**INPUT VALIDATION**

There are several ways that an attacker can include ***malicious*** code in a SQL statement of a web page, including:

* Injection through user input: An attacker can include malicious SQL code into the application by providing user input that is not properly validated or sanitized.
* Injection through cookies or headers: An attacker can manipulate cookies of HTTP headers to inject malicious code into the SQL statement.
* Injection through hidden form field. Attackers can insert hidden form fields containing malicious code into web pages which may then be sent to the server along with legitimate user input.
* Injection through stored procedures: An attacker could exploit vulnerabilities in stored procedures to execute arbitrary SQL code in the target database.
* Injection through third-party components: A third-party component may have a SQL injection vulnerability, which attackers can exploit to inject their own malicious code.

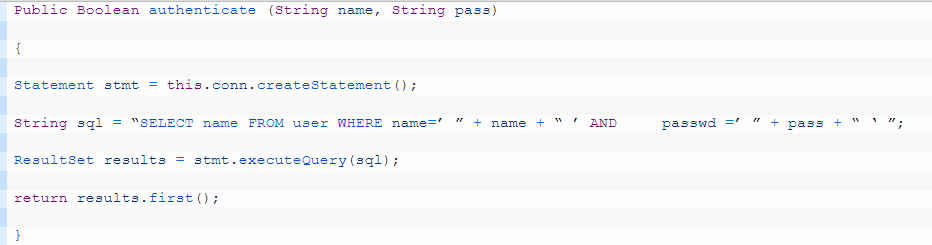
**Input validation** is a type of technique used to ensure that user input is safe and secure. It’s used in both BE and FE systems to prevent common types of attacks like SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). While input validation can be bypassed by skilled attackers, it’s an important layer of security that should always be included in applications.

1. **SQL INJECTION**

SQL injection is a major concern when developing a Web application. It occurs when the application accepts a malicious user input and then uses it as a part of SQL statement to query a backend database.

An attacker can inject SQL control characters and command keywords (e.g., single quote (‘), double quote (“), equal (=), comment (- -), etc.) to change the query structure. Using these control characters with common SQL commands (e.g., SELECT, FROM, DELETE, etc.) enables access or retrieval of data elements from a backend database server.

A successful attack requires a Web application to include malicious code from an attacker in a SQL statement. The malicious code usually comes from an untrusted source. In some cases, internal system databases can also be the source of malicious data. When malicious SQL statements execute against a backend database, an attacker can modify or access the database. This depends how the attacker crafts the malicious data.  
This attack example illustrates code to accept a username and password from a user. The user-provided input crafts a SQL query to execute against a database. The database includes a table named “user,” with columns for *name* and *password*.



Consider a user authenticating to an application with “admin” as a username and “xDK9&GoP1” as a password. These are valid credentials. During application login, follow the SQL statement that executes against the database server:

Now consider an attacker attempting to authenticate to an application using the “*password’ OR ‘a’=’a*” password value as the injection payload. During application login, following the SQL statement executes against the database server:

*SELECT name FROM user WHERE name=‘admin’ AND passwd=‘password’ OR ‘a’=‘a’*

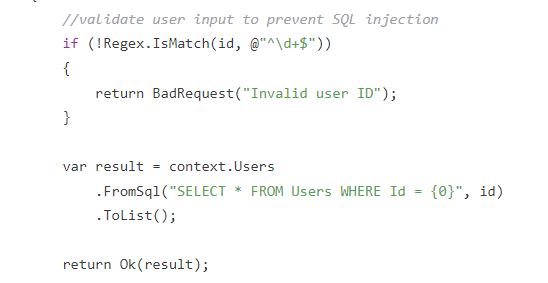
Upon execution of this query, an attacker successfully authenticates to an application since *‘a’=‘a’* always returns *true,* resulting in authentication bypass.

With a successful attack, an attacker can gain:

1. Unauthorized access to an application. An attacker can successfully bypass an application’s authentication mechanism to have illegitimate access to it.
2. Information disclosure*.* An attack could lead to a complete data leakage from the database server.
3. Loss of data availability. An attacker can delete records from the database server.
4. Compromised data integrity. As SQL statements are also used to modify or add the record, an attacker can use SQL injection to modify or add data stored in a database. This would lead to compromised data integrity.

***How can organizations mitigate SQL injection?***

Here is my code example:

****

The code uses a RegExpres pattern to validate that id only contains digits before proceeding with the database query.

1. **Cross-site scripting (XSS)**

Cross-site scripting (XSS) is a type of cyber attack that allows hackers to inject malicious code into web pages viewed by other users. The attacks can result in a variety of side effects, including stealing session cookies, redirecting users to malicious sites, and hijacking user accounts. To prevent XSS attacks, here are some steps you can take:

Input validation: Ensure all input from web forms or applications is properly validated and sanitized to prevent the injection of malicious scripts.

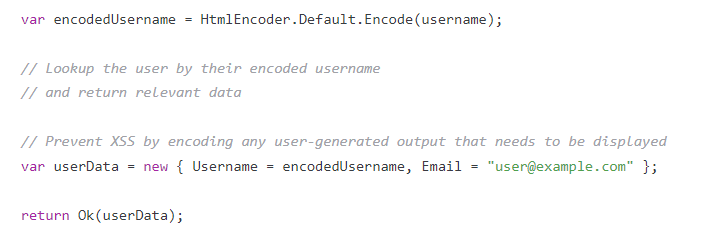
Encoding: Encode user-generated content before rendering it on a web page. HTML encoding replaces special characters with their corresponding escape sequences, preventing the browser from interpreting them as code.

Content Security Policy (CSP): Utilize CSP to specify valid sources of content injection within the application, reject unwanted or unsolicited resource requests.

HTTPS protocol: Using HTTPS helps secure communication between servers and clients.

Sanitize inputs: Use sanitizer libraries like Microsoft's AntiXss library to systematically scrub incoming input data for malicious input.

***Prevent XSS:***



In this example, when the GetUser action method receives a username parameter as a string from a GET request, it is first encoded using HTML encoding before being used for additional processing.

This step helps prevent malicious code input using parameters. The returned object userData contains only properly encoded values of sensitive or displayed data fields that are free of interference from any injected malicious code.

By using encoding everywhere that untrusted data is presented, organizations can help safeguard their web applications against XSS vulnerabilities. Other strategies such as Content Security Policy, HTTPS protocol enforcement and sanitation of incoming inputs can further reinforce Web APIs protection.

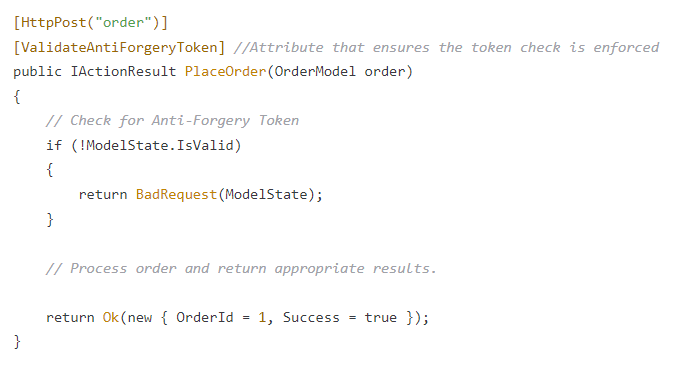
1. **Cross-site request forgery (CSRF)**

Cross-Site Request Forgery (CSRF) is a type of cyber attack that targets web applications by exploiting the trust relationship between a user's browser and a website. The attack involves tricking a user to unknowingly perform an action on a website using their active session credentials or cookies stored in their browser.

An attacker can lure a genuine user into clicking on a link or submitting a form that makes use of active authentication credentials. Once the user has done so, the attacker can inject malicious crafting that masquerades as legitimate input data that executes authorized actions.

Such attacks can cause a range of issues, including unauthorized transactions or tampering with user accounts. To mitigate such attacks, ensure that web applications have sufficient security checks and measures like token validation safeguarding requests handling.

***Prevent CSRF***

******

in this example, the ValidateAntiForgeryToken attribute is used to enforce a defense mechanism against CSRF.

By specifying the anti-forgery token as an attribute on the method invoked by suspicious POST requests, any submissions with missing, invalid or outdated tokens are rejected. The action method calls ModelState.IsValid to check for token validation for each incoming request.

Additionally, organizations should properly implement first party authentication mechanisms such as OAuth2 or JWT, secure cookie handling practices like HttpOnly cookie settings and other best practices for web application security.

By utilizing these measures in the implementation of Web APIs, sensitive actions - login, payment processing, and user account management- can be safeguarded from unauthorized access through CSRF exploits.