# **OWASP Top Ten**

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop and maintain applications that can be trusted.

The goal of the Top 10 project is to raise awareness about application security by identifying some of the most critical vulnerabilities facing organizations.

In this training module each vulnerability will be covered with an introduction, an example and recommendations on the prevention or mitigation of that vulnerability.



## **A1 – Broken Access Control**

The first vulnerability is ***Broken Access Control***.

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others.

These checks are performed after authentication and govern what 'authorized' users are allowed to do. Access control sounds like a simple problem but it is insidiously difficult to implement correctly.

A web application’s access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into several groups or roles with different abilities or privileges.

* **Exploitability** – Occurs when the attacker changes the parameter value, which directly refers to a system object for which they are authorized.
* **Security Weakness** –
  + The occurrence is common in applications and APIs where all user request privileges are not verified.
  + Easy to detect with manual testing, but not open to automatic dynamic or static testing.
* **Technical Impacts** – It is common to come across applications, which go to the trouble of applying robust security mechanism for authentication management. The main factor, which squanders that investment is overlooking to figure effective access control strategies on them. The main reason that these vulnerabilities are so widespread is that verifying access control should be performed for each request and operations on a resource, which a user tries to perform at a certain time. Conceptually, this vulnerability is simple – The application allows the user to do something they should not be permitted to.

**How to defend Broken Access Control Vulnerability?**

* Explicitly verify the access control needs for every chunk of application operation and document it. This requires comprising who can legitimately allow performing an action and what resources the user may access through the action.
* Drive entire access control decisions from the lower privileged user’s session.
* Employ a central application component for verifying access control.
* Verify every single request with this central applications component to decide whether the request from the user is permitted to access the resources.

|  |  |
| --- | --- |
| * Employ programmatic techniques to guarantee that there is no case of exceptions. * For more sensitive functionalities like accessing administrative pages, add additional access restriction with IP address to enforce only users from certain network are permitted to access the resources, irrespective of their login status. * Log each event where sensitive operation is performed that will help to detect and investigate if any access control breaches happen. | * Ensure prevention from forces browsing by providing access rights only to the users equal with their privileges. * Always test unprivileged roles or low-level access based on the information under separation of duties. You can capture and replay the privileged request to test the same. * Deny all by default that will treat everything not explicitly allowed is banned. * Review the server/application from time to time to detect the holes in the access controls. |

**Example Attack Scenarios**

Here are some example scenarios of an attack that can happen if improper or no verification is performed against the user.

* **Scenario 1** – The application uses unverified data in SQL call that is accessing account information:

An attacker simply modifies the browser’s 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user’s account.

* **Scenario 2** – An attacker simply forces browsers to target URLs. Admin rights are required for access to the admin page.

If an unauthenticated user can access either page, it is a flaw. If a non-admin can access the admin page, that is a flaw.

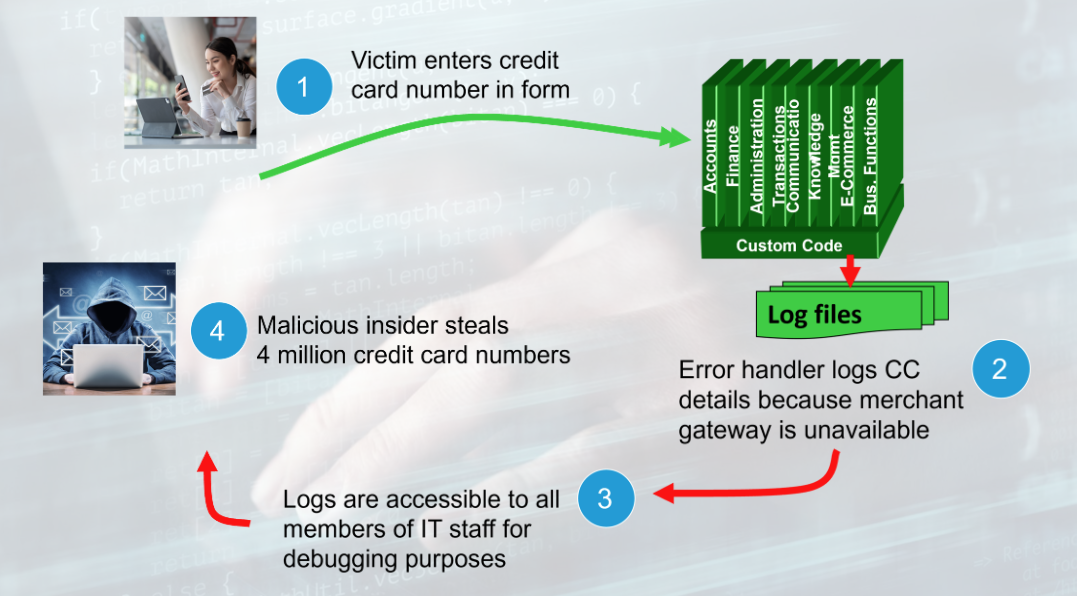
## **A2 – Cryptographic Failures**

Previously know as ***Sensitive Data Exposure***, which is more of a broad symptom rather than a root cause, the focus is on failures related to cryptography (or lack thereof). Which often lead to exposure of sensitive data.

Many web applications for not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identify theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

* **Storing and transmitting sensitive data insecurely**
  + Failure to identify all sensitive data.
  + Failure to identify all the places that this sensitive data gets stored (Databases, files, directories, log files, backups etc.)
  + Failure to identify all the places that this sensitive data is sent. (On the web, to backend databases, to business partners, internal communications.)
  + Failure to properly protect this data in every location.
* **Typical Impact**
  + Attackers access or modify confidential or private information, e.g., credit cards, health care records, financial data (yours or your customers)
  + Attackers extract secrets to use in additional attacks.
  + Company embarrassment, customer dissatisfaction, and loss of trust.
  + Expense of cleaning up the incident, such as forensics, sending apology letters, reissuing thousands of credit cards, providing identify theft insurance.
  + Business gets sued and/or fined.

**Example of Insecure Cryptographic Storage**



**Avoiding Cryptographic Failures**

To avoid insecure cryptographic storage, do the following:

* **Verify your architecture** -
  + Identify all sensitive data.
  + Identify all the places that data is stored.
  + Ensure threat model accounts for possible attacks.
  + Use encryption to counter the threats, do not just 'encrypt' the data.
* **Protect the architecture with appropriate mechanisms** -
  + File encryption, database encryption, data element encryption
* **Use the mechanisms correctly** –
  + Use standard strong algorithms.
  + Generate, distribute, and protect key properly.
  + Be prepared for key change.
  + Do not store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
* **Make sure to encrypt all sensitive data at rest.**
* **Encrypt all data in transit.**
* **Keys should be generated cryptographically randomly and stored in memory as byte arrays. If a password is used, then it must be converted to a key via an appropriate password base key derivation function.**
* **Verify the implementation –** 
  + A standard strong algorithm is used, and it is the proper algorithm for this situation.
  + All keys, certificates and passwords are properly stored and protected.
  + Safe key distribution and an effective plan for key change are in place.
  + Analyze encryption code for common flaws.

## **A3 – Injection**

The next vulnerability is Injection flaws.

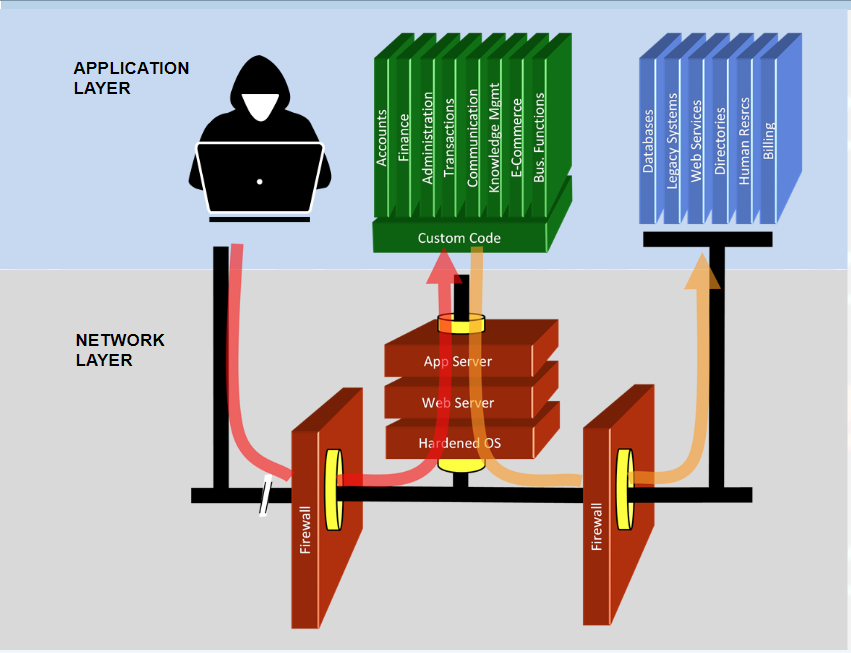
Injection flaws such as SQL, OS and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query.

The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

* **Injection means** – tricking an application into including unintended commands in the data sent to an interpreter.
* **Interpreters** –
  + Take strings and interpret them as commands.
  + SQL, OS shell, LDAP, XPath, Hibernate, etc.
* **SQL injection is still quite common**
  + Many applications still susceptible
  + Even though it is usually very simple to avoid (source code review is the best method of detecting if applications are vulnerable to injections).
* **Typical Impact**
  + Usually, severe. Entire database can be read or modified.
  + May also allow full database schema, or account access, or even OS level access.

**SQL Injection Illustrated**

The images below illustrate SQL injection.

1. An application presents a form to the attacker.
2. The attacker sends an attack in the form data.
3. Application forwards attack to the database in a SQL query.
4. Database runs query containing attack and sends encrypted results back to application.
5. Application decrypts data as normal and sends results to the user.

**Recommendations for avoiding Injection flaws.**

To read the [SQL injection prevention cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html) under references.

Let us look at some recommendations for avoiding injection flaws –

* Avoid the interpreter entirely.
* Use an interface that supports bind variables (e.g., prepared statements, or stored procedures).
* Bind variables allow the interpreter to distinguish between code and data.
* Encode all user input before passing it to the interpreter.
* Always perform positive server-side input validation on all user supplied input.
* Always minimize database privileges to reduce the impact of a flaw.
* Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Injection flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. Injection allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface websites, or redirect the user to malicious sites.

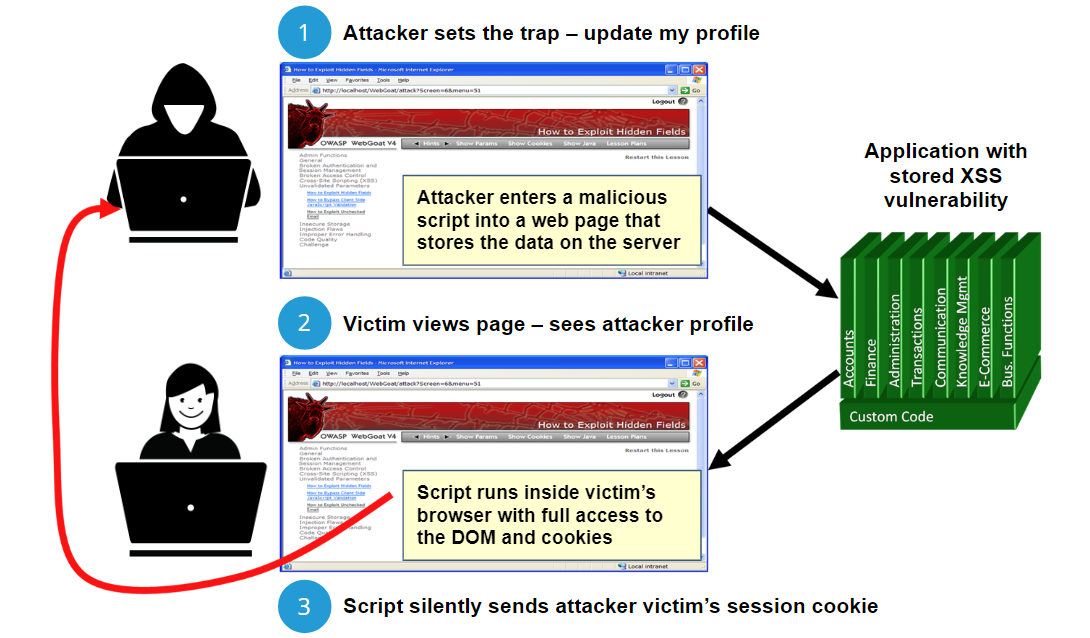
* Injection can occur any time – Raw data from an attacker is sent to an innocent user’s browser.
* Injection can impact raw data –
  + Stored in the database,
  + Reflected from web input (form field, hidden field, URL, etc.),
  + Sent directly into rich JavaScript client.
* Typical Impact of Injection –
  + Steal user’s session, steal sensitive data, rewrite web page, redirect user to phishing or malware site.
  + Most severe: Install XSS proxy which allows an attacker to observe and direct all user’s behavior on vulnerable site and force user to other sites.

**Cross Site Scripting Illustration**

Assume we have an application with a stored cross site scripting vulnerability.

The attacker enters a malicious script into a web page that stores the data to the server.

1. The attacker sets a trap – for example in update my profile.
2. The victim views the page.
3. The script runs inside the victim’s browser with full access to the DOM and cookies.



**Recommendations**

SQL injection flaws are introduced when software developers create dynamic database queries constructed with string concatenation which includes user supplied input.

To avoid SQL injection flaws is simple. Developers need to either:

1. Stop writing dynamic queries with string concatenation; and/or
2. Prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

**Eliminate Flaw** –

* Do not include user supplied input in the output page.

**Defend against the flaw** –

* Use Content Security Policy ([CSP](https://owasp.org/www-community/controls/Content_Security_Policy))
* Primary recommendation: Output encode all user supplied input (Use OWASP’s [ESAPI](https://owasp.org/www-project-enterprise-security-api/) or [Java Encoder](https://owasp.org/www-project-java-encoder/) to output encode).
* Perform “allow list” input validation on all user input to be included in page.
* For large chunks of user scripted HTML, use OWASP’s [AntiSamy](https://owasp.org/www-project-antisamy/) to sanitize this HTML to make it safe.

**XSS Prevention Rules Summary**

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Context | Code Sample | Defense |
| String | HTML Body | <span>UNTRUSTED DATA</span> | * [HTML Entity Encoding](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary) |
| String | Safe HTML Attributes | <input type="text" name="fname" value="UNTRUSTED DATA"> | * [Aggressive HTML Entity Encoding](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary) * Only place untrusted data into a allow list of safe attributes (listed below) * Strictly validate unsafe attributes such as background id and name. |
| String | GET Parameter | <a href="/site/search?value=UNTRUSTED DATA"> clickme</a> | * [URL Encoding](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary) |
| String | Untrusted URL in a SRC or HREF attribute | <a href="UNTRUSTED DATA"> clickme</a>  <iframe src="UNTRUSTED URL" /> | * Canonicalize input * URL Validation * Safe URL verification * Allow list http and https URL’s only (Avoid the JavaScript Protocol to Open a new Window) * Attribute encoder |
| String | CSS Value | <div style="width: UNTRUSTED DATA;">Selection</div> | * [String structural validation](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules-summary) * CSS Hex encoding * Good design of CSS Features |
| String | JavaScript Variable | <script>var currentValue='UNTRUSTED DATA';</script>  <script>someFunction('UNTRUSTED DATA');</script> | * Ensure JavaScript variables are quoted. * JavaScript Hex Encoding. * JavaScript Unicode Encoding. * Avoid backlash encoding (\" or \' or \\) |
| HTML | HTML Body | <div>UNTRUSTED HTML</div> | * [HTML Validation](https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html) |
| String | DOM XSS | <script>document.write("UNTRUSTED INPUT: " + document.location.hash);<script/> | * [DOM based XSS Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html) |

## **A4 – Insecure Design**

Application functions related to authentication and sessions management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities.

Insecure Design focuses on risks related to design and architectural flaws, with a call for more use of threat modeling, secure design patterns, and reference architectures. As a community we need to move beyond "shift-left" in the coding space to pre-code activities that are critical for the principles of Secure by Design.

One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

**Insecure Design**

* Broad category representing different weaknesses, expressed as "missing or ineffective control design".
* There is a difference between insecure design and insecure implementation. They have different causes and remediation.

**Secure Design**

* Secure design is a culture and methodology that constantly evaluates threats and ensures that code is robustly designed and tested to prevent known attack methods.
* Threat modeling should be integrated into refinement sessions (or similar activities).
* Look for changes in data flows and access control or other security controls.
* Fail Securely: Analyze assumptions and conditions for expected and failure flows, ensure they are still accurate and desirable.
* Document results in the user story. Learn from mistakes, secure design is neither an add-on nor a tool that you can add to software.

As a roadmap to secure design, reference the [Security by Design Principles According to OWASP](https://patchstack.com/articles/security-design-principles-owasp/).

**Prevention**

Secure software requires a secure development lifecycle, some form of secure design pattern, paved road methodology, secured component library, tooling, and threat modeling.

|  |  |
| --- | --- |
| * Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls. * Establish and use a library of secure design patterns or paved road ready to use components. * Use threat modeling for critical authentication, access control, business logic and key flows. * Integrate security language and controls into user stories. * Integrate plausibility checks at each tier of your application (from frontend to backend). | * Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases and misuse-cases for each tier of your application. * Segregate tier layers on the system and network layers depending on the exposure and protection needs. * Segregate tenants robust by design throughout all tiers. * Limit resource consumption by user or service. |

## **A5 – Security Misconfiguration**

**XML External Entity (XXE)** refers to a specific type of **Server-side Request Forgery (SSRF)** attack, whereby an attacker can cause Denial of Service (DoS) and access local or remote files and services, by abusing a widely available, rarely used feature in XML parsers.

An **XML External Entity attack** is a type of attack against an application that parses XML input. The attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser.

**What is XXE?**

XML External Entity (XXE) refers to a specific type of [Server-Side Request Forgery (SSRF)](https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/) attack, whereby an attacker is able to cause Denial of Service (DoS) and access local or remote files and services, by abusing a widely available, rarely used feature in XML parsers.

**What is a Server-Side Request Forgery (SSRF) Attack?**

* Server-Side Request Forgery (SSRF) refers to an attack where in an attacker is able to send a crafted request from a vulnerable web application. SSRF is usually used to target internal systems behind firewalls that are normally inaccessible to an attacker from the external network. Additionally, it is also possible for an attacker to leverage SSRF to access services from the same server that is listening on the loopback interface (127.0.0.1).
* Typically, Server-Side Request Forgery (SSRF) occurs when a web application is making a request, where an attacker has full or partial control of the request that is being sent. A common example is when an attacker can control all or part of the URL to which the web application makes a request to some third-party service.

XML Entities however can be used for much more than denial of Service since XML entities do not necessarily have to be defined in the XML document. In fact