Secure Development Coding(Basics)Level 1

Slide 1: Introduction to Secure Coding

What is Secure Coding?

Example: Secure coding is like locking the doors and windows of your house to stop burglars from breaking in. In the same way, secure coding protects your application and data from attackers.

Why is it Important?

Example: Just as you lock your doors to keep your home safe, secure coding ensures your user data and system are safe from hackers who might try to steal or damage them.

Slide 2: Common Security Threats

1. Injection Attacks (e.g., SQL Injection)

Example: Imagine a customer sneaks a fake order to the chef to mess with the kitchen. In SQL Injection, hackers insert harmful code into your database to steal or manipulate your data.

2. Cross-Site Scripting (XSS)

Example: It’s like someone writing a bad message in a hotel guestbook for everyone to see. XSS is when hackers inject malicious code into a website that affects other users who view it.

3. Cross-Site Request Forgery (CSRF)

Example: A hacker pretends to be you and sends a fake request to your bank to steal your money. CSRF tricks a user into performing unintended actions on a website.

Slide 3: Protecting Against SQL Injection (ASP.NET, PHP, C#)

What is SQL Injection?

Example: Imagine a hacker sneaks into a restaurant's kitchen and changes the order list. SQL Injection allows hackers to manipulate your database queries to steal or change data.

How to Prevent it:

Example: Like using a fixed menu instead of letting customers make their own orders, parameterized queries ensure that database commands are safe.

Code Example (C#):

SqlCommand cmd = new SqlCommand("SELECT \* FROM Users WHERE Username = @username", connection); cmd.Parameters.AddWithValue("@username", username);

Slide 4: Input Validation and Sanitization (C#, JavaScript, PHP)

Never Trust User Input

Example: Letting customers write whatever order they want could lead to trouble. Always validate what users can input to ensure it's safe.

How to Validate?

Example: Just like you wouldn’t let someone order anything on the menu, you set rules for what users can input (e.g., only numbers for phone numbers or letters for names).

Code Example (C#):

if (!Regex.IsMatch(userInput, @"^[a-zA-Z0-9]\*$")) { throw new ArgumentException("Invalid input."); }

Slide 5: Password Handling and Authentication (ASP.NET, C#, PHP)

Secure Password Storage

Example: You keep valuables in a safe instead of a drawer, so no one can easily access them. In the same way, hashing passwords ensures that even if attackers steal the database, they can't read the passwords.

ASP.NET Example: Use ASP.NET Identity to securely hash and store passwords.

PHP Example:

password\_hash($password, PASSWORD\_BCRYPT);

Slide 6: Cross-Site Scripting (XSS) and How to Prevent It

What is XSS?

Example: It's like someone writes a nasty message in a party guestbook that other guests will see. XSS happens when attackers inject bad code into a website that affects other users.

How to Prevent it:

Example: Just like cleaning food before serving it, sanitize user input to ensure harmful scripts don’t get injected.

csharp

@Html.Encode(userInput)

Slide 7: Using HTTPS for Secure Communication (ASP.NET, PHP)

What is HTTPS?

Example: Sending a letter in an unsealed envelope means anyone could read it. HTTPS is like sealing the envelope so your message stays private.

Why Use HTTPS?

Example: HTTPS encrypts sensitive data (like passwords) so hackers can't steal it while it's being sent between the server and the user.

ASP.NET Example:  
Use RequireHttpsAttribute to make sure the site only uses HTTPS.

Slide 8: Session Management (ASP.NET, PHP)

What is a Session?

Example: A concert ticket lets you into the event and proves you're the rightful user. Sessions are used to keep track of who’s logged in and protect that information.

Best Practices:

Example: Keep your ticket safe by using HttpOnly cookies for storing session IDs, and set timeouts to log out users after inactivity.

Slide 9: Handling Errors Securely (C#, ASP.NET, PHP)

Avoid Information Leakage

Example: Don’t tell anyone how your store's security system works, or they might break in. Similarly, don’t expose detailed error messages that could give attackers clues about how your system works.

Best Practices:

Example: Show general error messages to users (like "Something went wrong") but log detailed errors for developers to fix.

ASP.NET Example:  
Use customErrors in web.config to hide detailed errors from users.

Slide 10: Secure Code Culture

Security is Everyone's Responsibility

Example: Just like in a sports team, everyone needs to play their part. Developers, testers, and managers must all care about security.

Key Points:

Example: Regular code reviews help catch mistakes early, like checking your map before a trip. Keep learning about new security practices, just like learning the rules of a game.

**Assessment Questions**

**Question 1: What is Secure Coding?**

A) Writing code as fast as possible.

B) Ensuring that an application is free of bugs.

C) Writing code to protect applications and data from security threats.

D) Writing code that only works in specific environments.

Answer: **C) Writing code to protect applications and data from security threats.**

**Question 2: Why is Secure Coding Important?**

A) To make the code faster.

B) To prevent data loss, breaches, and attacks.

C) To reduce the size of the application.

D) To make the code more complex.

**Answer: B) To prevent data loss, breaches, and attacks.**

**Question 3: What is an Injection Attack (e.g., SQL Injection)?**

A) A type of malware that damages software.

B) A method where hackers inject harmful code into a website's database.

C) A way of encrypting data to make it unreadable.

D) A technique to speed up website performance.

**Answer: B) A method where hackers inject harmful code into a website's database.**

**Question 4: What is Cross-Site Scripting (XSS)?**

A) A technique used to improve website design.

B) A method of storing encrypted passwords.

C) A vulnerability where hackers inject malicious scripts into websites.

D) A way to make websites run faster.

**Answer: C) A vulnerability where hackers inject malicious scripts into websites.**

**Question 5: How can SQL Injection be Prevented?**

A) By allowing users to directly input database queries.

B) By using parameterized queries or Object-Relational Mapping (ORM).

C) By storing passwords in plain text.

D) By disabling all user input.

**Answer: B) By using parameterized queries or Object-Relational Mapping (ORM).**

**Question 6: What is the Best Way to Validate User Input?**

A) Accept all user input without any validation.

B) Use regular expressions or allow-lists to only accept valid data.

C) Never allow any user input.

D) Only validate input on the front-end (browser-side).

**Answer: B) Use regular expressions or allow-lists to only accept valid data.**

**Question 7: How Should Passwords Be Stored Securely?**

A) In plain text in the database.

B) By hashing the passwords before storing them.

C) By storing them in the user's browser.

D) By encrypting passwords using basic methods.

**Answer: B) By hashing the passwords before storing them.**

**Question 8: What is HTTPS?**

A) A way to optimize website performance.

B) A protocol that encrypts data between the client and server for security.

C) A method to increase website loading speed.

D) A type of user authentication system.

**Answer: B) A protocol that encrypts data between the client and server for security.**

**Question 9: What is a Session in Web Security?**

A) A form of encryption for storing passwords.

B) A mechanism that keeps track of a user's actions while they are logged in.

C) A method for improving website performance.

D) A technique used to store data offline.

**Answer: B) A mechanism that keeps track of a user's actions while they are logged in.**

**Question 10: How Can You Foster a Secure Code Culture?**

A) Ignore security during development and fix it later.

B) Conduct regular code reviews and stay updated on security best practices.

C) Only focus on speed, not security.

D) Only focus on testing the code once it’s deployed.

**Answer: B) Conduct regular code reviews and stay updated on security best practices**.

**Level 2: Intermediate**

**Slide 1: Cross-Site Request Forgery (CSRF) Protection (ASP.NET, PHP)**

**What is CSRF?**

An attacker forces a user to unknowingly perform an action in a web application.

**ASP.NET MVC Protection:**

Use AntiForgeryToken in forms and validate with @Html.AntiForgeryToken().

**Slide 2: Securing APIs (ASP.NET, C#)**

**What is an API?**

APIs allow different applications to communicate with each other.

**Securing APIs:**

* Always use HTTPS.
* Use OAuth or API Keys for authentication and authorization.

**C# Example:**

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme) .AddJwtBearer(options => { options.Authority = "https://authserver"; options.Audience = "api1"; });

**Slide 3: Secure File Uploads (PHP, ASP.NET)**

**Risks of File Uploads:**

Malicious files can be uploaded (e.g., viruses, scripts).

**Best Practices:**

Limit file types to trusted formats (e.g., images).

**ASP.NET Example:** Use FileUpload.PostedFile.ContentType to validate file types.

**Slide 4: SQL Injection and ORM (ASP.NET MVC, PHP)**

**ORM for Safer Queries:**

ORM frameworks like Entity Framework (ASP.NET) or Doctrine (PHP) prevent SQL injection.

**Example in Entity Framework:**

csharp

var user = dbContext.Users.Where(u => u.Username == username).FirstOrDefault();

**Slide 5: Secure Authentication and Authorization (ASP.NET, PHP, C#)**

**What’s the Difference?**

* Authentication verifies identity (login).
* Authorization determines access rights (permissions).
* ASP.NET: Use ASP.NET Identity for managing authentication.
* C#: Use role-based authorization ([Authorize(Roles = "Admin")]).

**Slide 6: Secure Database Connections (SQL Server, ASP.NET)**

**Use Parameterized Queries:**

Always use parameterized queries to prevent SQL injection.

**Example in C#:**

**csharp**

SqlCommand command = new SqlCommand("SELECT \* FROM Users WHERE Id = @Id", connection); command.Parameters.AddWithValue("@Id", userId);

**Slide 7: Managing Dependencies and Libraries Securely (ASP.NET, PHP)**

**Risks of Using External Libraries:**

Vulnerabilities in outdated or malicious libraries.

**Best Practices:**

Regularly update libraries.

Use trusted sources and verify their integrity (e.g., use NuGet for .NET packages).

**Slide 8: Protecting Against Cross-Site Scripting (XSS) in JavaScript**

**XSS in JavaScript:**

Attackers inject malicious scripts into dynamic web pages.

**How to Prevent:**

* Sanitize user inputs before rendering them.
* Use libraries like DOMPurify for sanitization.

**Slide 9: Session Management in ASP.NET MVC**

**Session Fixation Attack Prevention:**

* Regenerate session IDs on login to prevent session hijacking.
* Use secure, HttpOnly cookies for session data.

Example: Use Session.Abandon() to clear session data after logout.

**Slide 10: Error Handling and Logging (ASP.NET, C#)**

**Centralized Logging:**

Use a logging framework like NLog or Log4Net for tracking errors securely.

**Example in ASP.NET:**

Configure logging in appsettings.json for secure error tracking.

**Assessment Questions**

### ****1. What is Cross-Site Request Forgery (CSRF)?****

* **A)** An attacker steals user credentials.
* **B)** An attacker forces a user to unknowingly perform an action in a web application.
* **C)** An attacker injects malicious code into a web application.
* **D)** An attacker gains unauthorized access to a database.

**Answer**: **B)** An attacker forces a user to unknowingly perform an action in a web application.

### ****2. How does ASP.NET MVC prevent CSRF attacks?****

* **A)** By using JWT for authentication.
* **B)** By validating the @Html.AntiForgeryToken() in forms.
* **C)** By encrypting all user data.
* **D)** By enforcing HTTPS on all pages.

**Answer**: **B)** By validating the @Html.AntiForgeryToken() in forms.

### ****3. What is the primary function of an API?****

* **A)** To encrypt user passwords.
* **B)** To allow different applications to communicate with each other.
* **C)** To prevent SQL injection attacks.
* **D)** To protect against XSS attacks.

**Answer**: **B)** To allow different applications to communicate with each other.

### ****4. What is the recommended approach for securing APIs in ASP.NET?****

* **A)** Use plain HTTP for all API communications.
* **B)** Always use HTTPS and implement OAuth or API Keys for authentication.
* **C)** Only allow requests from internal servers.
* **D)** Disable logging for security.

**Answer**: **B)** Always use HTTPS and implement OAuth or API Keys for authentication.

### ****5. What is the main risk associated with file uploads?****

* **A)** Malicious users can upload viruses or scripts.
* **B)** File uploads can cause network slowdowns.
* **C)** Users might upload excessively large files.
* **D)** File uploads can make APIs vulnerable to DDoS attacks.

**Answer**: **A)** Malicious users can upload viruses or scripts.

### ****6. How does ASP.NET validate file types during uploads?****

* **A)** By checking the file extension against a whitelist.
* **B)** By using FileUpload.PostedFile.ContentType to check the MIME type.
* **C)** By running virus scans on uploaded files.
* **D)** By limiting file sizes.

**Answer**: **B)** By using FileUpload.PostedFile.ContentType to check the MIME type.

### ****7. What is an ORM and how does it help prevent SQL injection?****

* **A)** Object-Relational Mapping, and it escapes input data to prevent injection.
* **B)** Object-Relational Mapping, and it automatically performs SQL queries without user input.
* **C)** Object-Relational Mapping, and it generates safe SQL queries with parameterization.
* **D)** Object-Relational Mapping, and it encrypts all SQL queries.

**Answer**: **C)** Object-Relational Mapping, and it generates safe SQL queries with parameterization.

### ****8. How do parameterized queries prevent SQL injection in ASP.NET?****

* **A)** They encrypt SQL queries.
* **B)** They directly concatenate user input into the SQL query.
* **C)** They bind user input to SQL parameters, preventing malicious modifications.
* **D)** They store queries in a static location.

**Answer**: **C)** They bind user input to SQL parameters, preventing malicious modifications.

### ****9. What is the difference between authentication and authorization?****

* **A)** Authentication verifies identity; Authorization grants access rights.
* **B)** Authentication and authorization are the same.
* **C)** Authentication is used to encrypt data; Authorization is used for login.
* **D)** Authentication verifies access rights; Authorization manages user roles.

**Answer**: **A)** Authentication verifies identity; Authorization grants access rights.

### ****10. How can session fixation attacks be prevented in ASP.NET?****

* **A)** By regenerating session IDs on login.
* **B)** By storing session data in local storage.
* **C)** By using the same session ID across all requests.
* **D)** By disabling sessions altogether.

**Answer**: **A)** By regenerating session IDs on login.

**Level 3: Advanced (practical)(think of developer perspective)(practical assessment)**

….

**Slide 1: Data Encryption Best Practices**

* **Title**: *Data Encryption for Secure Software Development*
* **Content**:
  + **Encrypt Sensitive Data**:
    - Use **AES-256** (Advanced Encryption Standard with a 256-bit key) to encrypt sensitive data at rest and in transit.
    - For data in transit, always use **TLS (Transport Layer Security)** for HTTPS communication to prevent MITM attacks.
  + **Real-World Example**: In **ASP.NET**, use the RijndaelManaged class for AES-256 encryption to encrypt user data before saving it in the database. Always ensure that your connection strings use **SSL/TLS** encryption when interacting with the database.

**Slide 2: SQL Server Encryption Techniques**

* **Title**: *SQL Server: Transparent Data Encryption (TDE)*
* **Content**:
  + **Transparent Data Encryption (TDE)** in **SQL Server 2019** provides encryption for **entire databases** without requiring changes to the application.
  + TDE helps protect data at rest by encrypting database files, log files, and backups.
  + **How TDE Works**:
    - It automatically encrypts and decrypts data as it is written to or read from disk.
    - Transparent encryption means your application doesn’t need any changes.
  + **Real-World Example**: You have a sensitive customer database, and even if someone physically steals the database backup, TDE ensures the data is unreadable without the correct decryption key.

**Slide 3: Introduction to Secure SDLC**

* **Title**: *What is the Secure Software Development Lifecycle (SDLC)?*
* **Content**:
  + SDLC refers to the stages involved in the development of software from planning to deployment.
  + **Secure SDLC** integrates security at each stage (design, development, testing, and deployment) to mitigate vulnerabilities early.
  + **Real-World Example**: Imagine securing a network of connected systems, ensuring protection at each layer (data encryption, access controls, testing for vulnerabilities).

**Slide 4: Integrating Security in SDLC**

* **Title**: *Embedding Security into the SDLC Process*
* **Content**:
  1. **Design Phase**: Implement secure design principles (e.g., secure coding guidelines, threat modeling).
  2. **Code Phase**: Enforce secure coding practices (e.g., input validation, avoiding dangerous functions like eval()).

**Slide 5: Advanced SQL Injection Prevention (C# & SQL Server)**

* **Title**: *Defending Against Advanced SQL Injection Attacks*
* **Content**:
  + **Use ORMs**: In C#, use **Entity Framework** (EF) or **Dapper** to prevent SQL injection by automatically parameterizing SQL queries.
  + **Least Privilege for Database Users**: Ensure that the database user accounts have the minimum permissions needed to perform their job (e.g., no sysadmin role unless absolutely required).
  + **Best Practices**:
    - Always use **parameterized queries** to avoid direct SQL injection vulnerabilities.
    - Avoid dynamic SQL like EXEC sp\_executesql without parameters.
  + **Real-World Example**:

csharp

// Using Entity Framework to prevent SQL Injection

var user = context.Users.Where(u => u.UserId == userId).FirstOrDefault();

* + - The above **LINQ** query is safe from SQL injection because it automatically handles parameterization behind the scenes.
    - In contrast, writing raw SQL like SELECT \* FROM Users WHERE UserId = @userId without parameters could be dangerous.

### Slide 6: ****Slide 6: Secure API Design with OAuth 2.0 and JWT****

#### ****Content****:

1. **OAuth 2.0**:
   * **OAuth 2.0** is a widely adopted **authorization framework** that allows users to grant third-party applications access to their resources (like profile data, email, etc.) **without sharing their login credentials** (username and password).
   * OAuth 2.0 allows users to authenticate with services such as Google, Facebook, or GitHub and then grant the application access to certain resources like their profile information, calendar, or photos.
   * **OAuth 2.0 Flow**:
     + The client application asks the user to authenticate with the authorization server.
     + The server responds with an **access token**, which the client can use to make API requests to access the user’s data.
2. **JWT (JSON Web Tokens)**:
   * **JWT** is a compact, URL-safe way to represent claims between two parties. It consists of three parts: Header, Payload, and Signature.
     + **Header**: Specifies the algorithm used to sign the token (e.g., HMAC SHA256 or RSA).
     + **Payload**: Contains the claims, such as user ID, roles, permissions, or any other user-specific information.
     + **Signature**: Ensures that the token has not been tampered with. It’s generated by signing the encoded header and payload with a secret key or RSA private key.
   * JWTs are **stateless**, meaning that once issued, they can be used independently for authentication without the need for a server-side session. They are commonly used in **API security** to represent user authentication and authorization.
3. **How OAuth 2.0 and JWT Work Together**:
   * OAuth 2.0 provides a way to obtain an **access token** that can be used to access protected resources.
   * **JWT** is used to securely carry the access token. The server generates a JWT containing important claims (like user ID, roles, etc.) and returns it to the client.
   * The client can then include the JWT in API requests to authenticate and authorize their access.
4. **Real-World Example**:
   * **Scenario**: Imagine you are building a mobile app where users can log in using their Google account.
     + The app sends a request to Google’s OAuth server, asking for access to the user's email address.
     + Google prompts the user to log in and grant the necessary permissions.
     + Once authenticated, Google responds with an **access token**.
     + The app’s backend then generates a **JWT** containing user information (e.g., email, roles) and sends it to the client.
     + On subsequent API requests, the client sends the JWT in the request header. The backend validates the JWT to ensure the user is authorized and can access the requested resources.

### ****Slide 7: OAuth 2.0 Flow & JWT in API Requests****

#### ****Content****:

1. **OAuth 2.0 Authorization Flow**:
   * **Step 1: Authorization Request**:
     + The client (app) redirects the user to the authorization server (e.g., Google, Facebook) with a request for access.
     + The request typically includes a **client ID**, **redirect URI**, and **scopes** (requested permissions).
   * **Step 2: User Authorization**:
     + The user is prompted to authenticate and approve the requested access (e.g., "Allow app to access your email").
   * **Step 3: Authorization Grant**:
     + If the user approves the request, the authorization server sends an **authorization code** to the client.
   * **Step 4: Access Token Exchange**:
     + The client sends the authorization code to the server along with its **client secret**.
     + The server responds with an **access token** (and optionally a refresh token).
   * **Step 5: Using Access Token**:
     + The client can now use the access token to make API requests to the resource server (e.g., fetch the user's profile data).
2. **JWT in API Requests**:
   * **Step 1: Issuing the JWT**:
     + Once the OAuth flow is completed, the authorization server (or backend server) creates a JWT containing the user’s information, roles, or claims and signs it.
     + The token is sent to the client as a response.
   * **Step 2: Storing the JWT**:
     + The client (browser or app) stores the JWT in **local storage** or **cookies** for subsequent API requests.
   * **Step 3: Sending the JWT in Requests**:
     + For each subsequent API request, the client sends the JWT in the **Authorization** header as a **Bearer token**:

Authorization: Bearer <JWT>

* + - This tells the server that the client has already authenticated and is authorized to access the requested resource.
  + **Step 4: Validating the JWT**:
    - When the API receives a request with a JWT, it validates the token:
      * **Signature verification**: The server checks that the JWT has been signed by the expected issuer and that the payload has not been tampered with.
      * **Expiration check**: The server checks that the token has not expired (using the exp claim).
    - If the JWT is valid, the server processes the request; otherwise, it returns an error (e.g., 401 Unauthorized).

#### ****Slide 8: Real-World Example Revisited:****

1. **OAuth Flow Example** (Google login):
   * User logs in via Google and grants permissions.
   * The backend issues a JWT with the user’s email and roles (e.g., admin, user).
2. **API Request with JWT**:
   * The client sends an API request to access user data:

http

GET /user/profile

Authorization: Bearer <JWT>

1. **Server Validates JWT**:
   * The server decodes the JWT and checks the claims to validate the user’s identity and ensure they have the necessary permissions to access the profile.
2. **Secure Resource Access**:
   * If the JWT is valid, the server allows access to the user’s profile data. If not, it returns an error like 401 Unauthorized.

**Slide 9: API Rate Limiting to Prevent Abuse**

* **Title**: *Implementing Rate Limiting in APIs*
* **Content**:
  + **What is Rate Limiting?**: Limiting the number of requests a user can make to an API within a defined time window (e.g., 100 requests per minute).
  + **Why Rate Limiting?**: Protect APIs from abuse (e.g., DoS attacks, brute-force login attempts) and ensure fair use.
  + **How Rate Limiting Works**:
    - You can set rules such as 100 requests per IP address per minute.
    - Use **HTTP headers** like X-RateLimit-Limit and X-RateLimit-Remaining to communicate limits to the client.
  + **Real-World Example**: If you're building a public API, you can use an API gateway (e.g., **Kong**, **AWS API Gateway**) to apply rate limits. For example, **AWS API Gateway** can be set up to allow only 100 requests per minute per user.

**Assessment Questions**:

### ****1. What is the primary benefit of using AES-256 encryption for sensitive data?****

* **A)** It ensures data is encrypted and unreadable to unauthorized users.
* **B)** It speeds up data processing for sensitive data.
* **C)** It stores data in a compressed format.
* **D)** It automatically prevents SQL injection.

**Answer**: **A)** It ensures data is encrypted and unreadable to unauthorized users.

### ****2. What does Transparent Data Encryption (TDE) in SQL Server provide?****

* **A)** It encrypts the entire database, including backup files.
* **B)** It encrypts data only during transmission.
* **C)** It requires application changes for database encryption.
* **D)** It automatically encrypts data in the database but requires manual decryption.

**Answer**: **A)** It encrypts the entire database, including backup files.

### ****3. Which of the following describes Secure SDLC?****

* **A)** It is a model that integrates security at each stage of software development.
* **B)** It is a process to test for SQL injection only.
* **C)** It focuses on encryption of sensitive data only.
* **D)** It only applies security practices during the testing phase.

**Answer**: **A)** It is a model that integrates security at each stage of software development.

### ****4. Which secure coding practice helps prevent SQL injection in C#?****

* **A)** Using dynamic SQL queries.
* **B)** Using ORMs like Entity Framework or Dapper.
* **C)** Using direct SQL queries with user inputs.
* **D)** Using eval() function to parse user input.

**Answer**: **B)** Using ORMs like Entity Framework or Dapper.

### ****5. How does OAuth 2.0 help in securing APIs?****

* **A)** It authenticates users with a username and password directly.
* **B)** It provides a way for third-party applications to access user resources with the user's consent, using access tokens.
* **C)** It prevents SQL injection by validating input.
* **D)** It encrypts all API data during transmission.

**Answer**: **B)** It provides a way for third-party applications to access user resources with the user's consent, using access tokens.

### ****6. What is the role of JWT (JSON Web Tokens) in securing API requests?****

* **A)** JWTs carry authentication and authorization information, ensuring secure access to protected resources.
* **B)** JWTs are used to store user credentials securely.
* **C)** JWTs encrypt the API data during transmission.
* **D)** JWTs perform rate limiting on API requests.

**Answer**: **A)** JWTs carry authentication and authorization information, ensuring secure access to protected resources.

### ****7. How does OAuth 2.0 Authorization Code Flow work?****

* **A)** The client directly exchanges user credentials for an access token.
* **B)** The client redirects the user to the authorization server to grant permissions, and the server sends an authorization code to the client.
* **C)** The client sends a refresh token to the server to get a new access token.
* **D)** The client embeds an access token within the request URL.

**Answer**: **B)** The client redirects the user to the authorization server to grant permissions, and the server sends an authorization code to the client.

### ****8. What is the correct way to send a JWT in an API request?****

* **A)** Send the JWT in the request body.
* **B)** Send the JWT in the URL as a query parameter.
* **C)** Send the JWT in the HTTP Authorization header as a Bearer token.
* **D)** Send the JWT in a custom header called X-Token.

**Answer**: **C)** Send the JWT in the HTTP Authorization header as a Bearer token.

### ****9. What is the purpose of rate limiting in APIs?****

* **A)** To limit the number of authentication attempts from a user.
* **B)** To prevent abuse by restricting the number of requests made to the API within a time window.
* **C)** To validate input data in API requests.
* **D)** To ensure data encryption during API requests.

**Answer**: **B)** To prevent abuse by restricting the number of requests made to the API within a time window.

### ****10. Which of the following is a best practice for implementing rate limiting in APIs?****

* **A)** Allow unlimited requests to avoid performance issues.
* **B)** Use HTTP headers like X-RateLimit-Limit and X-RateLimit-Remaining to communicate limits to the client.
* **C)** Use rate limiting only during high traffic periods.
* **D)** Ignore user IP addresses when applying rate limits.

**Answer**: **B)** Use HTTP headers like X-RateLimit-Limit and X-RateLimit-Remaining to communicate limits to the client.