**3.4**

**Describe how to prevent XSS flaws in a web application.**

Input Validation and Sanitization:

* Always validate and sanitize user inputs on both the client and server sides.
* Use a whitelist approach, allowing only known and expected input. Reject or filter out any input that doesn't match the expected format or content.
* Before rendering user-generated content, escape it to ensure that any potentially malicious code is treated as plain text, rather than executable code.
* Use appropriate encoding functions provided by your web framework or library (e.g., htmlspecialchars in PHP, encodeURIComponent in JavaScript).

**What can be done by a user (on the client- (browser-) side to protect against XSRF?**

Use a Modern Browser: Ensure you are using an up-to-date and well-maintained web browser.

Log Out: After using a website or web application, always log out to terminate your session. This prevents XSRF attacks from abusing your authenticated session.

Use Strong Passwords: Ensure that you have strong, unique passwords for all your online accounts to reduce the impact of any successful CSRF attack.

Be Cautious with Links and Emails:

* Avoid clicking on links or opening email attachments from untrusted sources. Before clicking on a link in an email or on a web page, hover over it to see the destination URL. Make sure it's what you expect. Double-check the URL in your browser's address bar when visiting sensitive websites. Make sure it matches the official domain.

**Name at least two alternatives. How can you prevent XSRF vulnerabilities on the server-side?**

Use Anti-CSRF Tokens:

Implement anti-CSRF tokens in your web application. These tokens are unique for each user session and are generated on the server side. Include these tokens in forms, requests, or any action that changes the state of the application, such as changing passwords or making financial transactions.

Ensure Stateless Requests: Make sure that all state-changing requests require authentication and that they are stateless. In other words, every request that modifies data or performs an action should require the presence of a valid anti-CSRF token.

**4.4 Questions**

**Describe how parameterized queries help protect against SQL injection. What other protection techniques can be used on the application’s code level?**

Parameterized queries help protect against SQL injections by separating SQL code from user input, making it much more difficult for attackers to inject malicious SQL code into the application. One example is automatic data sanitization: When user inputs are bound to parameters, the database driver automatically handles data sanitization and escaping. This means that user input is treated as data, not code, reducing the risk of SQL injection.

SQL injection attacks typically involve trying to inject malicious SQL code into the user inputs to manipulate the query's behavior. With parameterized queries, this becomes extremely difficult because the user input is not interpreted as SQL code. Even if an attacker attempts to inject SQL code, it is treated as data and doesn't have the desired effect.

Other protection techniques:

* Input Validation: Validate and sanitize all user inputs on both the client and server sides. Ensure that input data adheres to expected formats and constraints. Reject or sanitize any input that doesn't meet these criteria.
  + While sanitation is negligent to forego and in theory sufficient protection by itself, flaws in sanitation tools are often responsible for vulnerabilities and should not be relied upon.
* Principle of Least Privilege: Limiting database user permissions to the minimum required for the application to function and avoiding using highly privileged accounts for routine database operations.
  + Though an important tool to mitigate injection attacks that do occur, the comparatively rough nature of database permissions means there will always be some damage at any (useful) level thereof.
* Error Handling: Implementing custom error handling that doesn't reveal sensitive information about the database structure or queries to users, and using generic error messages instead of detailed ones.
  + At the same time, make sure error messages provide enough information that users can submit useful reports of them.

**What is a stored procedure? Describe how it protects against SQL injection. What other protection techniques can be used at the database level?**

* A stored procedure is a precompiled and stored SQL code block within a database management system (DBMS). It is a set of one or more SQL statements that are defined and stored in the database and can be executed repeatedly with different parameters. Stored procedures are typically used to encapsulate and execute frequently used database operations, such as querying, data manipulation, or complex transactions.
* When you require the application to use stored procedures, you limit the direct access that application code has to the underlying database tables and schema. This reduces the attack surface because the application interacts with the database through the stored procedures, and attackers cannot manipulate the SQL queries directly, even with control of the application.
* This comes at the cost of some flexibility in the application, since such a requirement means it cannot use arbitrary parameterized queries.

Other protection techniques:

* Database Auditing and Logging: Enable auditing and logging features provided by the DBMS to monitor and track database activity. This helps in identifying and investigating potential security incidents or unauthorized access.
* Database Encryption: Implement encryption at rest and in transit to protect data from unauthorized access.
  + For the mechanism of encryption to provide actual security it must be carefully designed, and weighed against the availability of backups and the like.

**What combination of protection techniques would you choose, to mitigate the SQL injection threat?**

To effectively mitigate the SQL injection threat, it's important to adopt a multi-layered approach, combining protection techniques at both the application and database levels as far as possible. We would probably use all of the above techniques in order to make the application as secure as possible.

At the application level:

* Parameterized Queries (if available)
* Input validation
* Principle of Least Privilege

At the database level:

* Implement stored procedures to encapsulate and execute database operations.
* Database Auditing and Logging
* Database Encryption