EXPERIMENT 05

Aim: Study of OWASP vulnerabilities.

To do:

1. Injection Mitigation
2. Top 10 OWASP vulnerabilities and their solutions

Theory:

What is Injection Mitigation?

SQL injection is a type of cyber attack that allows attackers to execute malicious SQL statements on a database. These statements can be used to manipulate data, retrieve sensitive information, or even delete entire databases. It is one of the most common and dangerous types of web vulnerabilities, and it can affect any website or web application that uses a SQL database.

Mitigation Strategies:

1. **Secure Coding & SDLC**

   Security driven programming practices will always be the best defense against SQL Injection attacks. Ensuring developers are aware of the risks, tools, and the techniques which can mitigate SQL vulnerabilities is your first and best line of defense. However, cultivating the use of secure programming techniques will also require a commitment to their implementation throughout the SDLC. Developing security minded education, planning, testing, and review practices are just a few components within an SDLC that will help prevent SQL Injection vulnerabilities from making their way into your application.

2. **Input Validation & Sanitation**

   Client side input sanitization and validation should only be considered a convenience for the end user, improving their user experience. For example, it could prove useful to provide feedback on a proposed username, indicating whether or not it will meet the application's criteria. However, client side sanitization and validation can be bypassed, and as such, server side solutions should be employed.

   Server side sanitization and input validation ensures data supplied by the user does not contain characters like single or double quotes that could modify an SQL query and return data not originally intended in the application's design. Specifically, validation makes sure that user supplied data satisfies an application's criteria while sanitization refers to the process which modifies user input in order to satisfy the criteria established by validation. Combining both results in a scenario where single quotations contained within a user submitted string are modified or removed as a result of sanitization and then

validated ensuring single quotations are no longer present satisfying the application's requirements.

3. **Stored Procedures & Parametrization**

   Query parameterization occurs when stored procedures, defined as sub-routines an application uses to interact with a relational database management system (RDBMS), employ variable binding. Variable binding is a process which requires the definition of an SQL statement prior to the insertion of variables allowing for a clear delineation of code and user input. Essentially, prepared statements which parameterized queries protect the intent of an SQL query.

4. **Prepared Statements**

   Prepared statements are essentially server side parameterized queries and when used with secure coding techniques, can produce equally secure code. Simply, the construction of a secure prepared statement results in the automatic parametrization of user input. However, the use of dynamically constructed queries should be avoided unless special libraries and techniques are used to protect against gaps in security coverage that might emerge. Libraries like opaleye for Haskell and SQL Builder for Python can be used to this effect. However, if dynamic SQL must be used proper sanitization and validation of user input will be necessary to safeguard an application.

What is OWASP?

The Open Web Application Security Project, or OWASP, is an international non-profit organization dedicated to web application security. One of OWASP's core principles is that all of their materials be freely available and easily accessible on their website, making it possible for anyone to improve their own web application security. The materials they offer include documentation, tools, videos, and forums.

OWASP Vulnerabilities & their Solutions:

1. **Injection**

   Injection attacks happen when untrusted data is sent to a code interpreter through a form input or some other data submission to a web application. For example, an attacker could enter SQL database code into a form that expects a plaintext username. If that form input is not properly secured, this would result in that SQL code being executed. This is known as an SQL injection attack.

   Injection attacks can be prevented by validating and/or sanitizing user-submitted data. (Validation means rejecting suspicious-looking data, while sanitization refers to cleaning up the suspicious-looking parts of the data.) In addition, a database admin can set controls to minimize the amount of information an injection attack can expose.

2. **Broken Authentication**

Vulnerabilities in authentication (login) systems can give attackers access to user accounts and even the ability to compromise an entire system using an admin account. For example, an attacker can take a list containing thousands of known username/password combinations obtained during a data breach and use a script to try all those combinations on a login system to see if there are any that work.

Some strategies to mitigate authentication vulnerabilities are requiring two-factor authentication (2FA) as well as limiting or delaying repeated login attempts using rate limiting.

3. **Sensitive Data Exposure**

If web applications don't protect sensitive data such as financial information and passwords, attackers can gain access to that data and sell or utilize it for nefarious purposes. One popular method for stealing sensitive information is using an on-path attack.

Data exposure risk can be minimized by encrypting all sensitive data as well as disabling the caching* of any sensitive information. Additionally, web application developers should take care to ensure that they are not unnecessarily storing any sensitive data.

Caching is the practice of temporarily storing data for reuse. For example, web browsers will often cache web pages so that if a user revisits thosepages within a fixed time span, the browser does not have to fetch the pages from the web.

4. **XML External Entities (XEE)**

This is an attack against a web application that parses XML* input. This input can reference an external entity, attempting to exploit a vulnerability in the parser. An 'external entity' in this context refers to a storage unit, such as a hard drive. An XML parser can be duped into sending data to an unauthorized external entity, which can pass sensitive data directly to an attacker.

The best ways to prevent XEE attacks are to have web applications accept a less complex type of data, such as JSON**, or at the very least to patch XML parsers and disable the use of external entities in an XML application.

*XML or Extensible Markup Language is a markup language intended to be both human-readable and machine-readable. Due to its complexity and security vulnerabilities, it is now being phased out of use in many web applications.

**JavaScript Object Notation (JSON) is a type of simple, human-readable notation often used to transmit data over the internet. Although it was originally created for JavaScript, JSON is language-agnostic and can be interpreted by many different programming languages.

5. **Broken Access Control**

Access control refers to a system that controls access to information or functionality. Broken access controls allow attackers to bypass authorization and perform tasks as though they were privileged users such as administrators. For example a web application could allow a user to change which account they are logged in as simply by changing part of a url, without any other verification.

Access controls can be secured by ensuring that a web application uses authorization tokens* and sets tight controls on them.

*Many services issue authorization tokens when users log in. Every privileged request that a user makes will require that the authorization token be present. This is a secure way to ensure that the user is who they say they are, without having to constantly enter their login credentials.

6. **Security Misconfiguration**

Security misconfiguration is the most common vulnerability on the list, and is often the result of using default configurations or displaying excessively verbose errors. For instance, an application could show a user overly-descriptive errors which may reveal vulnerabilities in the application. This can be mitigated by removing any unused features in the code and ensuring that error messages are more general.

7. **Cross-Site Scripting**

Cross-site scripting vulnerabilities occur when web applications allow users to add custom code into a url path or onto a website that will be seen by other users. This vulnerability can be exploited to run malicious JavaScript code on a victim's browser. For example, an attacker could send an email to a victim that appears to be from a trusted bank, with a link to that bank's website. This link could have some malicious JavaScript code tagged onto the end of the url. If the bank's site is not properly protected against cross-site scripting, then that malicious code will be run in the victim's web browser when they click on the link.

Mitigation strategies for cross-site scripting include escaping untrusted HTTP requests as well as validating and/or sanitizing user-generated content. Using modern web development frameworks like ReactJS and Ruby on Rails also provides some built-in cross-site scripting protection.

8. **Insecure Deserialization**

This threat targets the many web applications which frequently serialize and deserialize data. Serialization means taking objects from the application code and converting them into a format that can be used for another purpose, such as storing the data to disk or streaming it. Deserialization is just the opposite: converting serialized data back into objects the application can use. Serialization is sort of like packing furniture

away into boxes before a move, and deserialization is like unpacking the boxes and assembling the furniture after the move. An insecure deserialization attack is like having the movers tamper with the contents of the boxes before they are unpacked.

An insecure deserialization exploit is the result of deserializing data from untrusted sources, and can result in serious consequences like DDoS attacks and remote code execution attacks. While steps can be taken to try and catch attackers, such as monitoring deserialization and implementing type checks, the only sure way to protect against insecure deserialization attacks is to prohibit the deserialization of data from untrusted sources.

9. **Using Components With Known Vulnerabilities**

Many modern web developers use components such as libraries and frameworks in their web applications. These components are pieces of software that help developers avoid redundant work and provide needed functionality; common examples include front-end frameworks like React and smaller libraries that used to add shared icons or a/b testing. Some attackers look for vulnerabilities in these components which they can then use to orchestrate attacks. Some of the more popular components are used on hundreds of thousands of websites; an attacker finding a security hole in one of these components could leave hundreds of thousands of sites vulnerable to exploitation.

Component developers often offer security patches and updates to plug up known vulnerabilities, but web application developers don't always have the patched or most-recent versions of components running on their applications. To minimize the risk of running components with known vulnerabilities, developers should remove unused components from their projects, as well as ensuring that they are receiving components from a trusted source and ensuring they are up to date.

10. **Insufficient Logging And Monitoring**

Many web applications are not taking enough steps to detect data breaches. The average discovery time for a breach is around 200 days after it has happened. This gives attackers a lot of time to cause damage before there is any response. OWASP recommends that web developers should implement logging and monitoring as well as incident response plans to ensure that they are made aware of attacks on their applications.

Conclusion:

In this experiment we studied various OWASP vulnerabilities.