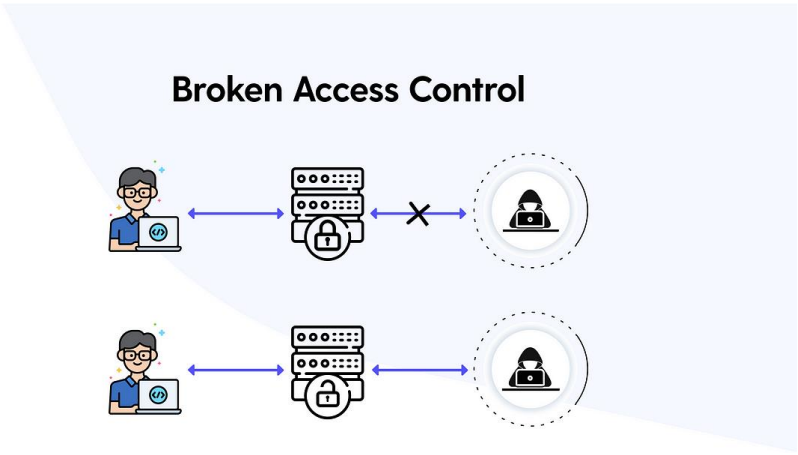
Top 10 OWASP vulnerabilities in websites

And how to fix them

Priyanshu Srivastav
21BAI1233
AI for Cyber Security with IBM Qrador

1. Broken Access Control

Access control systems are intended to ensure that only legitimate users have access to data or functionality. Vulnerabilities in the broken access control category include any issue that allows an attacker to bypass access controls or that fails to implement the principle of least privilege. For example, a web application might allow a user to access another user's account by modifying the provided URL.



How to fix it:

1. Continuous Inspection and Testing Access Control:

Efficient continuous testing and inspecting the access control mechanism is an effective way to detect the newer vulnerabilities and correct them as soon as possible.

2. Deny Access By Default:

Design access control in such a way that not everyone can get access to the resources and functionalities unless it is intended to be publicly accessible. You can apply JIT (Just-in-Time) access, which helps to remove the risks associated with standing privileges.

3. Limiting CORS Usage:

CORS (Cross-Origin Resource Sharing) protocol provides a controlled way to share cross-origin resources. The implementation of the CORS relies on the Hypertext Transfer Protocol (HTTP) headers used in the communication between the client and the target application. When CORS protocol is misconfigured, it makes it possible for a domain to be controlled by a malicious party to send requests to your domain.

4. Enable Role-based Access Control:

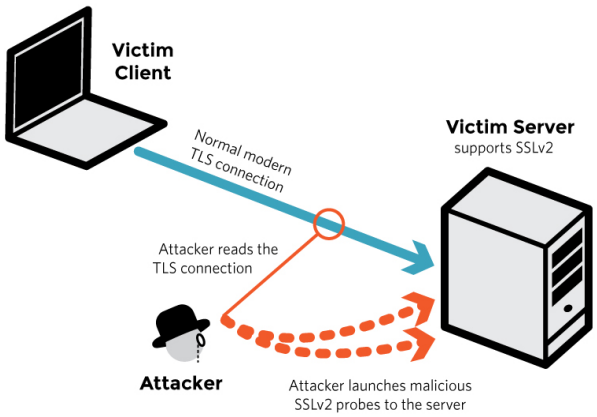
This is a widely used access control mechanism. According to this, users are given permissions based on their roles. Instead of identifying each user individually, users are assigned to a group of roles, this way the struggle of IT support and administration can be reduced, and operational efficiency will be maximized.

5. Enable Permission-Based Access Control:

This is an access control method, where the authorization layer checks if the user has permission to access particular data or to perform a particular action, typically by checking if the user's roles have this permission or not.

2. Cryptographic Failures

Cryptographic algorithms are invaluable for protecting data privacy and security; however, these algorithms can be very sensitive to implementation or configuration errors. Cryptographic failures include a failure to use encryption at all, misconfigurations of cryptographic algorithms, and insecure key management. For example, an organization might use an insecure hash algorithm for password storage, fail to salt passwords, or use the same salt for all stored user passwords.



How to fix it:

1. Choose Strong Algorithms: Select well-established and robust cryptographic algorithms that are widely recognized and have undergone thorough security analysis. Avoid using weak or deprecated algorithms that are known to be vulnerable.

2. Implement Proper Key Management: Ensure secure key generation, storage, and distribution. Use strong random number generators for key generation and employ secure key storage mechanisms, such as hardware security modules (HSMs), to protect keys from unauthorized access.

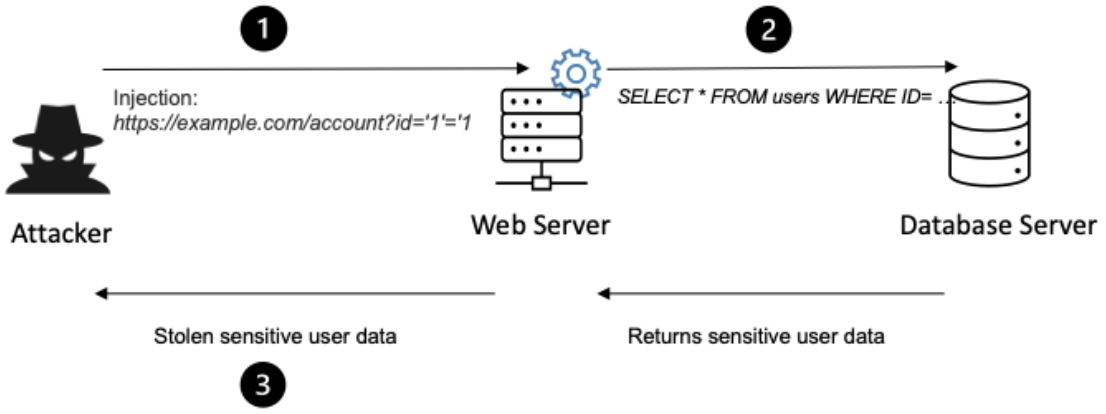
3. Keep Software Up to Date: Regularly update your software, frameworks, and cryptographic libraries to ensure you have the latest security patches. Stay informed about any vulnerabilities or weaknesses discovered in cryptographic implementations and promptly apply updates.

4. Validate and Test Implementations: Thoroughly test and validate your cryptographic implementations to ensure they function correctly and securely. Use established testing methodologies, such as fuzzing or penetration testing, to identify any potential weaknesses or vulnerabilities

5. Follow Best Practices and Standards: Adhere to established cryptographic best practices and standards when designing and implementing secure systems. Rely on widely accepted cryptographic protocols and avoid rolling out custom protocols unless you have the necessary expertise and resources for extensive review and analysis.

3. Injection

Injection vulnerabilities are made possible by a failure to properly sanitize user input before processing it. This can be especially problematic in languages such as SQL where data and commands are intermingled so that maliciously malformed user-provided data may be interpreted as part of a command. For example, SQL commonly uses single (‘) or double (“) quotation marks to delineate user data within a query, so user input containing these characters might be capable of changing the command being processed.



How to fix it:

1. Input Validation and Parameterized Queries: Implement strong input validation by validating and sanitizing user input on the server side. Use parameterized queries or prepared statements in database interactions to separate code logic from user input, preventing malicious input from being executed as commands.

2. Use Whitelisting Approach: Apply a whitelisting approach to input validation, where only known safe values or patterns are allowed, rejecting all other inputs. This helps prevent injection attacks by strictly controlling the types and formats of expected input.

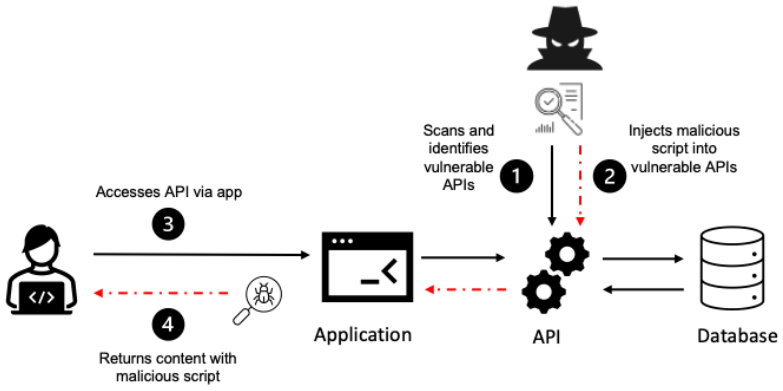
3. Avoid Dynamic Query Building: Avoid constructing SQL queries dynamically by concatenating user input. Instead, use ORM (Object-Relational Mapping) frameworks or query builders that provide built-in protections against injection attacks.

4. Secure Database Configuration: Configure database permissions and access controls properly, ensuring that the application's database user has the least privilege required. Restrict database user permissions to prevent unauthorized access or modification of data.

5. Secure Coding Practices: Train developers on secure coding practices and educate them about the risks of injection vulnerabilities. Encourage the use of secure coding frameworks and libraries that have built-in protections against injection attacks.

4. Insecure Design

Vulnerabilities can be introduced into software during the development process in a couple of different ways. While many of the vulnerabilities on the OWASP Top Ten list deal with implementation errors, this vulnerability describes failures in design that undermine the security of the system. For example, if the design for an application that stores and processes sensitive data does not include an authentication system, then a perfect implementation of the software as designed will still be insecure and fail to properly protect this sensitive data.



How to fix it:

1. Threat Modeling: Perform a thorough threat modeling exercise to identify potential security risks and vulnerabilities early in the design phase. Consider the application's architecture, data flow, trust boundaries, and potential attack vectors. This will help you understand the potential security risks and make informed design decisions.

2. Secure Architecture: Design a secure architecture that incorporates security controls at each layer of the application. Apply security principles such as defense-in-depth and least privilege. Use proven architectural patterns and frameworks that enforce security controls, such as the Model-View-Controller (MVC) pattern.

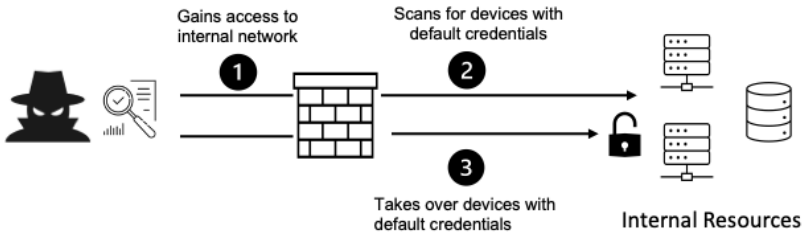
3. Secure Data Management: Ensure secure handling of sensitive data. Implement strong encryption techniques for data in transit and at rest. Use secure protocols such as HTTPS/TLS for data transmission. Employ proper data access controls, authentication mechanisms, and secure storage practices.

4. Access Control and Authorization: Implement robust access control mechanisms to enforce proper authentication and authorization. Apply the principle of least privilege, granting users only the necessary permissions to perform their intended actions. Use role-based access control (RBAC) or attribute-based access control (ABAC) to manage user permissions effectively

5. Secure Input Validation: Implement strict input validation to prevent common vulnerabilities such as injection attacks or cross-site scripting (XSS). Validate and sanitize user input on the server side, following secure coding practices, and avoid relying solely on client-side validation.

5. Security Misconfiguration

In addition to its design and implementation, the security of an application is also determined by how it is configured. A software manufacturer will have default configurations for their applications, and the users may also enable or disable various settings, which can improve or impair the security of the system. Examples of security misconfigurations could include enabling unnecessary applications or ports, leaving default accounts and passwords active and unchanged, or configuring error messages to expose too much information to a user.



How to fix it:

1. Secure Default Configurations: Review and modify default configurations of your web server, application framework, and components. Default configurations often have unnecessary services enabled or weak security settings. Disable or configure them securely to minimize potential vulnerabilities.

2. Patch and Update Regularly: Keep all software, frameworks, libraries, and plugins up to date with the latest security patches and updates. Vulnerabilities in outdated software versions are often exploited by attackers. Establish a process to regularly check for updates and apply them promptly.

3. Secure Server and Network Configurations: Configure your server and network settings securely. Disable unnecessary services, ports, or protocols that are not required for the application's functionality. Implement proper firewall rules, network segmentation, and intrusion detection systems to protect against unauthorized access.

4. Strong Authentication and Access Controls: Implement strong authentication mechanisms, such as multi-factor authentication (MFA), to ensure that only authorized users can access the application. Enforce secure password policies, account lockouts, and session timeouts. Implement proper access controls to restrict privileged operations or sensitive data access to authorized users.

5. Error Handling and Logging: Ensure that error messages do not disclose sensitive information and follow secure coding practices for error handling. Implement detailed logging mechanisms to capture relevant security events and incidents. Regularly review and analyze logs to detect and respond to security incidents promptly.

6. Vulnerable and Outdated Components

Supply chain vulnerabilities have emerged as a major concern in recent years, especially as threat actors have attempted to insert malicious or vulnerable code into commonly used libraries and third-party dependencies. If an organization lacks visibility into the external code that is used within its applications — including nested dependencies — and fails to scan it for dependencies, then it may be vulnerable to exploitation. Also, a failure to promptly apply security updates to these dependencies could leave exploitable vulnerabilities open to attack. For example, an application may import a third-party library that has its own dependencies that could contain known exploitable vulnerabilities.

7. Identification and Authentication Failures

Many applications and systems require some form of identification and authentication, such as a user proving their identity to an application or a server providing a digital certificate verifying its identity to a user when setting up a TLS-encrypted connection. Identification and authentication failures occur when an application relies upon weak authentication processes or fails to properly validate authentication information. For example, an application that lacks multi-factor authentication (MFA) might be

vulnerable to a credential stuffing attack in which an attacker automatically tries username and password combinations from a list of weak, common, default, or compromised credentials.

8. Software and Data Integrity Failures

The Software and Data Integrity Failures vulnerability in the OWASP Top 10 list addresses weaknesses in the security of an organization's DevOps pipeline and software update processes similar to those that made the SolarWinds hack possible. This vulnerability class includes relying on third-party code from untrusted sources or repositories, failing to secure access to the CI/CD pipeline, and not properly validating the integrity of automatically applied updates. For example, if an attacker can replace a trusted module or dependency with a modified or malicious version, then applications that are built with that dependency could run malicious code or be vulnerable to exploitation.

9. Security Logging and Monitoring Failures

Security Logging and Monitoring Failures is the first of the vulnerabilities that are derived from survey responses and has moved up from the tenth spot in the previous iteration of the list. Many security incidents are enabled or exacerbated by the fact that an application fails to log significant security events or that these log files are not properly monitored and handled. For example, an application may not generate log files, may generate security logs that lack critical information, or these log files may only be available locally on a computer, making them only useful for investigation after an incident has been detected. All of these failures degrade an organization's ability to rapidly detect a potential security incident and to respond in real-time.

10. Server-Side Request Forgery

Server-side request forgery (SSRF) is unusual among the vulnerabilities listed in the OWASP Top Ten list because it describes a very specific vulnerability or attack rather than a general category. SSRF vulnerabilities are relatively rare; however, they have a significant impact if they are identified and exploited by an attacker. The Capital One hack is an example of a recent, high-impact security incident that took advantage of an SSRF vulnerability.

SSRF vulnerabilities can exist when a web application does not properly validate a URL provided by a user when fetching a remote resource located at that URL. If this is the case, then an attacker exploiting the vulnerability can use the vulnerable web application to send a request crafted by the attacker to the indicated URL. This allows the attacker to bypass access controls, such as a firewall, which would block direct connections from the attacker to the target URL but is configured to provide access to the vulnerable web application.