Module 4 – DATABASE SECURITY AND FILE SECURITY (HTML Examples)

1. Database Security Principles

Database security is essential for protecting data from unauthorized access, manipulation, and threats. Below are key principles of database security:

1.1 Structured Query Language (SQL) Injection

SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries an application makes to its database. It is one of the most common cyber threats and can lead to unauthorized data access, data corruption, or even database destruction.

Vulnerable HTML Login Form

```
<form method="POST" action="login.php">

   <label>Username:</label>

   <input type="text" name="username">

   <label>Password:</label>

   <input type="password" name="password">

   <button type="submit">Login</button>

</form>
```

Secure HTML Form (Preventing SQL Injection)

Use **input validation** and **hidden fields** to add an extra layer of security.

```
<form method="POST" action="secure_login.php">

   <label>Username:</label>

   <input type="text" name="username" required pattern="[A-Za-z0-9]{3,}">

   <label>Password:</label>

   <input type="password" name="password" required minlength="6">

   <input type="hidden" name="csrf_token" value="random-generated-token">

   <button type="submit">Login</button>

</form>
```

□ **Key Enhancements:**

- The pattern attribute restricts inputs to alphanumeric characters.
- The minlength attribute ensures strong passwords.
- A **CSRF token** is added to prevent attacks.

How SQL Injection Works

SQL Injection occurs when an application fails to properly validate and sanitize user inputs, allowing an attacker to manipulate an SQL query. This can be achieved by injecting malicious SQL code into input fields such as login forms, search boxes, or URL parameters.

Example of a Vulnerable SQL Query:

SELECT * FROM users WHERE username = 'admin' AND password = 'password';

If an attacker enters the following input in the username field:

admin' --

The resulting SQL query would become:

SELECT * FROM users WHERE username = 'admin' -- ' AND password = 'password';

Since -- is a comment in SQL, everything after it is ignored, potentially allowing an attacker to bypass authentication.

1.3 Types of SQL Injection

1. **Classic (In-Band) SQL Injection**
   - **Error-Based SQL Injection**: Exploits database error messages to extract data.
   - **Union-Based SQL Injection**: Uses the UNION statement to combine results from different tables.
2. **Blind SQL Injection**
   - **Boolean-Based Blind SQL Injection**: Sends queries that return TRUE or FALSE, allowing attackers to infer information.
   - **Time-Based Blind SQL Injection**: Uses database time delays (SLEEP() in MySQL) to infer data.
3. **Out-of-Band SQL Injection**
   - Uses external resources such as DNS or HTTP requests to exfiltrate data when other techniques are blocked.

1.2 Setting Database Permissions

Proper database permissions ensure that users only have the required access levels.

Proper database permission management ensures that users and applications only have access to the data and actions necessary for their role. This reduces the risk of data leaks, unauthorized modifications, and accidental deletions.

Key Principles of Database Permissions

1. **Principle of Least Privilege (PoLP)**
    - Users should only be granted the minimum level of access required to perform their tasks.
2. **Role-Based Access Control (RBAC)**
    - Instead of assigning permissions to individual users, assign them to roles that users belong to.
3. **Separation of Duties**
    - Different roles should be assigned to different individuals to prevent conflicts of interest (e.g., a user who enters financial data should not approve transactions).
4. **Regular Audits and Reviews**
    - Periodically review database permissions to ensure they align with organizational needs.

5. Common Database Roles

Role	Description
DBA (Database Administrator)	Full access to manage database users, structure, and security.
Read-Only User	Can only perform SELECT queries.
Read-Write User	Can SELECT, INSERT, UPDATE, and DELETE data.
Application User	Has access only to specific tables or procedures required by the application.

Best Practices for Database Permissions

1. **Use Strong Authentication Methods**
    - Enforce multi-factor authentication (MFA) where possible.

2. **Use Encrypted Connections**
   - Use SSL/TLS to secure database connections.
3. **Avoid Using the Root/Superuser Account**
   - Create separate accounts for different database operations.
4. **Monitor and Log Database Activity**
   - Keep logs of database queries and permission changes for security auditing.

Implementing Database Permissions

2.3.1 MySQL Database Permissions Example

To create a new user and grant specific permissions:

```
CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'securepassword';

GRANT SELECT, INSERT ON mydatabase.* TO 'testuser'@'localhost';
```

To revoke a permission:

```
REVOKE INSERT ON mydatabase.* FROM 'testuser'@'localhost';
```

2.3.2 SQL Server Database Permissions Example

To create a user and assign roles:

```
CREATE LOGIN testuser WITH PASSWORD = 'securepassword';

USE mydatabase;

CREATE USER testuser FOR LOGIN testuser;

EXEC sp_addrolemember 'db_datareader', 'testuser';
```

HTML Example: Admin and User Role-Based Access

```
<select name="role">

  <option value="admin">Admin</option>
```

```
    <option value="user">User</option>
</select>
<!-- Restricted Content (Only for Admins) -->
<div id="adminPanel" style="display: none;">
    <h2>Admin Controls</h2>
    <button onclick="deleteUser()">Delete User</button>
</div>

<script>
    let userRole = "user"; // This should come from the server after login
    if (userRole === "admin") {
        document.getElementById("adminPanel").style.display = "block";
    }
</script>
```

 Key Enhancements:

- The admin panel is hidden using JavaScript unless the user is an admin.
- Always validate user roles **on the server-side**.

1.3 Stored Procedure Security

Stored procedures provide an added layer of security for databases by controlling access to data and operations. Here are some examples of stored procedure security measures:

1. Using EXECUTE AS for Privilege Control

A stored procedure can execute under a specific user’s security context, limiting access based on their permissions.

```
CREATE PROCEDURE GetEmployeeSalary
WITH EXECUTE AS 'HR_Manager'
```

```
AS
BEGIN
    SELECT EmployeeID, Salary FROM Employees;
END;
```

Security Benefit: Prevents unauthorized users from directly accessing salary data.

2. Preventing SQL Injection

Use parameterized queries to avoid SQL injection vulnerabilities.

```
CREATE PROCEDURE GetUserByID
@UserID INT
AS
BEGIN
    SELECT * FROM Users WHERE UserID = @UserID;
END;
```

Security Benefit: Prevents malicious SQL injection attempts.

3. Restricting Direct Table Access

Grant users permission to execute the procedure but not access the table directly.

```
GRANT EXECUTE ON GetEmployeeSalary TO SalesTeam;
REVOKE SELECT ON Employees FROM SalesTeam;
```

Security Benefit: Users can only access data via the stored procedure, preventing unintended modifications.

4. Encrypting Stored Procedures

Hide the procedure's definition from unauthorized users.

```
CREATE PROCEDURE GetSensitiveData
WITH ENCRYPTION
AS
BEGIN
    SELECT * FROM ConfidentialTable;
END;
```

Security Benefit: Prevents users from viewing the logic of the stored procedure.

5. Auditing and Logging Access

Track stored procedure executions for security monitoring.

```
CREATE PROCEDURE LogUserActivity
@UserID INT, @Activity VARCHAR(100)
AS
BEGIN
    INSERT INTO AuditLog (UserID, Activity, LogDate)
    VALUES (@UserID, @Activity, GETDATE());
END;
```

Security Benefit: Helps detect unauthorized access or suspicious activity.

6. Using Digital Signatures for Integrity

Sign procedures to ensure they haven't been tampered with.

```
ADD SIGNATURE TO GetUserByID
BY CERTIFICATE SecurityCert;
```

Security Benefit: Ensures only signed, trusted procedures execute.

1. Preventing SQL Injection

Example: Secure Authentication Query

Web applications should use stored procedures with parameterized queries instead of dynamic SQL to prevent SQL injection.

```
CREATE PROCEDURE ValidateUser
    @Username NVARCHAR(50),
    @Password NVARCHAR(255)
AS
BEGIN
    SELECT UserID FROM Users
    WHERE Username = @Username AND PasswordHash = HASHBYTES('SHA2_256',
@Password);
END;
```

Security Benefit:

Prevents attackers from injecting malicious SQL statements via web input fields.

2. Restricting Direct Table Access

Example: Role-Based Data Access

Only allow web applications to interact with the database through stored procedures, not direct queries.

```
GRANT EXECUTE ON ValidateUser TO WebAppUser;
REVOKE SELECT, INSERT, UPDATE, DELETE ON Users FROM WebAppUser;
```

Security Benefit:

The web application can only call the stored procedure, reducing the risk of direct table manipulation.

3. Implementing Application-Level Access Control

Example: Fetching User Data with Role Validation

Stored procedures can enforce role-based access control (RBAC) before returning sensitive data.

```
CREATE PROCEDURE GetUserProfile
    @UserID INT,
    @RequesterID INT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM UserRoles WHERE UserID = @RequesterID AND Role = 'Admin')
    BEGIN
        SELECT * FROM Users WHERE UserID = @UserID;
    END
    ELSE
    BEGIN
        SELECT UserID, Username, Email FROM Users WHERE UserID = @UserID;
    END
END;
```

Security Benefit:

Ensures that users can only see data they are authorized to access.

4. Encrypting Stored Procedures

Example: Protecting Business Logic

Prevent attackers from reading or modifying the stored procedure's logic.

```
CREATE PROCEDURE GetSensitiveData
WITH ENCRYPTION
AS
BEGIN
    SELECT * FROM Transactions;
END;
```

Security Benefit:

Protects against code tampering and data leakage.

5. Logging and Auditing

Example: Logging Stored Procedure Calls for Security Audits

Web applications should log all sensitive operations in a separate audit table.

```
CREATE PROCEDURE LogUserAction
    @UserID INT,
    @Action VARCHAR(255),
    @Timestamp DATETIME
AS
BEGIN
    INSERT INTO AuditLog (UserID, Action, Timestamp)
    VALUES (@UserID, @Action, @Timestamp);
END;
```

Security Benefit:

Tracks potentially suspicious activities in the web application.

6. Using EXECUTE AS for Privilege Control

Example: Running Procedures with Limited Permissions

Stored procedures can execute under a different user context with minimal privileges.

```
CREATE PROCEDURE DeleteUserAccount
WITH EXECUTE AS 'LimitedUser'
AS
BEGIN
    DELETE FROM Users WHERE UserID = @UserID;
END;
```

Security Benefit:

Prevents privilege escalation by ensuring stored procedures run with least privilege.

7. Using Digital Signatures to Prevent Tampering

Example: Ensuring Only Authorized Procedures Run

A stored procedure can be digitally signed to prevent unauthorized modifications.

```
ADD SIGNATURE TO GetUserProfile
BY CERTIFICATE WebAppSecurityCert;
```

Security Benefit:

Ensures only signed procedures execute, preventing malicious alterations.

Stored procedures help secure queries by restricting direct database access.

HTML Example: Secure Search Form Using Stored Procedures

<form method="POST" action="search.php"> **HTML: Secure Search Form**

This form collects user input and sends it to search.php for processing.

html

```
<form method="POST" action="search.php">
  <label>Search User:</label>
  <input type="text" name="search_term" required pattern="[A-Za-z0-9 ]{3,}">
  <button type="submit">Search</button>
</form>
```

□ **Security Features:**

- **required attribute** ensures the field is not left empty.
- **pattern="[A-Za-z0-9]{3,}"** restricts input to letters, numbers, and spaces (prevents SQL injection attempts).

2️⃣ MySQL: Secure Stored Procedure

Create a stored procedure in MySQL to fetch user details **safely**.

```
DELIMITER //
CREATE PROCEDURE GetUser(IN search_term VARCHAR(100))
BEGIN
  SELECT username, email FROM users WHERE username LIKE CONCAT('%', search_term, '%');
END //
DELIMITER ;
```

□ **Security Benefits:**

- Uses **prepared statements** internally.
- Prevents direct query execution.

3️⃣ PHP: Securely Calling Stored Procedure

```
<?php
// Database connection
$conn = new mysqli("localhost", "root", "", "test_db");

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

// Securely retrieve search term
$search_term = isset($_POST['search_term']) ? trim($_POST['search_term']) : '';

if (!empty($search_term)) {
    // Prepare stored procedure call
    $stmt = $conn->prepare("CALL GetUser(?)");
    $stmt->bind_param("s", $search_term);
    $stmt->execute();

    // Fetch results
    $result = $stmt->get_result();
    while ($row = $result->fetch_assoc()) {
        echo "Username: " . htmlspecialchars($row['username']) . " - Email: " .
htmlspecialchars($row['email']) . "<br>";
    }

    $stmt->close();
}
```

```
$conn->close();

?>
```

□ **Security Features:**

- **Uses stored procedure** to avoid raw SQL queries.
- **bind_param("s", $search_term)** prevents SQL injection.
- **htmlspecialchars()** sanitizes output to prevent XSS.

```
    <label>Search User:</label>

    <input type="text" name="search_term" required>

    <button type="submit">Search</button>

</form>
```

□ **Best Practices:**

- Instead of embedding raw SQL queries in search.php, use stored procedures on the backend.

1.4 Insecure Direct Object References (IDOR)

IDOR occurs when users manipulate URLs to access unauthorized data.

Vulnerable Profile Link

```
<a href="profile.php?user_id=1">View Profile</a>
```

An attacker might change ?user_id=1 to ?user_id=2 to access another user's profile.

Secure Profile Access (Using Sessions)

```
<!-- Secure User Profile Access -->

<form method="POST" action="view_profile.php">

    <input type="hidden" name="user_id" value="SESSION_USER_ID">

    <button type="submit">View Profile</button>
```

```
</form>
```

1️⃣login.php – User Login & Start Session

This page handles user login and stores the session.

```
<?php
session_start();
$_SESSION['user_id'] = 1; // Assume user with ID 1 is logged in
$_SESSION['username'] = "Alice"; // Example user
header("Location: dashboard.php");
?>
```

2️⃣dashboard.php – Secure Profile Access

This page shows a **button** to view the profile **without exposing user_id in the URL**.

```
<?php session_start(); ?>
<!DOCTYPE html>
<html>
<head>
   <title>Dashboard</title>
</head>
<body>
   <h2>Welcome, <?php echo $_SESSION['username']; ?>!</h2>
```

```
    <!-- Secure Profile Access Form -->
    <form method="POST" action="view_profile.php">
        <input type="hidden" name="user_id" value="<?php echo $_SESSION['user_id']; ?>">
        <button type="submit">View Profile</button>
    </form>
</body>
</html>
```

3️⃣ view_profile.php – Show User Profile Securely

This page **verifies the session** before displaying the profile.

```
<?php
session_start();

if (!isset($_SESSION['user_id'])) {
    die("Access denied!");
}

echo "<h2>User Profile</h2>";
echo "<p>User ID: " . $_SESSION['user_id'] . "</p>";
echo "<p>Username: " . $_SESSION['username'] . "</p>";
?>
```

✅ Why Is This Secure?

✔ **No user_id in URL**
✔ **Only the logged-in user can view their profile**
✔ **Session-based authentication**

Now, an attacker **cannot** change ?user_id=2 to view someone else's profile!

 Key Enhancements:

- Uses a **hidden field** with a **session user ID** instead of exposing it in the URL.
- The backend should verify the user session.

---

2. File Security Principles

2.1 Keeping Your Source Code Secret

Keeping your source code secret involves implementing measures to ensure that the codebase of an application is not accessible to unauthorized users. This is crucial because source code can contain sensitive information, such as API keys, database credentials, and proprietary algorithms.Examples:

- Access Control: Implement strict access controls using role-based access control (RBAC) to limit who can view or modify the source code. For instance, only developers and certain team leads should have access to the repository.
- Code Repositories: Use private repositories on platforms like GitHub or GitLab, ensuring that only authorized personnel can access the code.
- Environment Variables: Store sensitive information in environment variables instead of hardcoding them into the source code. This way, even if the code is exposed, sensitive data remains protected.

If source code files are exposed, attackers can find vulnerabilities.

Vulnerable File Directory Listing

```
<a href="source_code.php">View Source Code</a>
```

If source_code.php is accessible, an attacker may download sensitive files.

Secure File Protection Using .htaccess

Add the following rule in .htaccess to prevent direct file access:

apache

CopyEdit

```
<FilesMatch "\.(php|config|env)$">
   Order Allow,Deny
   Deny from all
</FilesMatch>
```

□ **Key Enhancements:**

- Blocks access to PHP and configuration files.

2.2 Security Through Obscurity

Security Through Obscurity (STO) is a controversial principle that suggests that keeping the details of a system's design or implementation secret can provide security. The idea is that if attackers do not know how a system works, they are less likely to exploit its vulnerabilities.Criticism: While it may seem logical, relying solely on obscurity is generally considered a flawed approach. Security experts argue that it does not address the underlying vulnerabilities and can create a false sense of security

.Examples:

- Hidden URLs: An application might use non-obvious URLs for sensitive endpoints (e.g., /admin12345 instead of /admin). However, if an attacker discovers the URL, the security is compromised.
- Closed Source Software: Some organizations use closed-source software, believing that keeping the source code hidden will protect them from attacks. However, this approach is criticized because it does not allow for independent security audits.

Obscuring URLs can add minor security benefits, but it should **not** be the sole security measure.

Hidden Admin Login Form

```
<form method="POST" action="admin_secure_987.php">
  <input type="password" name="admin_pass" placeholder="Enter Secure Key">
  <button type="submit">Access Admin Panel</button>
</form>
```

 Key Enhancements:

- Instead of using admin/login.php, the URL is changed to admin_secure_987.php.

⚠ **Limitations:**
Obscurity alone is **not** security! Always enforce authentication.

2.3 Forceful Browsing

Forceful Browsing refers to a technique where an attacker attempts to access restricted areas of a web application by manipulating the URL or parameters. This can expose sensitive information if proper access controls are not in place.

Examples:

Directory Traversal: An attacker might try to access files outside the web root directory by using patterns like ../../etc/passwd in the URL. If the application does not properly validate user input, this could lead to unauthorized access to sensitive files.

Parameter Manipulation: If a web application uses predictable parameters in its URLs (e.g., example.com/user?id=123), an attacker could change the ID to access other users' data (e.g., example.com/user?id=124).

To mitigate the risks associated with forceful browsing, developers should implement robust input validation, proper authentication, and authorization checks to ensure that users can only access resources they are permitted to view.

By understanding and implementing these principles, organizations can significantly enhance their file security and protect sensitive information from unauthorized access and exploitation.

Attackers try to access unauthorized files by modifying URLs.

Vulnerable File Download Link

```
<a href="download.php?file=report.pdf">Download Report</a>
```

An attacker might change ?file=report.pdf to ?file=../../etc/passwd to access system files.

Secure File Access (Allowlisting)

```
<form method="POST" action="secure_download.php">
  <label>Select File:</label>
  <select name="file">
    <option value="report1.pdf">Report 1</option>
    <option value="report2.pdf">Report 2</option>
  </select>
  <button type="submit">Download</button>
</form>
```

□ **Key Enhancements:**

- Uses a dropdown menu instead of a free-text field to prevent unauthorized file access.

Summary of Secure HTML Practices

Threat	**Vulnerable Example**	**Secure Example**
SQL Injection	<input type="text" name="username">	<input type="text" name="username" required pattern="[A-Za-z0-9]{3,}">
Database Permissions	admin panel visible to all	admin panel hidden unless userRole === "admin"
Stored Procedure Security	Direct SQL queries in PHP	Stored procedures for fetching data
IDOR	<a href="profile.php?user_id=1">	<input type="hidden" name="user_id" value="SESSION_USER_ID">
Source Code	<a href="source_code.php">View Source	.htaccess to block PHP/config file

Threat	**Vulnerable Example**	**Secure Example**
Exposure	Code</a>	access
Security Through Obscurity	admin/login.php	admin_secure_987.php
Forceful Browsing	<a href="download.php?file=report.pdf">	Dropdown with pre-selected filenames