myspace in 2005 (Samy, 2005). Cross site scripting works in a similarly to the aforementioned SQL Injection where the user injects their own malicious code into the web application, usually in the form of JavaScript. However, unlike SQL injection attacks, these attacks affect the web application’s users rather than the web application itself as the malicious script will be run every time they open the web page. How this occurs depends on the type of Cross Site Scripting. A reflected XSS attack occurs when the malicious code is “reflected” off the server being used to host the application and is sent to the user as part of a request, for example, the user opening a malicious link in a phishing email (OWASP, 2018). Whereas, a stored XSS attack occurs when the malicious code is stored in the web server itself, usually in a database, to be sent out to any users that attempt to access the web application (Margaret Rouse, 2018).

There are 3 main techniques and methods that can be employed when attempting to prevent cross site scripting: Escaping or Validating the User Input and Sanitization (Sarah Vonnegut, 2017). The first of these techniques, escaping, involves replacing the potentially harmful characters, for example <, with their equivalent in ASCII characters to prevent them being inserted into the HTML or JavaScript of the application for malicious purposes (Richard Ishida, 2010). The second technique, validating user input, is essentially filtering out any characters that could potentially be used for malicious purposes from a user’s input. For example, filtering out the characters < and > from a text box used to submit a user’s name as those characters are not used within first names or last names and could potentially be used in malicious script that the user may be attempting to inject into the web application. Naturally, these filters will need to be context sensitive and allow the usage of certain characters in certain scenarios, eg the @ character for an email address. The final technique, sanitization, is rather similar to the second technique but with one key difference. Instead of simply filtering out the potentially malicious characters, sanitization “corrects” the user’s input by automatically removing or changing any of these characters (oracle, No Date Given).

The final threat of what is considered the 3 most severe threats to a web application’s security is called session hijacking (David Shirey, 2012). As the threat’s name implies, session hijacking involves taking control of a user’s session through their session ID and then pretending to be said user, allowing all sorts of malicious things to be done to the user’s account or the web application itself if they turn out to be an administrator (Margaret Rouse, 2006). There are 2 main ways in which the session ID of a user can be obtained for a session hijack: session sniffing and cross-site scripting. Cross site scripting is simply using the previous security threat to access the cookie that contains the session ID of the user, thus allowing the session to be hijacked. Naturally this is particularly worrisome if the session ID is being obtained through stored XSS as that will enable the hijacking of every session that is using the server. The other way the ID can be obtained is by using something called a sniffer to search for sessions that are connected to the web server, then attempting to “capture” the session ID before it can be sent to the web server (OWASP, 2014).

The most effective way to prevent session hacking is to prevent it before it can occur, that is, prevent the hacker from accessing the session ids of the web application’s users. Naturally, by protecting the web application from cross site scripting using the previously mentioned methods, you will also be protecting its users from one of the methods used to obtain their session ids. As such, the only other way the user’s session ID could potentially be obtained is through the usage of a sniffer. However, this can also be quite easily prevented by simply encrypting the cookie and the session ID that it contains. This means that, even if the cookie was captured by someone before it reached the web server, said person would not be able to use it as they would not have the decryption key required to decrypt the encryption on the cookie and thus view the session id (Mike Chapple, 2009).

While each of these techniques are, by themselves, not particularly effective at preventing all of the threats to the web application’s security, when used in conjunction with one another, they prove to be an effective prevention method and deterrent for anyone that would potentially make and malicious changes to the web application.