Penetration Testing Automation System

Author: Oriol Caño Bellatriu

Supervisor: Manuel Garcia-Cervignon Gutierrez

A thesis submitted in fulfilment of the requirements for the Degree in Informatics Engineering

in the

Barcelona School of Informatics

June 2014
Acknowledgements

I would like to thank my supervisor, Manuel Garcia-Cervignon, for the idea of the project and all the help provided. Also, I would like to thank Marc Sampé, for his continuous interest and help, and Toni Colom, for a key contribution he had that improved significantly the project.
# Contents

## Acknowledgements

1

## Contents

2

## List of Figures

5

## Abbreviations

6

## 1 Introduction

1

1.1 Context .............................................. 1
1.2 Planning ............................................. 2
1.3 Documentation Structure .......................... 3

## 2 Related Work

4

2.1 Golismero ........................................... 4
2.2 OWSAP OWTF ......................................... 6

## 3 Penetration Testing Guide

7

3.1 Information Gathering ................................ 7
  3.1.1 External Footprinting ............................. 8
    3.1.1.1 Active Footprinting ......................... 8
    3.1.1.2 Passive Footprinting ....................... 11
  3.1.2 Web Application Fingerprinting ................... 13
    3.1.2.1 Surface Discovery ......................... 13
    3.1.2.2 Application Entry Points ................... 15
    3.1.2.3 Robots.txt ................................ 15
    3.1.2.4 Analysis of error codes .................... 16

3.2 Configuration Management Testing ................. 16
  3.2.1 SSL/TLS Testing ................................ 16
  3.2.2 DB Listener Testing ............................ 17
  3.2.3 Infrastructure Configuration Management Testing ... 18
  3.2.4 Application Configuration Management Testing .... 18
  3.2.5 Testing for File Extensions Handling ........... 19
  3.2.6 Old, Backup and Unreferenced Files ............. 19
  3.2.7 Infrastructure and Application Admin Interfaces ... 20
  3.2.8 Testing for HTTP Methods and XST ............. 20

3.3 Authentication Testing ............................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Technical Problems</td>
<td>91</td>
</tr>
<tr>
<td>6.2 Economical Analysis</td>
<td>92</td>
</tr>
<tr>
<td>6.3 Future Work</td>
<td>93</td>
</tr>
<tr>
<td>6.4 Conclusions</td>
<td>94</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Command line interface</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Vulnerability Report</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>PDF reporting</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>OWTF - Reporting system</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>Audit</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Machine with Port</td>
<td>54</td>
</tr>
<tr>
<td>4.3</td>
<td>Vulnerabilities</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>Employee</td>
<td>55</td>
</tr>
<tr>
<td>4.5</td>
<td>User and Network</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>Phase and Action</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Complete Data Model</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Register form</td>
<td>76</td>
</tr>
<tr>
<td>4.9</td>
<td>Login page</td>
<td>77</td>
</tr>
<tr>
<td>4.10</td>
<td>Network List</td>
<td>78</td>
</tr>
<tr>
<td>4.11</td>
<td>Audit detail before launching</td>
<td>79</td>
</tr>
<tr>
<td>4.12</td>
<td>Audit detail after launching</td>
<td>80</td>
</tr>
<tr>
<td>4.13</td>
<td>Employee list</td>
<td>80</td>
</tr>
<tr>
<td>4.14</td>
<td>Machine List</td>
<td>83</td>
</tr>
<tr>
<td>4.15</td>
<td>Port detail</td>
<td>84</td>
</tr>
<tr>
<td>4.16</td>
<td>Altair vulnerabilities</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>Employees - Metasploitable</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Machines - Metasploitable</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Vulnerabilities - Metasploitable</td>
<td>88</td>
</tr>
<tr>
<td>5.4</td>
<td>Employees - Renfe</td>
<td>89</td>
</tr>
<tr>
<td>5.5</td>
<td>Machines - Renfe</td>
<td>90</td>
</tr>
</tbody>
</table>
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE</td>
<td>Common Vulnerabilities and Exposures</td>
</tr>
<tr>
<td>CWE</td>
<td>Common Weakness Enumeration</td>
</tr>
<tr>
<td>OSVDB</td>
<td>Open Source Vulnerability DataBase</td>
</tr>
<tr>
<td>CPE</td>
<td>Common Platform Enumeration</td>
</tr>
<tr>
<td>XSS</td>
<td>Cross Site Scripting</td>
</tr>
<tr>
<td>CSRF</td>
<td>Cross Site Request Forgery</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This is the first chapter of this project. Here an introduction to it will be provided. This introduction will cover basic aspects such as what will this project be about, how the project will be planned and developed, and how is this document structured.

1.1 Context

One of the main task of the security analyst job is to perform penetration tests in order to check whether the system under study is secure or not. In order to check it, some test need to be performed using several tools. The objective of these tests is to find vulnerabilities on the target system that can be exploited in order to gain control of it. There are lots of tools that can help a security expert performing his/her job. This tools help by automating some actions that are necessary in order to obtain information about or to exploit the target system. However, these tools are not enough to perform a good penetration test. High creativity is required, as well as high knowledge on the subject in order to succeed. But, the first steps of a penetration test require the security expert to use these tools. In these first steps, the tools that people use are always the same, and they have a natural order to be used in. This fact makes executing these tools a little tiring, since it’s the least creative part of the process. Some tools need to be used first, in order to gather the results and format them to be used as input for the next layer of tools. This could take a lot of time from the tester, who could be doing other more creative stuff.
Doing a penetration test should be more about focusing on the creative parts of it instead of having to do the first boring and repetitive steps, that is why the system that has to be developed in this project aims at fixing this.

By automating most of the tools that have to be launched in the penetration testing process, a security expert could focus solely on thinking out of the box and being creative about the system s/he is studying. All the tools that need to be launched, and all the interaction between them should be automatic. This will be the aim of this project, building a system that automates most of the phases that a penetration test needs to go through.

In order to develop this system, lots of research are needed to understand which steps can be automated and which tools are used in each step. Then, with the penetration testing process in mind, a system that gathers the tools, executes them, and integrates their results has to be designed and developed. In order for this system to be useful for security experts, the results should be clear and easy to understand.

1.2 Planning

This project is part of the Degree’s Final Project. This means that the project has 750 hours planned to be completed. These 750 hours are structured in 6 months, the duration of a semester. This planning establishes nearly a full-time job of roughly 6 hours a day, 5 days a week, during the planned 6 months.

The planning of these 6 months is the following:

- Research - 1 month
- Analysis and design - 1 month
- Development - 2 months
- Experimentation and bug fixing - 1 month
- Documentation - 1 month

Since there is no prior knowledge of the penetration testing process, the research and analysis phases will last as long as the development phase. During the state of the art
research some documentation will also be done in order to make the last month lighter. At first sight, this planning should be enough to complete this project with a working system.

1.3 Documentation Structure

This documentation will be structured in 6 different chapters. First of all, this introduction, following the introduction, some state of the art solutions to the same problem will be analysed, after that, the process of a penetration test will be defined and detailed, as well as some of the tools used in it, in the form of a guide. After the guide, the system will be discussed, starting by its analysis and design, and ending with its implementation and deployment. After the system has been described, some test cases will be reproduced in order to demonstrate how the system works, and finally some conclusions will be make, together with some economical analysis and future work.
Chapter 2

Related Work

In this chapter two existing solutions similar to the project under development will be studied. This two solutions are both security frameworks aimed at scanning web servers in an automatic manner. This two frameworks are Golismero and OWASP OWTF.

2.1 Golismero

The first framework that will be studied is Golismero, a security framework built by a team of three Spanish developers/security experts. Golismero is a free software framework for security testing. It’s geared towards web security and it incorporates a lot of well-known security tools like OpenVAS, Wfuzz, SQLMap, DNSrecon, and many more. It uses the tools to test a web site, and then reads the results of the tools to obtain results and feedback the rest of the tools. Once it has finished, it merges the results and displays them.

It has integration with known standards for vulnerabilities, like CVE and CWE and it allows to develop plugins to incorporate new tools to the system using a cloud-based IDE.

The framework is platform independent and has no native library dependencies. This is accomplished by writing it in pure Python.

This framework is aimed towards the first stages of the penetration testing process. It does not try to exploit the vulnerabilities it founds. Instead, it generates reports
containing all the information it has found, in multiple formats. This formats include a static HTML page and pdf creation.

The framework is used through the command line and the authors claim it has a better performance than the rest of current security frameworks.
2.2 OWSAP OWTF

The *Offensive (Web) Testing Framework* is, as its name says, a testing framework that focus on uniting great security tools together to make pen testing more efficient.

This framework has been developed by Abraham Aranguren, a member of OWASP. The framework is OWASP Testing Guide-oriented, meaning that it classifies its findings as closely as possible to the testing guide provided by OWASP. It also reports its findings on the fly, meaning that as soon as a tool finishes, the results are updated on the database and can be explored.

It is written in Python and works in any Linux systems. However, it needs the tools to be installed in order to run them. It does not crash if a tool is not in the system, though. It is easy to control and easy to run, using a command line tool for it, and it provides some examples about its usage with several examples. It stores the information obtained on a database and extracts data from it in a parsable format as well.

However, the reporting system, an HTML page that is updated when a tool finishes, is a little bit chaotic and an upgrade project for this part is currently taking place.

This system, thanks to having OWASP behind it, has a great community of developers that are constantly upgrading it. This happens through the participation on the *Google Summer of Code* program, which invites students to work on a project baked by a mentor from a technology company. With this program, the companies providing the mentors also propose projects to upgrade their own systems. Besides from this output format upgrade, one of the most interesting *GSoC* projects that OWTF has is the application of Multiprocessing on the system.

![Figure 2.4: OWTF - Reporting system](image-url)
3.1 Information Gathering

The first stage in a penetration test is to gather as much information as possible about the subject of the test. This step is crucial to understand how the system we have to analyse is. In order to comprehend the target system we will use several tools and techniques to discover different things about it.

Knowing what is present on the target is one of the most important things of the test, since it defines the possible attack vectors. Discovering the target’s network structure, the operative systems and the applications each machine uses, and information about the users that use the system is vital to plan a successful attack. Once all this information is known, attacks can be planned. This attacks can pinpoint known vulnerabilities of the software used by the machines, and the information gathered about the users can help completing the attack with some social engineering.

This section will be divided on two main parts. The first part will be based on fingerprinting the target. Using a set of techniques and tools to discover all the information mentioned before. The second part will be directed to the fingerprinting of a web application itself. The web application is usually the visible part of the target and has to be tested as well.
3.1.1 External Footprinting

3.1.1.1 Active Footprinting

DNS Discovery

The Domain Name System is a hierarchical naming system for computers, services or any resource connected to the Internet or a private DNS network. This service helps users ease their lives by encoding IP directions to meaningful character strings. This protocol uses a lot of information and it stores this information in registers. This information can be consulted and will help perform a first mapping of the network infrastructure of the target.

Different kinds of information can be obtained from DNS servers. The main piece of information obtainable from these servers is the infrastructure of machines indexed by them. This servers provide the naming system to refer to all the machines inside the target’s network, and this information can be gathered from the outside. Also, this servers can provide collateral information about the browsing history of the users inside the network. This would not seem like really useful information, but it helps performing a social engineering attack.

There are several tools to discover the machines inside a domain. One of this tools is Dig. This command-line tool directly asks a DNS server about the information in its registers. This tool allows to discover the address of the target, the mail servers associated to the target, the list of DNS servers linked to the target, and even other domains assigned to these DNS servers.

Another way to get all the name servers associated to a domain is using a Metasploit module called dns_enum. This module lets the tester send multiple queries and is even able to perform a brute-force name discovery from a dictionary. After configuring it and launching it, this module presents the tester with the list of all the name servers it has found.

Another tool that lets the tester obtain information from the name server is Nmap with the option -sL. This option allows to perform a reverse DNS lookup for an IP range. This scan will find, besides the name servers, all the domain names that these servers are providing, together with their respective addresses.
Chapter 3. *Penetration Testing Guide*

As mentioned before, an interesting feature of DNS servers provides testers with the ability to discover which domains have been visited from inside of the target’s organisation. This feature is the cache of the DNS, and the technique to get this information is called **DNS Cache Snooping**. This technique gives an attacker the necessary background to perform a *phishing* attack. When a domain is visited for the first time, the name server stores the IP address of that domain in the cache. There are two ways of getting this information from the DNS cache.

The first one is through *non-recursive queries*, which consists of realising non-recursive DNS queries. With these queries, an attacker can ask whether a domain is stored in the DNS server’s cache or not. If it is on the cache, the query will return the IP address of the queried domain. This method does not pollute the cache, but it is not always available since sometimes recursive queries are forced.

The other method is through *recursive queries* This method does not make a difference between cached and non-cached domains, but it is still possible to deduce if the domain asked was fetched from the cache or not. To deduce it, the tester has to take into account the TTL time. Two different times will be noticed, the time that takes for a cached answer, and the time that takes to ask for the domain translation to another authoritative name server. The problem with this method is that the cache gets polluted afterwards.

There are tools that automate this process. Some of them are command line scripts like *cache_snoop.pl*, or *dns-cache-snoop*. The tool *Foca* also has a *DNS Cache Snooping* module that does this process automatically, using a file with the domains that have to be checked.

**Banner Grabbing**

Banner grabbing is the technique that lets the tester extract information from banners offered by the different services. This information is mainly the type and version of the used software. There are several tools to get this information. Some of these tools are *Netcat* and *Ncat*, the latter being an evolution of the first one. This tools connect to an IP range and return the banners of the services that provide them. *Ncat* is able to work over SSL, and can obtain banners from HTTPS ports.

The banners from an HTTP port can sometimes give away valuable information, like for
example the methods allowed by the server. If the OPTIONS method is allowed, *Ncat* will return all the methods allowed by the server.

**Maltego**

*Maltego* is a proprietary software tool used to gather information from public sources and display it in a graphic way. This tool provides a set of transforms that can be done over entities, being them both infrastructures and people.

Even though the work that Maltego can perform is doable with other tools, the graphic way it presents its findings helps understand the relations that the entities under study have.

As an example, if a tester starts with a domain name and s/he wishes to discover as much information as possible, s/he can obtain -through transforms- information such as e-mail addresses related to the domain, PGP server accounts, domain information, etc. All this information is then presented in graph format, and the information obtained can be also expanded. For example, the obtained e-mails can be used to profile the users behind those e-mails using services like *Twitter* or *Facebook*, and the domain information can be used to find all the other domains served by the same DNS server, or even to find known vulnerabilities in the web server responding to that domain.

**SMTP**

*SMTP* is the standard protocol for receiving and sending mails over the Internet. While performing an attack or a pentest, the main interest usually is on obtaining e-mail addresses with the objective of performing some social engineering attack. However, besides this obvious usage, the mailing protocol can help obtain even more information about the target.

One of the ways it can be used is through the bounce back feature. In certain circumstances, mailing systems send automatic messages offering interesting information to the receiver of these e-mails. For example, when an e-mail is sent and the recipient’s inbox is full, the server sends back a message indicating that the message has not been delivered. From this kind of messages useful information can be extracted, such as machine names inside the network, IP addresses, or even firewall system’s signatures.

Another interesting way in which this protocol can be used is through the usage of the VRFY (abbreviation for verify) parameter. This parameter is often used by spammers
to identify usernames inside a web server. Using it, an attacker can ask whether a user-
name is present on the server or not, and if it is, the server will respond with the name and the mailbox of that user. There exist tools that using a dictionary can perform a brute force attack with this technique. Examples of such tools are Medusa or Metasploit. A list of possible usernames has to be provided in order to perform this kind of attack.

**VoIP**

Voice over IP makes it possible to send voice messages through the Internet using the IP protocol. Due to it being inexpensive, lots of companies are nowadays implementing this systems for internal communication. There are several protocols that implement this technology, such as SIP, IAX, Skype, etc. This technology is vulnerable to attacks such as eavesdropping third-party conversations, denial of service, information kidnapping, mailbox intrusion, etc. As a penetration tester, this services have to be scanned and analysed as well. A simple, non intrusive scanning could be using Shodan to detect any software related to VoIP. This could be done by searching the string “VoIP” or by directly scanning ports that are known to host VoIP related software. Another way to analyse this type of systems is using the suite sipvivious, formed by a set of tools aimed to audit SIP environments. This suite of tools allows the tester to discover devices such as software phones and switchboards, to detect extensions inside those devices, and even to launch brute-force attacks to access the devices using the previously known extensions.

### 3.1.1.2 Passive Footprinting

**Whois Protocol**

*Whois* is a TCP protocol that allows to consult information about a certain domain. It was typically used as a command-line order, but nowadays there are Internet services that offer the same functionality.

This protocol lets the tester specify a domain, and it returns interesting information such as the IP address, the name server serving it, the registrar of the domain and information about it, and contact information about the administrator and the tech support of that domain. Usually this contact information provides e-mail addresses and
telephone numbers, which can be later used in a social engineering attack. 
*Whois* also offers the option to do the opposite, to specify an IP address and receive the domain name, and all the information related to it.

**Google Hacking**

Google is the most widely used search engine nowadays and one of the reasons, besides its quickness, is that it indexes enormous quantities of web pages. It also offers the possibility of performing advanced searches using a set of operators. This operators let the users perform more effective queries, being able to restrict them to a site, or directly search in the url. This advanced operators have caused the appearance of what is known as *Google Hacking*, which consists of using search engines to find vulnerabilities in web applications, to find sensitive information or even to profile web servers.

There exist databases where it is possible to find Google queries to find interesting information. One of them is “*Google Hacking Database*”, created by Johnny Long. It indexes lots of queries grouped into categories like “Files containing passwords”, “Advisories and Vulnerabilities” or “Web Server Detection”. This queries use the said advanced operators to find information that is publicly available on the Internet, even though it probably should not be.

There also exist automated tools that perform this type of searches using known queries, available in databases like the mentioned before, into search engines like Google or Yahoo.

**Pastebin, Pastie and Github services**

Services like Pastebin, Pastie and Github offer the users a space to store text for a period of time. This services are usually used by developers to store and share pieces of code. Sometimes, this pieces of code contain sensitive information that is not supposed to be publicly available, like passwords, database structures or source code.

There exist automatic tools that go through this services searching for leaked information. One of them is called *Pastenum*, created by “Corelan Team”, which lets the tester search for a concrete character string in several services at the same time. Security teams inside companies should be aware of these practises and should monitor regularly that no information leakage is produced.
SNMP Discovery

The Simple Network Management Protocol eases the exchange of information amongst the devices connected to a network. Amongst its functionalities there is the possibility to obtain information about those devices, making it a good source of information. One of the easiest ways to discover SNMP services is through Shodan. Shodan allows to directly search for the port where this services are usually linked (port 161), discovering lots of SNMP services within seconds.

3.1.2 Web Application Fingerprinting

3.1.2.1 Surface Discovery

Web applications can be related to a given DNS name or IP address in three different ways, and we have to take them all into account.

The first one is using different base URLs. A web application can have its entry point at the base URL of the target, like this, http://www.example.com. However, there can be different applications associated to the same URL. For example, http://www.example.com/url1 and http://www.example.com/url2 could be two different applications associated to the same symbolic name. Here, the base URL could not be hosting anything, and both applications would be “hidden”, unless the exact name is known.

The second one is using different ports. A web application is usually sitting on port 80, but it could happen that another web application is placed on another port. This extra application could be an administrative interface, or something similar, and it has to be taken into account.

The third one is using virtual hosts. There could be more than one symbolic names associated to the same IP address. This 1-to-N relationship may be reflected to serve different content by using so called virtual hosts. To identify which host the request is referring, it is specified in the Host: header. An example of a virtual host would be http://www.webmail.example.com.

So, how are these two options checked?

For the first one, some guessing has to be made. If the URLs are “hidden”, no reference or knowledge of them will be present. In this case there are some options available,
first a Google search could be performed using the advanced search operator site:. This would display the “hidden” URLs if they can be indexed. If the URLs are not indexed, then a guessing process has to be made to find these URLs. This process can be made with the help of some dictionaries, or even with vulnerability scanners.

To address the second option, scanning the ports of the machine is needed. Probably this process will be already done in the fingerprinting phase of the test. Using a tool like Nmap will help discover web applications listening to non-default ports.

There are several ways to address the third option.  

*DNS zone transfers.* This method requests a zone transfer to the name server that serves our target in order to get the list of all the DNS entries for that domain. This method could not work since most DNS servers do not honour zone transfers any more. However, in order to try it the name server of the target is required. It can be obtained by issuing the host command:

```
$ host -t ns www.example.com
```

Once the DNS server is obtained, the zone transfer can be requested. The response to the request could be the desired list. This list could contain names that are not so obvious, like `webmail.example.com`, or similar. To request the zone transfer the following command has to be used, assuming that `ns1.secure.net` is the DNS server of `www.example.com`:

```
$ host -l www.example.com ns1.secure.net
```

*DNS inverse queries.* This method is similar to the previous one and could also not work. Here, the objective is requesting inverse (PTR) DNS records. In order to do that, the record type should be set to PTR and a query to the given IP address should be issued. This query should return a DNS name entry.

*Web-based DNS searches.* Similar to the zone transfer technique, but relying on web services that provide searches on DNS. This services can be queried with the target and a list of names is presented. Then the results have to be cropped to obtain the desired ones.

*Reverse-IP services.* Similar to DNS inverse queries, but querying web services instead of name servers. It is advisable to query multiple services since they often return partial results.

*Googling.* Using Google can help you refine and increment your perspective on the target. It could help discovering new non-obvious names belonging to the same target.
3.1.2.2 Application Entry Points

The structure of the target has been defined and a sitemap is available. Now it is necessary to find the attack surface of the web applications. The likely areas of weakness have to be detected and enumerated in this step.

This step aims at discovering all the browser-server connections that the application has. In order to do that, all the HTTP requests have to be analysed (GET and POST methods). This means that all the parameters and forms that these requests carry have to be noted down, as well as any custom headers and cookies that could appear. Usually, when sensitive information has to be sent, a POST request is used because it does not show at plain sight all the parameters of the request, as GET does. But even if it is a POST request, it is not safe at all, since several tools can capture this requests and extract the parameters and analyse them. This analysis can be done with an intercepting proxy or with a browser plug-in.

3.1.2.3 Robots.txt

The *robots.txt* is a file used by web site owners to give instructions to web robots about their site. When a robot wants to visit a URL it first checks the file to see if it is allowed to visit and index that URL. The file has the following format.

```
User-agent: *
Disallow: /
```

In this example, the file is excluding all robots from the entire server. You could make rules for determined user agents, or use different *Disallow* lines to exclude specific parts of your website. This is not a security mechanism, because robots can choose to ignore this file, but usually the interesting parts of a web server are listed here. This provides a good starting point for possible attackers and it has to be checked.

Getting the file from a web server is not difficult, it can be done with a simple *wget* call with the target main address with *robots.txt* appended at the end.

Once we got the *robots.txt* file, we have to analyse it. There exist some online checkers to see if the file is properly written and follows the *Robots Exclusion Protocol*. 
3.1.2.4 Analysis of error codes

When the error codes a web server provides are not handled correctly, they can give away lots of information about the web server itself and the underlying software, as for example, the database. So, the analysis of these error codes is really important because it can give away information like the OS, the type and version of the database, etc. This information can be used later on to find known vulnerabilities and the like.

To discover these error messages, requests have to be sent to the server. Some of them can be found with standard, well-formed requests, but it will be necessary to craft requests that generate error messages to get all of them.

3.2 Configuration Management Testing

The analysis of the infrastructure and the topology architecture of the target can provide lots of information about it. This has to be tested in order to limit the information leakage of the target.

3.2.1 SSL/TLS Testing

Even though there are really advanced cryptographic protocols and algorithms that would provide a high grade security, they are not as extended as it would need be. Even when the target has a high grade cryptographic protection, it could happen that due to a misconfiguration the use of a weaker cipher could be imposed by the tester, and this would suppose a security problem.

Nowadays, the http clear-text protocol is secured via an SSL or TLS tunnel. This provides https traffic, which encrypts the data in transit. These protocols can also provide identification through digital certificates. When a client connects to a server through SSL, the first thing it sends, amongst others, is the set of characteristics that the encrypted connection will have. This contains the encryption algorithm, the key length and the hash algorithm. The key length will be the aspect that will define the weakness of the whole cipher system.

So, this step is aimed at discovering weak cipher support. In order to do that, the ports associated to SSL/TLS wrapped services have to be found. To do that, the same steps
taken in the application discovery have to be found. Again, the best way is using nmap with the option -sV. This option makes nmap able to identify SSL services. Once the ports are identified, they have to be audited to check if they support weak ciphers. This can be done with OpenSSL.

Other ways of doing this would be through vulnerability scanners, such as Nessus. Nessus can scan the ports and reports all the weak ciphers that it finds.

Another aspect worth checking is certification. Certificates allow to establish the identity of one or both parts of the connection. To set up the communication a set of checks on the certificates have to be made. If any of these checks is not passed, a warning will be issued, and it could mean that a security hole exists there. So, it is important to check certificate validity. In order to do that, three different tests can be performed. First, checking if the Certificate Authority (CA) is a known, trusted one, second, checking that the certificate is currently valid, and third, checking that the name of the site and the name reported in the certificate match. All these checks are performed and reported by the browser. The list of certificates a browser has on its trust list can be checked and modified, so, it is quite easy to perform these tests.

### 3.2.2 DB Listener Testing

The Database listener is a network daemon unique to Oracle Databases. It waits for connection requests from remote clients. Since it is listening for connections, it can be attacked and the database can be compromised.

Usually, major Oracle installations do not expose this service to the Internet, so the tests should be done from the internal network. The listener is set by default to listen on port 1521, but the new officially registered ports for the TNS listener are 2483 and 2484 when using SSL. However, it is a good practise to change the port to an arbitrary one.

The potential areas of attack of this listener are to stop the listener, creating a DoS attack, to hijack the Database setting a new password, to leak information from any of the accessible files, and to obtain detailed information on the listener, database and application configuration.

To test the security of the listener, the first required thing to do is to find the port where it is listening. After doing that, a tool developed by Integrigy can be run. This tool
tests if the password for the listener is set or not, if the logging system has been enabled or not, and if the admin restrictions are set or not. These three aspects are important, since each one of them represents a different attack area of the listener, and they should be addressed if found incorrect.

### 3.2.3 Infrastructure Configuration Management Testing

The target of a penetration test is usually not only a single web server, but a complex infrastructure of web applications mixed together to provide a service. This infrastructure could contain not only the web server on itself, but also application servers, database servers, firewalls, proxies, and even administrative interfaces. When these infrastructures grow in size, the developers tend to pay more attention to the most important parts and neglect the smaller, non-crucial or outdated parts. But even if the smallest component of the infrastructure has a security hole, it could compromise bigger, more important parts of the system.

So, understanding the said infrastructure is vital in a penetration test. Discovering not only the web server and its version, but also the existence of application servers, the usage of proxies, firewalls or other security mechanisms, the existence of legacy systems still connected to the updated application. Not only discovering the different parts of the system is vital, but also understanding how they communicate with each other. All these could be used as an entry point for an attacker, and they have to be discovered and analysed.

Web applications also require the usage of administrative tools to manage content, application source code, or user authentication databases. This administrative tools can be built-in on the server itself or provided by a third party. Also, these tools could be accessible from a private Intranet or from the Internet. In the case these tools are accessible from the Internet, the authentication protocol and access control of these tools have to be analysed thoroughly, since they can suppose a perfect entry point for an attacker.

### 3.2.4 Application Configuration Management Testing

Once the whole structure has been checked, the configuration of the single applications inside the system have to be checked as well. Each component of the structure could
have its own vulnerabilities that might compromise the entire architecture. Most of the systems used in web services usually come with generic configuration files and code samples to show instantly how the service works. This is a great way for developers to get to know the systems they have to work with, but it can pose a thread to the overall system when these configuration files and examples are kept in the production environment. Some of these examples are known to have vulnerabilities and should be removed. To scan for these known files the use of a CGI scanner is recommended. Another thing that could give away information about the configuration of applications is code commenting. It is known to be another great tool for developers to understand code and to clarify it, but when comments reach the final user they could also be a thread to the application. Comments in the HTML or JavaScript visible from the client-side should be removed.

3.2.5 Testing for File Extensions Handling

File extensions give away information about the technologies used inside a web application and they can also help understand its behaviour. It could not seem like a security threat, but it tells the attacker where to start searching. Apart from giving away the technology of the application, file extensions should be controlled also in the means of which files does the client receive. There are certain files that contain private information, like configuration files, that always use the same extension. These extensions should be controlled and never returned to the client. Also, a deep inspection of the file types returned to the client needs to be done. All files should be checked to verify that they are indeed supposed to be served and that they do not contain sensitive information. This tests can be done using Web Spiders, Vulnerability Scanners, or manually inspecting the application, which can be done with the help of a search engine.

3.2.6 Old, Backup and Unreferenced Files

Most files in a server are used to provide the normal execution of the application itself. But sometimes, there are files that do not serve this purpose. These files could be old versions of current files that are working, automatic or manual backups, or even unreferenced files that none remembers that are there. These files could pose a thread
to the application giving away sensitive information like source code or configuration or credential files.

Backup files could be easy to forget, since sometimes they are automatically generated in a way transparent to the user. For example, when editing a file, some text editors will create a copy of it, and it could stay in the server if not removed. This backup files, or compressed user-generated files, if accessible from the client side, they will not be interpreted by the browser, but downloaded directly and they could contain important information such as source code, which would give the attacker a good insight of the inner workings of the application.

Another similar case are old files. When a file is updated and the old file is left there only with the appending of `.old`, if retrievable from the client side it will directly show the source code of a previous version of the application’s code. This could also give the attacker a huge advantage when trying to find a vulnerability in that code.

This situation may lead to severe threads to the application, and should be addressed with a security policy accepted by the developers which should forbid dangerous practises. This practises would include editing files on the web server file system directly, leaving behind archive files, not deleting old or unreferenced files, etc.

3.2.7 Infrastructure and Application Admin Interfaces

Admin interfaces are almost always present on web applications. They allow to perform privileged activities on the site, which should be only done by authorised people. Checking if this is so is an important part of a penetration test.

First, the tester has to find if an administrative interface is present. This could be as simple as requesting `/admin` on the main site, or it could be done through parameter passing or a cookie variable.

Once found, the authentication protocol has to be checked to make sure that only authorized users can access the interface. Some protection against brute force attacks should also be considered.

3.2.8 Testing for HTTP Methods and XST

There are several methods offered by HTTP, even though only a few are commonly used. However, if a web server is misconfigured, these HTTP methods could be used to
perform malicious actions on the server.

As it is known, GET and POST are the most common HTTP methods, used to access and modify information on the server respectively. But there are other methods that are used, for example in REST Web Services (like PUT and DELETE). So, besides these well known methods, there are methods that are not that well known and that could pose a security thread. The dangerous HTTP methods for a web service are PUT, which allows to upload files, DELETE, which allows to delete files on the server, CONNECT, which could allow a client to use the web server as a proxy, and TRACE, which echoes back to the client everything that has been sent to the server. There is another HTTP method that would allow attackers to discover which HTTP methods does the server allow. This method is called OPTIONS, and it returns the list of methods allowed.

So, if the server really needs to allow all these methods, it should do it in a really controlled and monitored way, to ensure that they are used correctly and by trusted users. There are some attacks that could be done with this setup. For starters, there are web servers that allow arbitrary HTTP methods. This means that a web server would allow to receive a "JEFF" or a "CATS" method and it would be treated as a GET method with the particularity that it would not be subject to role based access control. This means that an unauthorized user could send a GET request and receive the appropriate response. Using this, for example, the unauthorized user could create a new user and then give it admin credentials, all of this sending requests with arbitrary HTTP methods.

Another attack that could be performed using HTTP is XST, called Cross-Site Tracing because it uses the HTTP method TRACE. This method, as said before, returns everything that was sent to the server and it is used for debugging purposes. However, it can be used to perform a sort of Cross-Site Scripting attack. This attack would be also directed to stealing a user’s cookies. This is so because with the TRACE method everything that is sent to the server is returned, including the cookies. To perform this attack, the attacker would need to make the user send a TRACE request to the server. When returned, this cookie could be stolen using another vulnerability, be it server-side or client-side, and then used to steal the user’s session on the server.
3.3 Authentication Testing

Authentication is the act of identifying something or someone, making sure it is authentic. In computer science, authentication is the act of verifying the identity of the sender of a communication. This is usually made through a login process. Testing the authentication means understanding this process and using that information to bypass the authentication mechanism.

3.3.1 Credentials Transport Over an Encrypted Channel

When a user logs into a website, there are different options to send the user’s credentials to the web server. The credentials can be sent using the POST or the GET methods, and the connection between the client and the server can be through HTTP or HTTPS. In this stage of the penetration testing, the usage of HTTPS is verified. This is so because the credentials sent through HTTP are sent in clear and could be easily captured with a network sniffer. So, it is crucial to make sure that the credentials are sent through HTTPS because it is built on TLS/SSL to encrypt the data that is transmitted, ensuring that even if sniffed, it will not be available for the attacker.

To check whether the credentials are sent through HTTP or HTTPS a sniffer has to be used to capture the packets and inspect them. In the captured packet, it can be seen that if the credentials are sent through HTTPS, they will be addressed to the HTTPS port of the server, usually port 443, instead of the standard address. On the other hand, if the credentials are sent to the standard address, it means that the connection is made through HTTP and that these credentials can be stolen with a network sniffer.

3.3.2 Testing for User Enumeration

When a login is made with incorrect credentials, a message is received from the web server. Sometimes, these messages can give away some hints over the user database of the application. This fact can be used to identify a set of valid usernames, which could be later used in a brute force attack.

In this test, the tester should try to analyse the response messages from the authentication system, comparing the results when different credentials are provided. Sometimes,
when a valid username is provided, the server returns messages like “wrong password”, instead of “authentication failed” or something similar that does not give away that the user exists. When these messages are discovered, the tester can begin a guessing attack to identify a set of valid usernames simply by guessing the names and checking the return message. This can obviously be automatized.

Sometimes the difference in the return message is more subtle, like for example using URL redirections, or different error codes. The tester has to check for all the different possibilities.

Another strategy for discovering usernames is URI probing. This method consists in accessing user directories and checking the received error code. Some web servers, when an existing directory is visited an error 403 Forbidden is returned while a 404 Not found error is returned when visiting non-existing directories.

Analysing the recovery facilities of an authentication system is also a good way to discover usernames. When a valid username is introduced, the password reset mail will be sent, and it will be stated there that it was sent, while when an invalid username is used, a message will be displayed stating that the user did not exist. This method can be used when the recovery system works with usernames instead of e-mail addresses.

### 3.3.3 Default or Guessable (Dictionary) User Account

Most current web server engines and underlying softwares are provided with a default administrative account that is used by the developers. Usually this account is not deactivated on the production environment and could be used to gain access on the system. There can also be backdoors to the system left by the developers to access and test the application, or simply, administrators that use easy usernames and passwords.

All these things can be used by an attacker to gain access to the system with admin permissions. The default accounts on the current systems are known and can be found easily on the Internet. Also, developers, testers and administrators of web servers usually use common usernames and passwords, like “admin”, “test”, “root” or similar, with passwords that resemble the username.

Discovering usernames can be done by reading the comments on the available source code, by finding the names of the developers of the web application, or by trying permutations of the name of the company that owns the system. Using the testing method in the previous section can reveal the list of usernames with admin permissions of the
system. Once the list is long enough, a password guessing or brute force attack can be done.

### 3.3.4 Testing for Brute Force

At this point of the penetration test, a few usernames could have been retrieved. A brute force attack could be tried out now. In order to perform a brute force attack, it is necessary to know how the authentication of the target works. There are two common types of authentication.

The first one is using **HTTP authentication**. This method relies on HTTP messages to perform the authentication. When a user tries to access a resource that requires authentication, the server sends back a 401 response containing a “WWW-Authenticate” tag containing the type of authentication and the realm where the client wants to authenticate to. Then, the browser directly displays a message where it asks for the username and the password, and then send it to server. There are two different types of HTTP Authentication. The first one is called **Basic**, and it simply sends the username and the password separated by a colon, and then encoded with base64. This method could be easily be sniffed and decoded by an attacker. The second one is called **Digest**, and it uses a one-time nonce sent by the server to create an MD5 hash of the password. This is more secure than the Basic one, but the username is still sent in plain text.

The second one consists on custom **HTML form authentication**. This is the most widely used method, and it simply consists on creating an HTML form where the client enters the credentials. This credentials are then sent to the server. Usually, the credentials, or at least the password, are sent hashed, but it is not an HTML feature, the developers have to do it.

Once the Authentication type has been identified, there are several types of brute force attacks that could be performed.

*Dictionary attacks* consist of automated scripts that try to guess usernames and/or passwords from dictionary files. This dictionary files could consist of common passwords, or could be made from information gathered by the attacker from the web site or its clients.

*Search attacks* are the most known attacks. This attack is based on covering all possible combinations of a given set of characters and a length range. It is the slowest type of attack because it tries all possible combinations with the given set.
Rule-based attacks use rules to generate better candidates for passwords. For example, they can use part of the username and some pre-set rules to generate commonly used password variations.

There exist tools that automate these brute force attacks. One of this tools is Hydra.

3.3.5 Testing for Bypassing Authentication Schema

As it is known, the authentication process is meant to give access to private areas of a web site only to the users that have permissions to access them. However, the authentication schema used in some web sites is not completely adequate and can be bypassed.

There are several problems that could be found in an authentication schema, be it for negligence, ignorance, or understatement of security threats. This problems can be found in the different stages of software development. For example, on the design phase, the designers could make a wrong definition of the parts to be protected; on the development phase the input validation functionalities could be incorrectly implemented; and even on the deployment phase there could be errors due to poor documentation or to a lack of technical skills. These problems can be bypassed with several methods.

1. Direct page request: Some websites only control authentication on the login page. If this is so, some of the supposedly private resources could be accessible only by directly accessing them with the browser. Simply writing the page on the address bar on the browser is the most straightforward way to test this method.

2. Parameter Modification: Some servers use parameters on the request to check if the user is authenticated or not. To test this method, the tester should tamper the request, either directly on the URL, or using a proxy to capture the request and modify the POST parameters.

3. Session ID Prediction: Some servers use session identification values to automatically manage authentication. If these session IDs are predictable, the tester could try to find a valid session ID and gain unauthorized access to the application.

4. SQL Injection: This method could be used to authenticate without knowing the correct password of the user. To use this method certain conditions have to be met, which will be discussed later on.
3.3.6 Testing for Vulnerable Remember Password and Pwd Reset

The password reset function is really handy when someone forgets or loses his/her password. However, it could be a thread to the user depending on how is it implemented. The remember password feature is also useful for the average user, but depending on its implementation it could allow an attacker to gain access to the system impersonating a user.

**Password Reset**

This feature has two common ways of working. The first one is *sending an e-mail* to a previously known address. The security of this method relies only on the security of that e-mail address. The system assumes that it has not been compromised. The second one is through *secret questions*. If a higher grade of security needs to be achieved by the system, a combination of both methods could be implemented.

The secret questions method relies on a set of questions, usually predefined, that the user has to answer when s/he registers. This questions are usually personal, and should be difficult to answer unless you are the user. However, sometimes these questions can be answered with some Googling, or using some sort of social engineering attack. Another thing that should be taken into account is the number of questions the user has to answer, or the number of times the user can fail a question before the system blocks. All these aspects have to be taken into account when testing the system.

Another point to take into account is how the system presents the password to the user. There are several ways. The least secure would be presenting the password in clear text to the user. This would allow an attacker to log into the system without the user even knowing it. Another insecure method would be directly asking the user to change the password. This would provide the attacker with a way to log into the system, isolating the original user outside. So, the common method again would be to send the new password or a password reset link to a previously known e-mail address. This would again rely on the e-mail account not being compromised.

**Password Remember**

This feature helps end users save time when logging into systems. However, it could help an attacker gain access to the system. There are two ways of implementing this feature.

The first one is using the “cache password” feature in web browsers. This method could
be heavily insecure and it should be disabled. If an attacker would have access to the browser’s cache, s/he could obtain the password in cleartext. The second one is using permanent cookies. When using cookies, the password should be at least hashed, and never sent in cleartext. To check it, the cookies stored by the application have to be examined, and the hashing algorithm should also be checked to see if it is secure enough.

3.3.7 Testing for Logout and Browser Cache Management

The logout feature is a really significant one, since without it, a user’s session could be reused by an attacker to get inside the system and get sensitive information from it. First of all, the logout feature should be available in two different cases, when the user actively logs out and when the user remains idle for a specified period of time. The first case is easily checkable, since a logout button or a logout function should be available from the moment the user has logged in. In the concrete, web site login case, the tester also has to check what has happened to the session cookies. They should be properly invalidated, setting them to null or to an invalid value. From here, the next immediate test is checking that the session is correctly invalidated. This is done hitting the “back button”. If after hitting the back button the tester can see pages that s/he previously visited, but not navigate into new pages, this means that the session has been correctly invalidated and that some pages were cached in the browser. After this easy test, a more complex one can be performed. It consists on setting the session cookies manually to a valid one, using a proxy or a browser add-on, and checking whether the system allows to navigate to pages that only logged in users can navigate. If it is possible, it means that the server does not does not keep track of the active and non active session cookies, and it could be a security threat. In the case of a user staying idle for a specified amount of time, the user should be logged out, and in the web case, the session cookie should be automatically invalidated. To test this, the tester simply has to wait for the system to log him/her out. The time it takes to log out a user due to inactivity should be taken into account, since longer times pose higher threats to the users, and it also depends on the system under study. Also, the tester has to check whether the log out is enforced by the server, by the client, or both. This could be checked simply by inspecting the session cookie. If it has a login time, or expiration date, it could possibly mean that the log out is enforced from the
client. Tampering the session cookie can be a good way to ensure that the server side also controls invalidated sessions.

One last thing that should be also tested are the cached pages. If the system allows the browser to cache pages with sensitive data on it. This data could be easily gathered by an attacker simply by accessing the browser cache. There are two ways of enforcing the browser not to cache certain pages. It can be done through an HTTP response header, or directly through an HTML tag on the code.

3.3.8 Testing for Multiple Factors Authentication

Some systems that work with sensitive data, such as banking systems, usually provide multiple factors authentication to increase their security. This authentication system works by providing an additional step in the authentication process. This new step is usually related to “something you have” instead of “something you know”, providing the user with a hardware device besides from the password.

There are several threats for this type of processes nowadays. These threats include Credential Theft, Weak Credentials, Session based attacks, Trojan and Malware attacks and Password Reuse. The optimal solution should address all the attacks related to these categories. Unfortunately, it is really difficult to be protected from highly motivated attackers.

There are several solutions out there, but not all solutions protect from all the mentioned threats. This solutions provide with different ways of obtaining an additional password that the user has to enter when s/he logs in. The process to obtain this password can be done in several ways:

1. One-time password (OTP) generator token
2. Grid Card, Scratch Card
3. Crypto devices like USB tokens or smart cards
4. Randomly generated OTPs transmitted through a GSM SMS message

These methods have their weaknesses and their strengths, but there’s only one of them that addresses all the threats mentioned before. Transmitting the randomly generated
OTP through a secondary channel, such as an SMS, is considered one of the best ways to do a two-factor authentication.

3.4 Session Management Testing

The Web is based upon HTTP communication, which is a stateless technology. But to provide some user experience and allow interaction, web servers need to keep track of the users’ actions on their sites. In order to do that, a session system has to be created on top of HTTP. This system will link all user’s activities to a single stream of actions, called session. Sessions need custom, third-party solutions to work. This solutions could be customly implemented by the developers or provided by a middleware. To manage these sessions, an authentication token needs to be issued to the user. This tokens are usually called Session ID or Cookie.

3.4.1 Testing for Session Management Schema

As said before, sessions manage the actions of a user on the system through authentication tokens. This tokens, called cookies, have to be created in secure and unpredictable ways and transmitted and stored encrypted or hashed. If these is not so, an attacker could easily steal the cookies or forge a new one in order to hijack a session from a legitimate user.

A cookie is created when the user first logs in and is sent to the user’s browser. Afterwards, every time the user performs an action, this cookie is sent to the server to authenticate the user. The data inside the cookie could be diverse, providing the server with information about the user, and thus, creating a state inside a stateless protocol like HTTP.

The aim of this section is to try to forge a valid cookie to enter the system as a logged user. In order to do this three steps will have to be completed. First, collecting a sufficient number of cookie samples; second, reverse engineer the collected cookies to analyse the generation algorithm; and lastly, try to forge a valid cookie to perform an attack.

When collecting cookies, the tester has to bear in mind that there could be multiple factors that affect the cookie generation method. These factors could contain, amongst others, the time of generation, the user browsing the system, the past actions of the user.
or the point in the application where the cookie has been generated. So, it is important to collect cookies noting down all these factors, and trying to variate just one factor between cookie collections. Automated cookie collection is advisable, since small time variations could significantly change the cookie value if the timestamp is used as seed for the generation process.

Once a sufficient amount of cookies has been collected, they have to be analysed in order to discover their structure and possible information leakage. Cookies can sometimes provide specific data about the user, this has to be checked. Cookies are often encoded or hashed. The tester has to discover which type of encoding has been used and try to decode it. Sometimes a hexadecimal or base64 encoding is used, making it easy to decode. If a hashing algorithm has been used it could be more difficult to accomplish. When decoded, the structure of the cookie can be seen directly or deduced. When the structure is known, the tester should try to assess its predictability and randomness. Examining different cookies generated almost equally, the generation algorithm could be deduced. This process could be easy, being able to deduce it manually, or really complex, needing some automatic mathematical or statistical tool to understand how the cookie is generated.

Once the tester has discovered as much information as possible about the cookie’s structure and generation algorithm, the next obvious step is to try to forge a valid cookie in order to hijack another user’s session. This step completely depends on the structure of the cookie and how it is generated and transmitted through the net. The consequences of modifying a cookie could vary from impersonating another user, to accomplishing a buffer overflow of the variable where the cookie is stored on the server.

### 3.4.2 Testing for Cookies Attributes

As seen before, Cookies are the usual method to maintain a session for a user. Cookies have a series of default attributes that should always be used correctly. If these attributes are not correctly specified, they can provoke vulnerabilities on the session management system. To check that these attributes are correctly used, an intercepting proxy can be used to intercept the responses where a cookie is set. Once the response has been captured, the cookie can be analysed. The attributes that could provoke a vulnerability are the following.

**Secure** - This attribute specifies if the cookie has to be sent through an encrypted,
secure channel or not. So, when a cookie contains sensitive information, this flag to be specified in order for the browser to always send the cookie through HTTPS instead of HTTP.

**HttpOnly** - This attribute does not allow the cookie to be accessed by any technology besides Http. This attribute is used to avoid a cross-site scripting attack by not allowing JavaScript to access the cookie. This attribute should be always used, even though not all browsers support this feature.

**Domain** - This attribute specifies the domain where the cookie can be sent to when making a request. The domain attribute should be specified correctly, otherwise it could provoke a threat. For example, if a cookie’s domain attribute is set to `app.domain.com`, the cookie will be submitted to this domain and all its subdomains, but not to another domain, such as `app2.domain.com`. However, if the domain is set too loosely, for example at `domain.com`, then the cookie would be submitted to `app.domain.com` and also `app2.domain.com`. In this case, if one of these domains were compromised, it could be used to harvest cookies.

**Path** - It is checked after the domain attribute, and it is used to specify the URL path where the cookie is valid. If the domain and the path match, then the cookie will be sent in the request. Again, this attribute should be also specified correctly because it could cause the same problem as the previous one.

**Expires** - Used to set persistent cookies. It specifies the date when the cookie expires. Passed this date, the browser deletes the cookie. If this attribute is not specified, the cookie only lasts for the current browser session. When a cookie has an expiration date, it is stored in the user’s hard drive and could be retrieved by an attacker. So, if a persistent cookie has to be used, no sensitive information should be stored in it.

### 3.4.3 Testing for Session Fixation

The authentication system of a web server should issue a new cookie identifier upon logging in. Otherwise, a user account hijack could be possible.

More concretely, as an example, a user requests a web page from a server, this server returns the web page issuing a new cookie with a session id. Then, the user authenticates to the web server sending the cookie s/he has received. Two things could happen now. The web server could respond to the authentication sending a new session id to the user. This would be the correct behaviour. But, it could happen that the server responds to
the user without creating a new session id. In this case, the server would be vulnerable to a session hijacking.

In order to perform it, an attacker could send a valid identifier to a user and wait for him/her to authenticate in the system. When the user authenticates, the attacker will have a cookie with the privileges of the authenticated user, having achieved a session hijack.

3.4.4 Testing for Cross Site Request Forgery

The CSRF attack is based on using the authentication process of a website to make a logged user do something in that website without its consent. It works by sending a known URL to a user. This URL usually performs an action on the site that requires a logged user. If the authentication system relies only on information known by the browser, the action will be performed without the user even noticing.

The usual way of sending this URLs to vulnerable users is through the \textit{img} HTML tag. This tag allows to load an image from an external source, allowing to indicate a URL in the \texttt{src} attribute. So, if the attacker sends a link to the victim that contains an image tag with 0 width and 0 height and the action from the desired website as source, the victim will not even notice s/he is in a compromised website.

To test this issue, the tester can take the roles of attacker and victim if s/he has access to the restricted area of the target. If this is so, the tester should search for the appropriate URL, then build an html page containing the http request referencing the said URL, make sure that s/he is logged on the application, access the compromised html page and finally check if the web server executed the request. If the request has been executed, the target is vulnerable to CSRF.

In order to avoid this kind of attacks, the URLs that could be susceptible to being attacked should include session-related information. This way, the attacker will have a lot more trouble crafting the URL. Also, using POST instead of GET is a good countermeasure, since POST attacks are harder to mount.
3.5 Authorization Testing

Authorization is the step that comes after authentication. It is based on giving roles and privileges to the authenticated users, giving them access to certain resources. Testing for authorization means understanding how this process works and trying to bypass it.

3.5.1 Testing for Path Traversal

Many web applications provide access to files or resources, and they usually confine the files a user is able to access inside a “root directory” which represents a physical directory on the server. Web servers have mechanisms designed to control who can access certain resources and who cannot. So, for example, an external user will not be able to access resources outside the said “root directory”.

The problem here is that web applications usually have complete access to the server. So, if the web application uses a server-side script to access files that is poorly implemented, it could be an entry point for an attacker. This type of attack is known as path traversal.

In order to test for this, the following two steps have to be checked.

- Input vectors enumeration
- Testing techniques

**Input vectors enumeration**

The first step is to find all the application parts that could be vulnerable to input validation bypassing. This parts include GET and POST queries, HTML forms and file uploads, as well as all other parts that accept content from the user.

Some examples of possibly vulnerable URLs:

- [http://example.com/main.cgi?home=index.htm](http://example.com/main.cgi?home=index.htm)

There could also be cookies that are used for the dynamic generation of pages/templates.

**Testing techniques**

Once the input surface has been established, the input validation functions of this surface have to be analysed. So, starting from the first example URL, it can be seen that it loads the resource called content. If instead of content, an attacker would insert the
string “../../../etc/passwd”, the password hash file of a Unix system would be shown. Obviously, in order to perform this attack, the underlying system has to be known. It would not make any sense to request the passwd file in a Windows server. This attack could be used to display restricted files, since the application is the one requesting them. Also, this attack can be used to display the source code of the application itself, in order to gain more knowledge about it to discover new attack points. For example: 

http://example.com/main.cgi?home=main.cgi

This example would display the source code of the “main.cgi” file without even using any path traversal chars.

Usually, the applications use countermeasures to avoid this kind of attacks, such as not allowing the slash character, or by checking that the string ends with a certain file type. This kind of countermeasures can be bypassed by encoding the path traversal string or by using special characters such as the null character. It also has to be taken into account that each operating system uses a different character as path separator.

### 3.5.2 Testing for Bypassing Authorization Schema

This test focuses on verifying the authorization schema of the application. It tries to discover if a user can access resources or fire functions that are denied to its role. The objective of this test is to find whether a user without administrative privileges can access resources or functions that are reserved to administrative users. For example, the target could have a function to create a new user. If an unauthorized user were able to use this function to create a new user, this would be a security threat, because this newly created user could be created with administrative privileges, and thus, the attacker would have complete control of the application.

Another test would be to try to access resources that are only accessible for certain roles. For example, inside a shared directory where certain files are supposedly accessible only by users with roleA, if one of these files is accessible by a user with roleB, this could also be a threat to the system.

### 3.5.3 Testing for Privilege Escalation

Privilege escalation is the act of modifying a user account’s privileges without the permission to do so. There are two types of privilege escalation, vertical and horizontal.
Vertical privilege escalation is when a user gains access to resources granted to users with more privileges than him/her. This could be used to perform administrative actions on a system without being the administrator. On the other hand, horizontal privilege escalation is when a user performs actions that belong to a different user with similar privileges. This could be used to access another user’s bank account in an online banking system.

In order to test this, the tester has to check all the portions of the application where a user can create, receive or delete information from the database. The tester has to check whether s/he can access any portion of the application as another user or not. The tester should try to find where the role of the user is specified, if it is specified somewhere besides the server. If a GET or POST request contains a parameter indicating a group or role, this could be susceptible to a privilege escalation attack.

### 3.6 Business Logic Testing

The inners of an application, which determine how is it possible to interact with application data, is known as business logic. This part of the application determines how data can be created, displayed, stored and changed. Usually, the processes inside the business logic of an application have a well defined structure, following a certain order to perform certain actions. However, executing these actions in ways the developers did not expect, or trying to perform them in a different order from the one established could cause a flaw on the normal operation of the system. For example, if an authentication system needs three steps to be completed in order, what would happen if a malicious user goes from step 1 directly to step 3? Would it be possible to access the system without authenticating? Discovering this type of things is the aim of this test.

As the reader could have guessed, this type of tests cannot be performed by automatic vulnerability scanners. These tests are based on the understanding of the application and the creative thinking of the tester. Since these tests are based on the tester’s creativity and intuition, it is really difficult to systematically address them. There are no concrete guidelines for these tests, the tester will have to analyse the system and construct the tests for him/herself. An approach to constructing the tests is provided, though.
Understanding the application. This first step is crucial to discover the vulnerable points of the application. If available, the documentation of the application greatly helps with this task, since it describes the expected workflow of the application. The use cases designed by the developers could also aid the tester in this step, since they often describe the limitations of the acceptable usages of the application.

Creating raw data for designing logical tests. In this stage, data about the business logic has to be gathered. All the business scenarios should be identified, as well as the different workflows. On top of that, the roles available on the system have to be enumerated, identifying exactly which privileges each role has. If the application provides groups or departments, they should also be identified and enumerated. Once all this information is identified, a privilege table could be constructed to display it.

Developing logical tests. Here, the privilege table created on the previous step could be used to define the tests. Also, the sequences of user interaction with the system have to be checked, trying to visit them out of order. All alternative transaction paths should be checked, if the same thing can be done in different ways, all these ways should be tested. In addition, all client-side validations have to be examined as well, as they could not be double checked on the server.

Standard prerequisites. In this stage, the user could create dummy users with different permissions to perform the tests.

Execution of logical tests. All the tests should be performed, analysing them with care and trying to discover if there are vulnerabilities or not.

3.7 Data Validation Testing

Not validating the data that comes into the application is one of the most common weaknesses of web applications. All data coming from a user has to be validated properly before using it. Failing to do so can lead to any of the major vulnerabilities in web applications. These vulnerabilities include cross site scripting, SQL injection, interpreter injection and buffer overflow attacks. Input data from a client should never be trusted, and always validated. This is a difficult task in bigger system though, because large systems usually have lots of entry points and validating all of them is a huge task for developers. In this section, the attacks regarding data validation will be visited.
3.7.1 Testing for Reflected Cross Site Scripting

This type of XSS attack is one of the most used nowadays. It does not load with the web server, it loads from the URI followed by the user. This attack has a great social engineering component, since the attacker has to get the victim to follow a URI that contains some script in it that will perform a malicious action. Usually, this malicious actions contain installing key loggers, stealing cookies or changing the contents of the page.

To test for this vulnerability two steps have to be followed, first, all input vectors have to be detected. Wherever the application let’s the user input something through the URL is a susceptible input vector. Second, all the input vectors have to be analysed to detect potential vulnerabilities. In this step, the tester has to craft some input data to see if some response is triggered. The usual crafted input is an alert, since it is harmless but graphical. If an alert works, any JavaScript code could be inserted there, having discovered a vulnerability.

An example of a XSS attack would be the following.

Imagine a simple web site with a welcome message that outputs “Welcome %username%”, if the URI to this site is http://example.com/index.php?user=Tester, the site will output Welcome Tester. Now, if a malicious user crafts the following URI, an unexpected behaviour will take place.

http://example.com/index.php?user=<$script$>$alert(Hello!)$<$script$>$_$

When this URI is loaded, an alert will be displayed. In this case, this alert is harmless, but it means that a XSS attack could be performed, using any JavaScript code the attacker wants.

The tester has to take into account that most applications might filter out key words like “<script>” as a sanitation process. Writing the tags with a different encoding is a usual practise to avoid this type of filters.

3.7.2 Testing for Stored Cross Site Scripting

Stored XSS is a different type of attack, arguably the most dangerous one. Instead of relying on the URI the user clicks, it relies on a piece of JavaScript code directly stored in the web site. So, when a user browses to the compromised website, his experience will be the usual one, but the JavaScript will be executed by his/her browser. This attack
does not have the social engineering component of the previous one. In order to store
the JavaScript into the system, the attacker simply has to use an input form that stores
information on the database. When this information is viewed by another user, the code
is executed. This attack is more dangerous because it is not focused on a single user,
onece the web site is compromised, all the users that browse through it will be attacked.
This includes possible administrators of the same website.
In order to test this kind of attack, the first thing the tester needs to do is to identify
all the input forms of the target. This includes user profile page, shopping carts, file
managers, etc. When all the input forms have been identified, the HTML code that
composes them has to be analysed, to check if it will be directly stored in the database,
and how it is positioned in the context of the page.
If there exist some client-side validations of the input forms, JavaScript can be disabled,
or the tester can use an HTML proxy to modify the request after the validation has
been passed.
With this kind of attacks, JavaScript exploitation frameworks can be used. These frame-
works help the attacker craft some JavaScript code that will directly connect all the com-
promised browsers directly to the framework itself, gaining full control of the browsers.
Another attack vector that the tester has to take into account are the file upload fea-
tures. If they allow to upload TXT or HTML files, the feature could be vulnerable to a
XSS attack. It should be verified that the upload does not allow to set arbitrary MIME
types.

### 3.7.3 Testing for DOM Based Cross Site Scripting

Both previous XSS attacks rely on information that is sent by a malicious user, reflected
or stored by the server, and seen by one or many users. DOM based XSS does not
rely on this technique, and it makes it harder to discover. This type of attack relies
on client-side code that populates the web site. For example, JavaScript code that uses
the DOM element document.URL to fill some information. The same example as the
reflected XSS attack can be used. In this case, the code that writes down the username
could be a script that uses the document.URL element. For example, with this code.

```html
<SCRIPT>
var pos=document.URL.indexOf("user=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```
Using the same URI as the reflected XSS attack would generate an identical attack, it would go first to the server, and then back to the user. If XSS prevention techniques are applied on the server, the attack would be useless. However, if the following URI is used, the attack would be totally different.

http://example.com/index.php#user=$<$script$>$alert(Hello!)$$<$$script$$>$

The only change is replacing ? by #. This simple replacement tells the browser that everything beyond the number sign is a fragment, not a parameter, and this is not sent to the server. With this simple modification, all the possible XSS detection techniques are ineffective.

In order to test for this attack, the tester has to search for all the DOM references used on the client side code susceptible for manipulation. This references include:

- document.URL
- document.URLUnencoded
- document.location
- document.referrer
- window.location

Besides from this references, all the scenarios where the DOM is modified by client-side code should also be inspected.

### 3.7.4 SQL Injection

This is one of the most well-known attacks, and it consists of injecting an SQL query in an input form of a website. A successful SQL injection could read sensitive data from the database, modify the database itself, execute administration operations on the database and recover the content of files from the database. When performing an SQL injection attack, there are two possible outcomes, the first one is the web site displaying an error message generated directly by the database. This output gives lots of information to the attacker. The second outcome is the web site displaying a custom error message. This gives no information about the database and forces the attacker to perform a “blind
SQL injection.
The first thing to do to test for SQL injection is to detect all input fields where the application connects with the database. Typical examples would be authentication forms, search engines and e-commerce sites. When all the input forms have been detected, the first easy test is to check if the inputs are susceptible to the characters single quote (') or semicolon (;). This two characters are the special characters for string and query terminator respectively, and if not correctly filtered they will cause an error on the database, which means that this field is susceptible to SQL injection. If the error is propagated to the user, a lot of information will be leaked out. Otherwise, if the user gets a custom error page, it will be more difficult to obtain that information.

Consider an authentication form that uses the following query to check if a user exists and the provided password is correct.

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

If the variables are directly obtained from the user via an insecure form, a malicious user could input the following.

```
$username = 1' or '1' = '1
$password = 1' or '1' = '1
```

The query will then be:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

As it can be seen, this query always returns true, so the malicious user could authenticate to the system without having any username and password.

Now consider the following query.

```
SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))
```

Here there are two issues, the parentheses, and the MD5 hash of the password. In order to address these two issues the following input can be used.

```
$username = 1' or '1' = '1')/*
$password = foo
```

Here, the characters /* have been used. This characters designate the beginning of a comment, and everything after them is ignored by the database. Also, a number of
parentheses has been added to match the opened parentheses of the query. This query returns a number of values, and sometimes the authentication code verifies that only one value has been returned. To bypass this check the only thing needed to do is adding the operator “LIMIT <num>”. This operator limits the number of values returned. Adding it just before the comment character will help bypass this type of checks.

Another family of SQL injections is the usage of the UNION operator. It allows to join two different queries in one, if they have the same number of parameters. This operator is useful because it lets the user unite queries from different tables. If the malicious user knows or discovers the names of tables with sensitive information, s/he can output the tables using this operator. Consider that the web application has the following query.

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

If a malicious user uses the UNION operator to append another query after the id, the final query could end up like the following one.

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

In this new query, the malicious user wants to output the credit card numbers stored on the database. The SELECT needs the same number of parameters as the original query, that’s why the malicious query has two extra parameters without names. The keyword ALL is also necessary to get around queries that use the DISTINCT keyword.

As mentioned before, there exists another type of SQL injection, named Blind SQL injection. With this type of injection, the attacker does not have any feedback from the application, due to custom error messages. Here, inference methods have to be used in order to obtain information from the database. This inference methods use a series of standard functions present in almost every database. This functions allow the tester to identify the characters of a value one by one. To do so, the query has to obtain a single character and then compare its ASCII value to the input of the user. Using this technique and trying one by one all the characters, the attacker can deduce the value of a field from the database. An example query would be the following.

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```
This query will return a value if and only if the first character of the value in the username field is equal to the ASCII value 97. If it does not return a value, the attacker increases the 97 to 98 and submits again. When the first character is found, the starting point of the SUBSTRING function is increased. As the reader could have guessed, this method is really slow. That is why there exist automated tools that perform this kind of attacks. One of this tools is SqlDumper, which performs requests to a MySql Database through GET requests.
Chapter 4

Design and Implementation

4.1 Objectives

The main objective of this project is to automate the penetration testing process in order to aid security experts in their work.

This main objective creates several key points that need to be noted as well. These key points will be addressed in this section.

- The system should cover as many parts of a penetration testing process as possible.
- The system should be configurable by the security expert using it, in order to suit its preferences.
- The system should be able to incorporate new tools in an easy way.
- The results found by the system should be presented in an easy-to-understand way.

So, this main key points are the objectives that this system needs to cover to be able to really help in the penetration testing process.

The first key point is really important, since the more parts of a test this system covers the more useful it will be. A system that only runs the tool Nmap will not be of any use to a penetration tester, since s/he can run Nmap directly instead. The objective is to cover most of the parts of a penetration test, beginning with the Information Gathering
step and using the information obtained to advance through the different steps and produce an easy to understand report. Since the process is linear, this objective will also be linear, starting with the Information Gathering step and once sufficient information is gathered, it will be used to start the Vulnerability Assessment step and so on.

A configurable system can adapt to different situations and different types of users. This objective is also important because it helps broaden the potential users of the system. For example, the system should let the user decide which types of tools use or choose which steps of a penetration test s/he wishes to do. This is important because a penetration testing process can be a slow and long one, and if some step needs to be repeated, it would be a complete loss of time to perform a complete penetration test again. It should also be desirable to be able to differentiate between inside and outside tests depending whether the tester is inside the network s/he wants to test or not, and in the case s/he is outside the target network, it could be interesting to be able to choose between passive and active tools.

Cybercrime is a growing issue and since technology keeps evolving, cyber criminals have to evolve their attacks as well in order to find new vulnerabilities. This new vulnerabilities and the way to exploit them has to be incorporated in the existing security tools or implement new security tools in order to address them. This process is continuous and a penetration tester has to be continually updating his/her knowledge base. This means that our system has to be able to update its knowledge as well. In order to do this, the system needs to be easily expandable. To expand a system like this means to add new tools that are either better than the current ones or that they cover new aspects of the cybersecurity world, like new technologies.

The reports presented by current vulnerability assessment frameworks are usually long documents with all the information gathered by the multiple tools used. This could seem the desired output, and sometimes is, but these documents can be more than 200 pages long, and reading all of it can be extremely boring. This documents usually have multiple parts. One of the important parts is the part where all the vulnerabilities found by the framework are listed, and a solution is proposed for these vulnerabilities. This part is important to the coders of the system under study because it pinpoints where the problems are, and usually offers a good solution to fix them.
part is the summary of the findings, usually directed to the management of the system under study to give a broad vision of the security of the system.

For security experts, the part where the vulnerabilities is listed is usually the vital one, since it tells the tester where to continue looking in order to exploit the system. However, this part is usually the output of all the vulnerability scanners combined, and it makes it really difficult to read and understand quickly. Finding a concrete vulnerability for a concrete service without knowing relevant information like the CVE of the vulnerability can be truly difficult.

For all these reasons is why one of this systems’ objectives is to provide an easy-to-read and easy-to-understand output. In order to comply with this objective, the output of the system will not be a large document with separate parts for separate tools. Instead of this, the system will collect the results of the different tools in a database and will present the results in a unified way.

4.2 Analysis

4.2.1 The tools

The most crucial part of this system will be the tools that it uses. This will be the main strength, since it will gather more information as it integrates more tools.

But, which types of tools does it need?

There are some aspects to be considered when selecting tools to incorporate unto the system. First of all, since the aim of this system is to be as automatic as possible, the tools should be as automatic as possible, this means that they should not require a constant interaction from the user. This is an important aspect, because it discards some of the tools that are currently being used. There exist several frameworks and complex tools that use graphical interfaces to interact with the user continuously. This type of tools or frameworks, even though they are usually the more advanced and powerful, are not recommended for this system. This is so because the interaction with the user wants to be minimal.

This minimal interaction with the user is easily achievable with command-line tools, or external tools that can be launched via the command-line and do not need any interaction. This does not mean that the tools need to be simple ones. For example,
Nmap is a command-line tool and it is a really powerful tool, being able to scan a whole network and discover all its machines and the services residing on the open ports of those machines. Another example would be The Harvester, a command line tool that uses search engines to discover usernames, email addresses and virtual hosts related to a target.

Another interesting aspect about the tools that have to be used is the ability to parse their results. The tools included in the system should preferably have an easy-to-parse output format. Usually, this kind of tools have XML output reporting, which is one of the easiest ways to parse the results into the database. There exist XML parsing libraries in almost all the current programming languages. The parsing of the tools will be one of the key points of the system, since it will be the part where the output of a tool is collected and integrated into the database. When a new tool wants to be added to the system, a parser for this tool has to be created and added to it.

There exist tools in the current market done by security companies that require really high fees to use them. Since this system is part of a Degree’s Final Project, all the tools used on it will be free of charge. However, due to it being expandable, proprietary tools could be added afterwards by the users.

Having all these aspects in mind, the tools chosen for this system are a subset of the optimal selection of tools that this system would need to be complete. Since the development time of this project will be not that long, the focus of the tools used will be on the information gathering step of a penetration test. Focusing on the information gathering step will allow the system to settle an attack surface. This attack surface has to be as complete as possible in order to establish a good starting point for the next steps. And since the information gathered in this first step has a really wide range, it will help complete the data model that will store all the information. This means that even though there are lots of different types of vulnerabilities and different tools to find them, they will always be a vulnerability stored in our system. On the other hand, the information gathering step can find things like machines and ports, but it also can find usernames, email addresses, the services residing on the found ports and even the software used to generate the files published on the Internet. Having this wide range of
information on the system is really important because it gives the security expert lots of possible entry points.

### 4.2.2 The programming language

After the research done in the first stages of this project, it is clear that the system that will be build has no specific programming language constraints. Any programming language that could launch command-line tools, parse the results of those tools and then display information in any way could suffice. However, after analysing the current security tools and frameworks that are being used nowadays, the tendency of those tools and frameworks is to code them in Python. Obviously not all tools are coded in Python, but most security frameworks and lots of open-source security tools are.

After discovering this fact, an analysis of the language has to be done before selecting it, to check if it suits all the needs of the system.

First of all, Python has lots of documentation available online and has a great community behind it. This fact, together with the fast learning curve it has, makes it really easy to learn how to code in Python for people who have never used it before. Obviously, a coding background is needed to learn it fast enough to code a system like this. The fact that Python uses an interpreter also helps a lot, because it lets the coder test everything really quickly.

Besides the learning facts, Python’s aim is to be extremely readable, and this makes it really good for collaborative projects or projects that have to be evaluated.

Python, thanks to its extensive community, has lots of useful libraries to do all kinds of things. After some research, interesting libraries can be found that are really useful for this project.

One of this libraries is called Pexpect. This library lets the user launch bash commands in a controlled way, and even communicate with the bash commands if necessary. It also retrieves the output of the command and presents it as a String. The functionality provided by this library is clearly useful for the system under construction. This functionalities could also be achieved with a Python standard library called os, but the fact that pexpect allows interaction with the commands is a great pro.

Another good point towards Python is that it has several libraries to code and deploy web pages. Libraries such as Django are widely used in the professional world to build
websites. A web interface could be a good way to display results in an easy-to-read, interactive way. However, Django is a bit too complex to suit the purpose of the web interface that could be created to display the results of the system. There exists, however, another library called Flask. This library is in fact a microframework for web development. It has the ability to easily build an API, and then it gives the coder the tools to build templates for the web interface of the API. All this is made in a simplistic way, and deploying the API is also extremely simple. This kind of microframeworks are not recommended for production environments, but for the purpose it has to be used in this project, Flask is one of the best options to build the web interface.

After all the research done, the conclusion is that Python is a really good candidate to code the system with. Also, the libraries found are only a small flavour of the whole lot of libraries that Python has. This two will be really useful by easing the process of building the system, but many more could be used, for example for parsing resources. All this research has made it clear that the system will be coded in Python.

4.2.3 The phases

In this section an analysis of the different phases of a penetration test will be related to the system under construction. The tools that will be used will be listed here, and some explanation about the problems found during the analysis of each phase.

Information Gathering

There are lots and lots of tools that could be used in this stage of a penetration test. There are lots of areas that need to be covered, and lots of information to be found about a target. An important part of this stage is that no previous knowledge on the target is required. This means that all the tools that will be launched in this stage will only need the target’s IP address or name to function.

In this phase lots of information will be discovered and integrated into the database. It will be the stage with more tools in our system and those tools will try to cover most aspects of the target.

First of all, one of the most important tools in penetration testing is Nmap. Obviously, this tool had to be used due to its high usefulness. Nmap is able to discover machines inside a network, and for each machine, it can discover interesting things such as the
operating system of the machine and the opened ports that this machine has. Besides from discovering the ports, it also discovers the service that is listening in each port. All this things make it an important tool to use, since all the information it finds is really useful and establishes the attack vector for next phases. In this system, Nmap will be used to discover the operating system of each machine found in the network, as well as the active ports in them.

Another area that needs to be analysed is the DNS system of the target. In order to find the name servers of the target a pair of tools will be used. First of all, Dnsenum will be used to find name and mail servers of the target. The machines found will be added to the system to be scanned later with Nmap. The other tool that will be used is Dnsrecon. This other tool is used for exactly the same as Dnsenum. While researching, it has happened that this two tools gave different results, and this is the main reason why two really similar tools are being used.

Another aspect that this phase has to cover is discovering subdomains. In order to do that a tool called Dmitry is used. This tool can be used to discover multiple things, like performing a whois lookup, discovering email addresses and subdomains, and performing TCP portscans. For this project, Dmitry will only be used to discover subdomains, this is so because other tools will be added that cover the other aspects of Dmitry.

Search engines can provide lots of information about a target. Usually the information gathered from search engines can be used in Social Engineering attacks. This information can range from subdomains of the target to the software used to produce the publications the target has made available. In this project, two tools will be used to gather information from search engines. The first one is Metagoofil. This tool gathers all the documents published by the target and extracts information from the metadata on those documents. The information contained in the metadata that Metagoofil finds and the system gathers is email addresses and usernames.

The second tool that will be used is called TheHarvester. This tool uses search engines to directly search everything related to the target. It finds subdomain paths that have been crawled by the GoogleBot crawler, and it can also find email addresses from people working on or related in any way with the target.

Web application firewalls are not really well-known pieces of software, but they can help prevent attacks from the outside. They control all input and output of the web server
and block any potential risks. There is a small tool that lets the tester discover if the
target is under a web application firewall, and in the case it is, it also tries to find out
which one is being used. This tools is called **WafW00f**. It will be used in the project
to find out this extra information about the target.

**Vulnerability Assessment**

This penetration testing step also has lots of tools available. Since there are lots of dif-
ferent vulnerability types, there has to be tools to cover them. However, most of these
tools need some kind of previous knowledge on the target. For example, to launch a
Wordpress vulnerability scanner the tester needs to know that the target has a Word-
press web server. Another example would be SQL Injection scanning. This type of
scanning needs a vulnerable link in order to launch all the known attacks. This tools
will only check if the link provided is vulnerable to a series of known attacks. This fact
makes it really difficult to launch this type of tools automatically.

However, there exist general scanners that, for example, scan a web server in order to
find interesting things and known vulnerabilities that globally affect the target. One of
such tools is **Nikto**. This tool will be the main tool used in this phase of the pene-
tration test. It scans web servers to find potentially dangerous files or CGIs, it checks
for outdated versions of the servers and it scans for server configuration items such as
multiple index files or HTTP server options.

When **Nikto** has finished scanning a web server, it presents the vulnerabilities it has
found with a description and a link to the vulnerability on the web server, amongst
others. This information will be stored in the database.

One of the interesting things that **Nikto** gives for each vulnerability is its id inside the
*Open Source Vulnerability Database* (OSVDB). This id can be used to find more infor-
mation about this vulnerability, for example, on the OSVDB website. However, there
is more than one way of classifying vulnerabilities and different entities have different
databases. The most common one is the *Common Vulnerabilities and Exposures* (CVE),
which is maintained by the **MITRE Corporation** and is funded by the **National Cyber
Security Division** of the **United States Department of Homeland Security**. Having this
entity funding it makes this system kind of the standard one. However, since there is
more than one system, it happens that different tools offer different information. In our
case, **Nikto** provides OSVDB ids. The desirable case would be to be able to convert
OSVDB ids to CVE ids, but doing this is more difficult than it seems.
There exist web servers that provide user interfaces to get the conversion from OSVDB to CVE, but there is no way to automate this conversion massively. There exists an API that allows queries to get the conversion, but it only accepts two queries per day, and this would be useless in this system.

In order to use more than one tool in this step, another scanning tool will be used. The tool is called **wpscan** and it is used to scan Wordpress websites. It scans for outdated plugins and themes and can obtain default users. In this system it will be used to scan only outdated plugins that are linked to a known vulnerability. This tool also provides vulnerabilities with the OSVDB system.

**Exploitation**

The number one exploitation framework is **Metasploit**. It provides the whole environment for detecting targets and exploiting them. However, our system aims at being automatic and using metasploit automatically is not that easy.

As it has been stated previously, there is not a clear standard on vulnerabilities. Due to that, there does not exist a clear relation between a concrete vulnerability and an exploit. When gathering information about known vulnerabilities, sometimes the database providing the information provides some known exploits for exploiting it, but this is not common practice. Due to that, intelligent automatic exploitation of a target is extremely difficult. A possible solution could be to launch all known exploits independently on the target, but that would be extremely time consuming. However, there exists a module for **Metasploit** called **db_autopwn**.

**db_autopwn** is a module that feeds from the output of **Nmap** and launches all the exploits it knows accordingly. It can be configured to launch exploits related to port numbers or directly launch all exploits. If an exploit is successful, the module opens a session on the target, meaning that the tester has access to it. This could seem like the perfect solution for this project, but there are some problems. This module is a deprecated one. It was deprecated because it was unstable, it crashed some systems, and it did not fit the framework’s scope. The developers of Metasploit do not encourage the usage of this module, but it’s the only automatic exploitation option currently available. Intelligent exploitation is the one advised by the framework’s developers, and this makes a lot of sense, since launching countless exploits to a target without even knowing if any of them will work is not a smart move. However, for the sake of demonstration, the system will be able to launch the **db_autopwn** module to the target if the tester desires it.
Post Exploitation

Once the pen tester has gained access to a system, s/he still has some work to do. Sometimes the access point is not that interesting and the tester has to jump from one machine to another until s/he finds interesting stuff. This process can also be done with Metasploit. The framework provides modules to jump from machine to machine in a semi interactive way, using console commands.

This process in itself would be really difficult to automate, since it is really environment-dependant, and could be different every time. On top of that, the system should be previously aware of the network inside the target, or have some methods of discovering it once it’s inside. On top of that, the system should be intelligent enough to know which machines would contain sensitive information inside the network. All these things are really complex to automate and would require an adaptable Artificial Intelligence. This is obviously out of the scope of this project.

So, having in mind that it would be really difficult to do, and also noting that the system is probably unable to exploit the target in the first place, this step will not be implemented on the system. It makes no sense to cover this part if intelligent exploitation is not possible.

Reporting

This is the step where the system creates a report of all the findings. As stated before, current system output large pdf files with all the vulnerabilities found and all the information available on them.

One of the objectives of this system was to provide easy-to-understand reports, and a good way to do it is to provide an interactive way of visualizing the results. Since Python has easy ways to code websites, the reporting system will be done through a web page that will interactively present all the findings of the system. This system makes finding information easier and it is much more intuitive than a 200-pages report. The information will be provided on a machine basis, and the tester will be able to select which information s/he wants to see.

All the information gathered by the system will be stored on the database. This fact gives the possibility of outputting a pdf report of all the findings, but it will not be a priority for this project. The web interface is believed to be a much more intuitive and easy-to-understand way of showing the results.
4.3 Design

4.3.1 Data Model

One of the most important parts of this system is the data model, because it is where all the information gathered by it will reside. This data model has to be able to represent machines inside a network, and the vulnerabilities each machine has.

The core of this system’s data model are the audits. An audit is the representation of the penetration testing process. As such, an audit always has a target, the date when it was created, and a pair of optional attributes that keep track of its progress.

As a result of running an audit, the system will find machines related to the target of the audit. This machines are represented as a data class as well. Each machine has an address and a name as primary identifiers. Different information about the machines in the target is collected, and this information has to be stored in the database as well. This information includes the Operating System of that machine and its associated cpe and if it has a Web Application Firewall or not. Each machine is scanned in order to find information about the ports it has in use, and the services listening to those ports.

For each port some information is stored. This information contains the number of the port inside the machine, the service it provides and its state, and the product that is giving the service, together with its cpe.
Once the ports have been identified, the system will search for vulnerabilities in those ports. For that, each port has associated a number of vulnerabilities. Each vulnerability has a name, a description, a link to the vulnerability and the tool that has found it. Additionally, they have the ids corresponding to the OSVDB and CVE systems, if the tool provides them.

The tools that the system uses can discover usernames and email addresses regarding to a target. This information has to be stored in the database as well. In order to store this information, there exists a class named `employee`. This class does not necessarily represent an employee of the target’s company, it is merely a container for either a username or an email address. In the case a tool is able to extract both the username and the email address of the same person, it could be stored together as well.
In order to control the audits, there exist two different classes. The first class is the *User*. This class represents a user of the system, and as such, it identifies it with a username and a password. It also asks for an email address, but it has no use yet. This class controls who creates Audits.

There could be the case where a tester wants to perform the same audit more than once. In this case, it could be helpful to have all the audits that are directed to the same target together. To provide this scenario, there exists a class in the system called *Network*. A network is identified by a company name. This company name represents the target that wants to be tested. Then, the network comprises all the Audits that are performed towards the same target.
store the tools the command line that calls them has to be saved, along with some options of each tool. For each tool some attributes are saved as well, to get some extra information. This extra information could provide some customization to the system, for example to call only passive tools. The phases of the penetration testing process will also be stored, in order to have each action linked to a phase. This allows to quickly select all the tools that a phase needs to complete.

![Table](image)

**Figure 4.6: Phase and Action**

In order to manage the data model, *Object-Relational Mapping* is used. This is achieved by the usage of a *Python’s* module called *peewee*. This module is a small and expressive ORM that includes support for *sqlite*, *mysql* and *postgres* databases. This module makes managing the database really easier, and allows for complex queries in really simple lines of codes. As this is a small and simple data model, it would be tempting to use a simple database engine like *sqlite*, however, since the aim of this project is to build an expandable system, the database engine chosen for the system is *MySQL*. *MySQL* is a powerful and well-known database engine which can be used in conjunction with *peewee* in order to make it simpler.

This is the complete data model for the system. However, *esCERT UPC* has provided the database of one of its tools, called *Altair*. This database contains information about known vulnerabilities, amongst others, and will be used to find vulnerabilities related to certain CPEs. This will provide some extra value to the system, getting information related to systems that is not gathered by any analysis tool.
4.3.2 Tool Integration

The only thing that all the tools that are used by the system have in common is that they all are command line tools. This could suppose a problem in the sense that a custom class should be created for each tool that has to be used. Each tool would have its way of launching it, with different options, and different output styles. This could mean lots of work if someone would want to expand the number of tools in the system. Instead of doing that, a simpler solution will be taken. In this solution, there exists a generic class able to call any command line tool with root privileges, and a collection of parsers, here, of course, one per tool, that would parse the output of the tools. With this system, calling any tool is really simple, and the only thing that has to worry the user is adding the parsing function to the collection. Since the parsing function had to be created anyway with the other model, this model does not add additional complexity.

This model of tool integration has two differentiate parts. The class that launches the tools, which will be called Tool, for obvious reasons, and the collection of parser functions, which will be stored in a file, together with a simple function that will retrieve each parsing function with the only need of the name of the tool it parses.

Figure 4.7: Complete Data Model
The class Tool has some attributes, which correspond to the information stored in the database for each tool:

- The name of the tool
- The options that the tool uses
- The target of the tool
- Whether the tool generates a file output or not

The only information that is not stored on the database is the target, which is set by the user at the start of the audit.

In almost all tools used in this project, the target is specified at the end of the command line. In the event that a tool cannot specify the target in the last place, but has to specify it in the middle of the options, a macro has been created. The class will search for the existence of the keyword %target% and if it exists, it will be replaced by the target of the tool. If it does not exist, it will be appended at the end. There also exists a second macro with the keyword %file% to specify the name of the output file. This macro has little use because it can be specified when the tool is stored in the database, but it has been created for special cases.

Once the Tool class has been instantiated with all the correct values and the target has been fixed, a function called run() can be used to execute the tool. This function uses the pexpect module to spawn a child with the execution command of the tool. It also calls the tools with root privileges because certain tools need root privileges to be used. When the tool has finished its execution, the standard output of the tool is stored in an attribute of the class. This output can be then requested if necessary. Since most of the tools used can generate a file output in a parsable format, this attribute is not always used.

When the tool finishes its execution, the parser function of the tool has to be retrieved and used to obtain the information this tool has gathered. The parsing functions will be retrieved by a function that uses the name of the tool as a key to the function that parses it. As all the functions are retrieved in the same way, they all need two obligatory parameters, the resource with the file or the string containing the output of the tool, and the audit where the tool execution belongs to. The parsing functions can have additional parameters but they need to have a default value.
Each parsing function is the one in charge of storing in the database the information gathered by the tool. All functions will be different and depending on the output format, they will use different strategies. The most common and simple is parsing an XML file with the results of the tool. This makes it really easy to obtain the information. It can happen, however, that the output XML file is malformed. When this happens, normal parsers usually stop working and report an incorrect XML format. To avoid this, another Python module has to be used. This module is called **Beautiful soup**, and it gets all the data out of malformed XML files. It saves hours and hours of XML fixing and it has been a great discovery.

Since each function will be different and will be related to the tool it parses, it makes no sense designing them one by one. Everything is set up in order to make it easy to incorporate new parsing functions into the system’s code. After pasting the new parsing function at the end of the file, the name of the function has to be linked to the name of the tool in the dictionary function that opens the file.

### 4.3.3 Monitor

The next step in the design of the system is the step where the audits are launched. In order to do that, the system needs to gather the different tools from the database and then use the **Tool** class to launch them. When each tool has finished, the parsing function of that tool has to be used to parse the results and incorporate them into the database. The part of the system that searches for the tools and overlooks their executions is called **Monitor**.

The **Monitor** has several functions to perform each step of the penetration testing process. These functions will first gather all the tools that are assigned to the current phase, and then will populate a **Tool** instance with each one of them. Once the **Tool** instance has been filled, it is launched. When the tool finishes its execution, the function has to obtain the parsing function of the tool and use it with the output of the tool. This output could simply be the string that the **pexpect** module returns or the **XML** file generated by the tool.

When the tool has finished its execution and its output has been parsed, the **monitor** has to update the progress of the audit.

Having a function for each step helps in customizing how the step has to be. For example, in the **information gathering** phase the tools find information about machines on
the system. This machines have to be analysed as well in the same phase. The design of the information gathering phase is the following. First, all the tools are run with the target of the audit as target. Once all tools have been run, the database is filled with new machines. This machines have to be analysed again with some of the tools that have already been used. The best example of such a tool is Nmap, because it will help discover the operating system of each machine and the open ports and services residing on them.

The execution time of certain security tools can be really long. That is why the monitor tasks have been decoupled from the part of the system which calls them. This way, when a user runs an audit, s/he does not need to wait until the audit has finished to continue exploring the system or preparing a new audit. This has been accomplished with Celery.

Celery is an asynchronous task queue based on distributed message passing. This task queue uses a broker to pass messages between the tasks it launches. It supports multiple brokers, but the recommended one and the used on this system is RabbitMQ.

Celery is a really easy system to use and set up, and it does not add any coding complexity to the system. It is really easy to use it and calling it from the web interface is like calling a normal function. Since the results of each phase of the penetration testing process are stored in the database, the system does not even need to control the returning results.

4.3.4 Web Interface

At first, the web interface was intended to only show the results of the system, by interactively showing the contents of the database. Since certain Python libraries were found which made writing web interfaces really easy, the scope of the web interface changed a little bit.

Now, the web interface controls all the system, not only providing the results in an interactive way, but allowing the user to create and run audits from it. This comes in handy in the sense that managing users, networks and audits is really easier with a web interface than with command line tools.

As has been said before, the web interface will be created with a Python module called Flask, which is a microframework for web development, which incorporates Jinja 2 as
a templating engine. *Flask* is also really easy to set up and use, and the template engine is intuitive and simple.

The web interface starts with a log in screen, with the option of registering new users. Once the user has logged in, a list with all the networks present on the system is shown. It also gives the opportunity to create new networks. When a network is selected the audits performed in that network are listed, and the option to create new ones is present as well. The user can create new audits or display/redo old ones. When an audit is selected, the user can choose to run it, be it as a whole or phase by phase, or to display its findings.

The results of the audit are separated in two main blocks, the first block contains all the user information found, the usernames and email addresses. The second block is the important one, which shows all the machines found in the system. The information gathered from the machines will be present as well. If any known vulnerabilities exist that are related to the CPE of that machine, a link will be provided with information from the Altair database. When a machine is selected, all the ports found by *Nmap* are displayed. If any of those ports has vulnerabilities assigned to it, a link will be available. Following that link will take the user to the vulnerabilities discovered from a certain port. Each vulnerability will be listed and a link to the official site will be provided if possible. This links will redirect to *OSVDB* or *CVE* vulnerability system’s websites.

## 4.4 Implementation

In this section the details of the implementation of the system will be listed, along with any implementation decisions and assumptions made.

### 4.4.1 Data model

As it has been said in the design section, to control the system’s database the *Python*’s module *peewee* is used. All the code below is stored in a python file named *db.py*. The first thing needed to work with peewee is to create the connection with the database. In order to do this, the database needs to be previously created, and a *MySQLDatabase*
object needs to be instantiated.

```python
db = MySQLDatabase('PTAS', user='ptas')
```

Once the creation of the database has been done, **peewee** needs to create models for each table on the database. Each model will need the database assignation inside it as a *Meta* class. This is why it is recommended to create a *Base model* first, and then use it with all the other models. The only purpose of this *Base model* is to link the database to all the models.

```python
class BaseModel(peewee.Model):
    class Meta:
        database = db
```

Once this base model has been created, all the other models can be created inheriting the *BaseModel*. Each model will be a *Python* class and a *MySQL* table in the database. *Peewee* makes it trivial to create the classes and it lets the coder easily establish foreign key relations. A good example to show all this is the *Audit* class, which has some attributes and two foreign key relations.

```python
class Audit(BaseModel):
    created = peewee.DateTimeField()
    target = peewee.CharField()
    current_phase = peewee.CharField(null=True)
    progress = peewee.FloatField(null=True)
    user = peewee.ForeignKeyField(User, related_name='audits')
    network = peewee.ForeignKeyField(Network, related_name='audits')
```

As it can be seen in the code snippet, **peewee** lets the coder put a name to the inverse relation of the foreign key. This means that from the other classes, in this case from *user* and *network*, it will be really easy to find the related audits.

*Peewee* lets the coder not think in terms of *MySQL* queries, but instead in terms of *Python* classes directly.

When all the models have been created, *Peewee* needs to create the tables on the database. In order to do that, it provides a built-in function that automatically creates the table. In order to ease the process, a simple function that creates all the tables
is provided. This function should only be used once, when setting up the system for the first time.

```python
def create_tables():
    User.create_table()
    Network.create_table()
    Audit.create_table()
    Employee.create_table()
    Machine.create_table()
    Port.create_table()
    Vulnerability.create_table()
    Phase.create_table()
    Action.create_table()
```

The tools that have to be used by the system have to be stored on the database. To do that, the tools are provided in text files. These text files, one per phase, contain the name of the tool, the options used by the tool, whether the tool generates a file output or not and whether the tool is active or passive, all separated by commas. There exists a function that reads all the tools from these files and loads them into the database. This makes adding new tools easier, since the only thing needed to do is adding the tool to the correct text file with the specified format and then running the `load_actions()` function.

```python
def load_actions():
    try:
        ig = Phase.get(Phase.order == 1)
    except Phase.DoesNotExist:
        ig = Phase(name='Information Gathering', order=1)
        ig.save()
    try:
        va = Phase.get(Phase.order == 2)
    except Phase.DoesNotExist:
        va = Phase(name='Vulnerability Assessment', order=2)
        va.save()
    try:
        ex = Phase.get(Phase.order == 3)
    except Phase.DoesNotExist:
        ex = Phase(name='Exploitation', order=3)
        ex.save()
    load_file(ig, 'IGactions.back')
    load_file(va, 'VAactions.back')
    load_file(ex, 'EXactions.back')

def load_file(phase, filename):
    f = open(filename).readlines()
```
Chapter 4. Design & Implementation

for line in f:
    l = line.split(',')
    try:
        a = Action.get(Action.tool == l[0])
    except Action.DoesNotExist:
        a = Action(tool=l[0], options=l[1], fileOutput=l[2], active=l[3], phase=phase)
    a.save()

With this, all the necessary steps towards the construction of the data model are done. The only thing that still needs to be done is importing the database dump provided by es-CERT. The first thing needed to be done is creating the database using the provided dump file. Once the database has been created, a peewee model has to be created. Luckily, peewee comes with a little script called pwiz. This script reads an existing database and creates the skeleton code that it needs to interact with the said database. Since the Altair database will be read-only, with the skeleton generated by pwiz will be enough to work with it.

4.4.2 Tool Integration

As it has been said in the design section, a generic class that will be able to run all the tools has to be created. This class is called Tool and will be stored in the file tool.py. This class uses the Python module pexpect, and has some simple functions and the function that runs the command line tool.

```python
import pexpect

class Tool:
    name = ""
    options = ""
    target = ""
    output = ""
    fileOutput = False

    def run(self, timeout=20000):
        if len(self.options) > 0:
            launch = "sudo " + self.name + " " + self.options + " " + self.target
        else:
            launch = "sudo " + self.name + " " + self.target
        log = open('log.txt', 'w')
        child = pexpect.spawn(launch, timeout=20000, maxread=20000, logfile=log)
        try:
            child.expect("password for ptas:", timeout = 5)
```
child.sendline("ptas")
child.expect(pexpect.EOF)
except pexpect.EOF:
    pass
self.output = child.before

def setTarget(self, tar):
    if self.options.find('%target%'):
        self.options = self.options.replace('%target%', tar)
    else:
        self.target = tar

def setFile(self, f):
    if self.options.find('%file%'):
        self.options = self.options.replace('%file%', f)

def getResult(self):
    return self.output

The simple functions setTarget and setFile search for the keywords and then replace them with the received parameters. In the case of the target, if the keyword does not exist, it appends the target at the end.

The more complex function is run, which first appends the name of the tool with the options and the target and then runs it using the spawn function. This function creates a child with whom it is possible to communicate. Since the launch string has the sudo keyword in order to obtain root privileges, the function has to provide the password when it is asked for. In order to do this the functions expect and send are used. The first one waits until the standard output of the launched process matches the string passed as argument, and the second one send the argument to the process as standard input. Finally, the function has to wait until the process finishes. This is done with the same expect function, but this time expecting the end of file special character. Once the process has finished, the output is stored in a parameter of the class.

The timeout established in the run function has to be long enough to wait for slow processes. For example, the tool named theHarvester, spends a lot of time running its normal execution, and the default timeout for the spawn function would be not enough.

The other main part of tool integration are the parsing functions. Each function is unique and aimed at a specific tool. All the parsing functions are stored in a file called
This file also has a function that given the name of a tool returns the parser of that tool.

```python
def getParser(name):
    return {
        'dnsenum': parseDNSENUM,
        'nmap': parseNMAP,
        'dnsrecon': parseDNSRECON,
        'dmitry': parseDMITRY,
        'theHarvester': parseHARVESTER,
        'xprobe2': parseXPROBE,
        'wafw00f': parseWAFW0OF,
        'nikto': parseNIKTO,
        'metagoofil': parseMETAGOOFIL,
        'wpscan': parseWPSCAN
    }.get(name, None)
```

This function needs to be update each time a new parsing function is added to the file. All functions are unique, but not all functions will be explained here, since most of them are similar. Some examples will be described.

One of the best examples of an XML parsing function is the function that parses the tool **Nmap**. It needs the `ElementTree` standard library, which is imported with the following line.

```python
import xml.etree.ElementTree as et
```

Since the parsing function of the **Nmap** tool is quite long, it will be explained by parts. First of all, this function can accept an extra parameter, which is a **machine**. This means that this function can be called with a machine that is already present on the system, in which case the first part will be ignored. The first part is where **Nmap** identifies the machine, as well as its operating system and its cpe.

```python
def parseNMAP(resource, audit, machine=None):
    tree = et.parse(resource)
    root = tree.getroot()
    for host in root.findall('host'):
        if machine == None:
            address = host.find('address').get('addr')
            try:
                os = host.find('os')
                match = os.find('osmatch')
                name = match.get('name')
                cpe = match.find('osclass').find('cpe').text
            except Exception, e:
As it can be seen in the code above, all the function is wrapped inside a for loop. This is so because Nmap can scan an entire network and find multiple machines on it. For each machine Nmap finds, the function gets the address, the operating system and the cpe. Then it looks into the database in order to find out if the machine already exists on the system, and if it exists, the function updates the machine with the operating system and the cpe. If the machine does not exists, the function creates the machine with all the information it has gathered. If a machine has been provided, it simply assigns the provided machine to the m variable, which will be used in the second part of the function.

The second part of the function is still inside the for loop.

```python
ports = host.find('ports')
for port in ports.findall('port'):
    number = port.get('portid')
    service = port.find('service').get('name')
    state = port.find('state').get('state')
    try:
        product = port.find('service').get('product')
        cpe = port.find('cpe')
    except Exception, e:
        product = None
        cpe = None
    try:
        p = db.Port.get((db.Port.machine == m) & (db.Port.number==number))
    except db.Port.DoesNotExist:
        p = db.Port(machine=m, number=number, service=service, state=state, product=product, cpe=cpe)
        p.save()
```
In this second part, the function looks for all the ports found by *Nmap* in the host that is currently analysing. For each port, if finds its number, the service that was listening to that port and its state. Then, if it exists, it gets the product fingerprinted by *Nmap* and its cpe. Once it has all the information, it searches the database for that exact port, and if it does not exist, proceeds to create it.

It can happen that some tools provide as output an HTML file that is not correctly well formed. In this case, using the *Python* standard parser would not be possible. If the *ElementTree* parser gets a malformed XML, it raises an exception. In order to avoid that, a *Python* module named *Beautiful Soup* can be used. This module accepts malformed XML files and tries to get all the information it can from them. In the system, this happened with two tools, *Metagoofil* and *theHarvester*.

The parsing function of the tool *theHarvester* is the following one.

```python
def parseHARVESTER(resource, audit):
    f = open(resource)
soup = BeautifulSoup(f)
    for tag in soup.find_all('ul '):
        print tag.get('class ')[0]
        if tag['class'][0] == "userslist":
            for elem in tag.find_all('li '):
                try :
                    e = db.Employee.get((db.Employee.email == elem.string) &
                        (db.Employee.audit == audit))
                    except db.Employee.DoesNotExist :
                        e = db.Employee(email = elem.string, audit = audit)
                        e.save()
                    print elem.string
                if tag['class'][0] == "pathslist":
                    for elem in tag.find_all('li '):
                        aux = elem.string
                        aux = aux.split(':')
                        address = aux[0]
                        name = aux[1]
                        try :
                            m = db.Machine.get((db.Machine.address == address) &
                                (db.Machine.name == name) &
                                (db.Machine.audit == audit))
                            except db.Machine.DoesNotExist :
                                m = db.Machine(address=address, name=name, audit=audit)
                                m.save()
```

As it can be seen in the function, using the *Beautiful soup* module is really simple. First of all, it reads the file with its custom parser and return a tree-like variable than can be
easily queried. In this concrete example we see that the function gets email addresses and creates Employees, and then gets virtual hosts and creates Machines with them.

The third possibility is that a tool does not provide a file output or that it provides a file output but not in a parsable format. In this case, the parsing function has to play with Strings and with the find() function provided by Python. A simple example is the tool called Wafw00f, which does not provide a file output. The parsing function is the following one.

```python
def parseWAFW00F(resource, audit, machine=None):
    aux = resource.find('Checking')
    output = resource[aux:]
    tar_end = output.find('
')
    target = output[9:tar_end]
    result = 'No WAF'
    lines = output.split('
')
    for line in lines:
        if line.find('The site ') >= 0:
            l = len(target) + 9
            result = line[l:]
        if machine != None:
            machine.waf = result
            machine.save()
    print result
```

This type of outputs require some creative string manipulation in order to obtain the part of the output that can be stored in the database. The usage of the find() function is a must, and playing with String delimiters also helps in obtaining the interesting bits.

This three types of output cover all the different outputs found in the tools used by the system. If a future user wants to add a new tool, a new parsing function has to be created, that correctly parses the output of the desired tool and transforms it into information in the system’s data model.

### 4.4.3 Monitor

In order to launch tools, the Monitor needs to retrieve those tools from the database and create Tool classes for each of them. This work is done in a file called monitor.py. This file has the function that will be launched when an audit starts. This function will be launched with the asynchronous task queue manager called Celery. This is why the
first thing needed is to import the module that controls it, and then to create a Celery instance with the desired broker. This is done with the following lines of code.

```python
from celery import Celery

app = Celery('tasks', broker='amqp://guest@localhost//')
```

The only thing needed now for Celery to work is to register the functions we want to be called with this system. This is done by simply adding `@app.task` before the definition of the functions we want to register. In this case, for simplicity, only one function will be registered. This function will receive a list which will tell it which phases have to be executed. This function is the following one.

```python
@app.task
def launch(audit_id, l):
    if "1" in l:
        runIG(audit_id)
    if "2" in l:
        runVA(audit_id)
    if "3" in l:
        runEX(audit_id)
    audit = db.Audit.get(db.Audit.id == audit_id)
    audit.current_phase = None
    audit.save()
```

As it can be seen, it receives a list and then launches the appropriate functions that represent each phase of the penetration testing process.

The first function to be explained will be the function that represents the Information Gathering phase. This function is a long one because it first scans all the tools and then scans some of them again targeting the new machines found previously. In order to make it more understandable, an auxiliary function has been created that gets all the parameters of an Action from the database and assigns them to a Tool instance and then runs it. The function is the following.

```python
def run(action, audit):
    t = tool.Tool()
    t.name = action.tool
    t.options = action.options
    t.setTarget(audit.target)
    t.fileOutput = action.fileOutput
    parser = parsers.getParser(t.name)
    fileName = "%s.xml" % t.name
```
try:
    t.run()
    if t.fileOutput:
        parser(fileName, audit)
        os.remove(fileName)
    else:
        parser(t.getResult(), audit)
except Exception, e:
    pass
print t.name, 'done'

This auxiliary function runs the Tool instance and then searches for the parser and uses it to parse the output of the function. If the function has generated a file output it tries to remove it after it has been parsed.

The Information Gathering function has two differentiated parts. In the first one it runs all the available tools assigning as target the target of the audit. This first part is the one that uses the auxiliary function described above.

def runIG(audit_id):
    audit = db.Audit.get(db.Audit.id == audit_id)
    q = db.Action.select().join(db.Phase).where(db.Phase.order == 1)
    audit.current_phase = 'Information Gathering'
    audit.progress = 0
    total = q.count()
    cont = 0
    audit.save()
    for a in q:
        run(a, audit)
        cont+=1
        p = float(cont) / total
        audit.progress = p
        audit.save()
    print 'second step'

As it can be seen above, the progress is separated for the two parts of the function since it is impossible to know beforehand how many machines will the process find. Apart from this clarification, the rest of this part of the function is trivial, retrieving all the Actions from the database and running them one by one.

The second part is longer because the auxiliary function cannot be used as is.
In this part of the function the first thing needed to be done is retrieving all the machines that the process has found. After getting them, the three tools that will be used have to be gathered as well. It can be noted that the progress total is updated with the number of machines times 3.

After getting the machines and the tools, each tool has to be executed for each machine. In order to do that, a Tool instance is created for each of them, and then executed. When it has been completed, the parser is used with the difference that now the machine that the tools are analysing is passed as a parameter. After each parser finishes, the progress is updated.

The second phase of the penetration testing process is the Vulnerability Assessment phase. In this phase, the system is checked for vulnerabilities on its services. In this
project, only HTTP vulnerability scanners have been used. This means that the function will have to run all the tools through all the ports that have HTTP as a service.

```python
def runVA(audit_id):
    audit = db.Audit.get(db.Audit.id == audit_id)
    q = db.Action.select().join(db.Phase).where(db.Phase.order == 2)
    audit.current_phase = 'Vulnerability Assessment'
    audit.progress = 0
    cont = 0
    audit.save()
    for port in p:
        for action in q:
            cont += 1
            addr = port.machine.address
            t = tool.Tool()
            t.name = action.tool
            t.options = action.options
            if action.options.find('% port %'):
                t.options = t.options.replace('% port %', str(port.number))
            t.setTarget(addr)
            t.fileOutput = action.fileOutput
            fileName = '%s.xml' % t.name
            if t.name == 'nikto':
                fileName = '%(name)s%(address)s%(port)s.xml' % {'name': t.name, 'address': addr, 'port': port.number}
                t.setFile(fileName)
            print t.options
            parser = parsers.getParser(t.name)
            try:
                t.run()
                if t.fileOutput:
                    parser(fileName, port)
                    os.remove(fileName)
                else:
                    parser(t.getResult(), port)
            except Exception, e:
                pass
            print t.name, 'done'
            pro = float(cont) / p.count()
            audit.progress = pro
            audit.save()
```

This function is really similar to the other one, with the difference that now the targets are the ports found in the previous phase. Here only the ports that run HTTP servers are selected, but if tools that scan other services are added, other services can be selected as well. For the rest of the function there is no big differences. A small clarification
would be necessary for the `fileName` construction. It is purely for debugging purposes and it was not changed.

The third phase that the system runs is the *Exploitation* phase. In this phase only the Metasploit console is used. This is why the following function is so simple. It also uses the `run()` function described previously.

```python
def runEX(audit_id):
    audit = db.Audit.get(db.Audit.id == audit_id)
    q = db.Action.get(db.Action.tool == 'msfconsole')
    audit.current_phase = 'Exploitation'
    audit.progress = 0
    cont = 0
    audit.save()
    run(q, audit)
    cont+=1
    p = 1
    audit.progress = p
    audit.save()
```

With this function the *Monitor* ends. All the tools present on the system are used and parsed. In the case more tools are added to the system, the *Monitor* could not need to be updated, since it gets the tools from the database as a whole.

### 4.4.4 Web Interface

The web interface is the visible part of the system. As it has been said before, the web interface will not only show the results, but will also control the system.

In order to do that, an API has been created with *Flask* and stored in a file named `web.py`, and then a series of templates have been coded with the *Jinja 2* template engine. The API will be detailed now, as well as the results of the templates.

First of all, a *Flask* app has to be created. This app will be used to specify the routes that the API will accept. Also, this app is the one that has to be run in order to start the web server.

```python
app = Flask(__name__)
app.secret_key = 'some_secret'
...
if __name__ == '__main__':
    app.run(host='0.0.0.0')
```
Those four lines of code are the base for constructing the web server. The first two lines are at the beginning of the file and the last are at the end.

So, one of the first things the system needs is user management. Since this system is not intended to be deployed on the Internet, the user management is really simple.

First, we have an API function to register users. This function is called `register()`.

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST' and request.form['username']:
        username = request.form['username']
        password = request.form['password']
        try:
            u = db.User.get(username=username)
            flash('Username already in use')
        except db.User.DoesNotExist:
            user = db.User(username=username, password=password, email=request.form['email'])
            user.save()
            auth_user(user)
            return redirect('/')
    return render_template('register.html')
```

This brief function shows how Flask works. The line above the function declaration states the route needed to follow to get to the register function. It also states the methods allowed when asking for this resource. The first line inside the function is the one that separates the behaviour depending on the method used to access it. If the method is GET, then the function’s sole objective is to render the HTML page with the registration form. If the method is POST, it means that the registration form has been filled, and that a new user has to be created. Here, the function asks for the username and the password chosen. If there is another user with the same username, it flashes an error message saying that the name chosen is already in use. If a user does not exist, it is created, authenticated and redirected to the homepage.

If a user already has a username and password, s/he can login into the system using the login page. The login page is linked to the login function, which is really similar to the register one. The function is the following.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST' and request.form['username']:
        try:
            user = db.User.get(
                username=request.form['username'],
```
password = request.form[‘password’]
)
except db.User.DoesNotExist:
    flash(‘The password entered is incorrect’)
else:
    auth_user(user)
    return redirect(‘/’)

return render_template(‘login.html’)

The main difference here is that if the user does not exist an incorrect password message is flashed. When the user is checked, the function auth_user() authenticates him/her. This function creates a session with information about the user. All the functionalities of the system require a logged user in order to work. This is why a function has been created in order to check whether a user is logged in or not. This function will then be used whenever a user is required to be logged in.

def auth_user(user):
    session[‘logged_in’] = True
    session[‘userID’] = user.id
    session[‘username’] = user.username
    flash(‘You are logged in as %s’ % (user.username))

def login_required(f):
    @wraps(f)
    def inner(*args, **kwargs):
        if not session.get(‘logged_in’):
            return redirect(url_for(‘login’))
When a user is logged in, the first thing s/he sees is the Network list. This screen shows all the networks present on the database, and allows the user to create new ones. In the case no networks are present on the system, it automatically redirects to the screen where the user can create a new one.

```python
@app.route("/networks")
@login_required
def networks():
    networks = db.Network.select()
    if networks.count() == 0:
        return redirect(url_for('createNetwork'))
    else:
        return object_list("network_list.html", networks, "network_list")
```

When the user selects a network, the systems shows the name of the audit and the user that created it, and offers two links, one to create an audit and another one to show all the audits of that network.

When the user chooses the See Audits button, the system shows the list of audits that have been performed in that network. The function that renders the list of audits is almost exactly the same as the networks one.

When creating an audit, the target that the user wants to attack has to be specified. The
following function is the one that covers this functionality. The function that creates a Network is almost the same.

```python
@app.route("/createAudit/<int:network_id>", methods=['GET','POST'])
@login_required
def create_audit(network_id):
    if request.method == 'POST' and request.form['target']:
        try:
            net = get_object_or_404(db.Network, id=request.form['network'])
            user = db.User.get(db.User.id == session['userID'])
            aud = db.Audit.create(
                target=request.form['target'],
                created=datetime.datetime.now(),
                current_phase=None,
                progress=0,
                user = user,
                network = net
            )
            aud.save()
            return redirect(url_for('audit_detail', audit_id=aud.id))
        except Exception, e:
            raise e
    net = get_object_or_404(db.Network, id=network_id)
    return render_template('create_audit.html', network=net)
```

It can be seen in this function that the route now accepts a parameter. The coder can specify the type of the parameter for easier usage. This parameter then needs to be specified in the function declaration as well. This function also has the method
separation in order to render the form and to accept its result. When accepting the result of the form via POST method, the first thing it needs to do is to get the network and the user. If the network does not exist, a 404 message is displayed using the `get_object_or_404()` function. When all the information is gathered, an audit is created and saved into the database. It finishes by redirecting the user to the audit detail page.

![Audit detail before launching](image)

**Figure 4.11:** Audit detail before launching

The audit detail page is where the system launches the penetration testing process. This page is rendered by a function that again accepts both GET and POST methods.

```python
@app.route("/audit/<int:audit_id>", methods=['GET', 'POST'])
@login_required
def audit_detail(audit_id):
    if request.method == 'POST':
        try:
            x = request.form.getlist('phase')
            monitor.launch.delay(audit_id, x)
            return redirect(url_for("audit_detail", audit_id=audit_id))
        except Exception, e:
            raise e
    audit = get_object_or_404(db.Audit, id=audit_id)
    return render_template("audit_detail.html", audit=audit)
```

In this function, the system only receives a list from the form. This list represents the checkboxes displayed on the audit detail page. This same list is the one that has to be passed to the `launch()` function in the monitor. This function call is done using *Celery*. In order to do that, a *Celery* function called `delay()` is used. This function puts the task into the asynchronous task queue. Since this system is asynchronous, the function does
not need to wait for the function to end, and can continue by redirecting the user to the same page. This could seem a little bit idiotic, but when the penetration testing is started, the audit detail screen changes to show the progress of the audit.

When the penetration testing process has finished, the screen goes back to its original state, and the tester can check if any results have been added to the database. The first thing the tester can check is if any username or email address has been found. This is done by a simple function that gathers all the employees related to the current audit.

![Employee list](image)

**Figure 4.13: Employee list**
The more interesting part of the results is the list of machines found by the penetration testing process. This page shows all the machines found by the system, and if the user clicks any machine, its characteristics and the ports found on it are displayed in a collapsible format. The function that renders this page is a really simple one. The interesting thing here is that `peewee` is able to traverse the database using the relations between tables. This means that only providing the machines to the template is enough to render both the machines and their ports. The template of this page will be shown as a good example of both the templating engine and this interesting `peewee` feature.

```plaintext
{% extends "layout.html" %}
{% block body %}

<h2>Machines</h2>

<ul>
{% for machine in machine_list %}
<li class="panel-group" id="accordion">
  <div class="panel panel-default">
    <div class="panel-heading">
      <h4 class="panel-title">
        <a data-toggle="collapse" data-parent="#accordion" href="#collapse{{ machine.id }}"><% print machine.name " ( " machine.address ") %></a>
      </h4>
    </div>
    <div id="collapse{{ machine.id }}" class="panel-collapsed collapse">
      <div class="panel-body">
        <div>
          Name: {{ machine.name }}</div>
        <div>
          Address: {{ machine.address }}</div>
        <div>
          OS: {{ machine.os }}</div>
        <div>Behind a Web Application Firewall?: {{ machine.waf }}</div>
        <div>CPE: {{ machine.cpe }}</div>
        <a href="{%url 'cpe_vuls', cpe=machine.cpe%}">Possible vulnerabilities</a>
        <br/>
        <div class="panel panel-default">
          <table class="table">
            <tr>
              <th>Port number</th>
              <th>State</th>
              <th>Service</th>
              <th>Product</th>
            </tr>
            {% for port in machine.ports %}
            <tr>
              {% if port.vulnerabilities.count() > 0 %}
              <td><a href="{%url 'port_detail', port_id=port.id%}%"></a>{{ port.number }}</td>
              {% endif %}
              <td>{{ port.state }}</td>
              <td>{{ port.service }}</td>
              {% if port.product %}
              <td>{{ port.product }}</td>
              {% else %}
              <td>Unknown</td>
              {% endif %}
            </tr>
            {% endfor %}
          </table>
        </div>
      </div>
    </div>
  </div>
{% endfor %}
</ul>
{% endblock %}
```
This HTML file starts by extending another one, called layout.html, that establishes a base for all the other files on the system, creating a unified view on all of them.

The first interesting thing that can be seen is the usage of a for loop inside the HTML code. Jinja uses the keywords {%%} to execute Python code inside HTML files. In this concrete case this is used to iterate over the list of machines passed by the machines() function defined in web.py.

Inside each machine, after the collapse feature code, the operating system, the web application framework and cpe of that machine can be seen. If the machine has a cpe that is present in the Altair database, a link to the known vulnerabilities will be provided. After this information, a table with all the ports discovered is presented. This can be done thanks to the peewee feature that was commented before. This feature allows to iterate over the ports of each machine with this simple line of code.

Inside the port loop, all the characteristics of the port are listed in a table layout. If a port has been scanned for vulnerabilities and some vulnerabilities have been found, a link is provided in order to inspect this vulnerabilities.

This HTML file is one of the more complete and complex ones. This shows how powerful and simple the Jinja template engine is in conjunction with Flask.

When a user selects a port with vulnerabilities, the port detail page is showed. This page shows all the vulnerabilities found by the penetration testing process in a table layout. For each vulnerability the OSVDB id is provided, as well as a brief description and the link provided by the vulnerability scanner. If the vulnerability is stored in the Altair database, a link is provided that leads to the information of that vulnerability in the Altair database. The function that provides this functionality is the following one.
Lastly, it has been said before that if Altair has vulnerabilities related to a cpe, a link will be provided in the machine detailed information. This link will lead to a table with all the known vulnerabilities, sorted by risk, and with some description and solutions about each one of them. This functionality is provided by the following piece of code.

```python
@app.route("/machine/vul/<path:cpe>")
@login_required
def cpe_vuls(cpe):
    if cpe != 'unknown':
        aux = "%" + cpe + "%"
        vuls = altair.Vulnera.select().where((altair.Vulnera.plat_afectadas + aux)).order_by(altair.Vulnera.riesgo == 'Muy Alto').count()
```

@app.route("/port/<int:port_id>")
@login_required
def port_detail(port_id):
    port = get_object_or_404(db.Port, id=port_id)
    vuls = port.vulnerabilities
    for vul in vuls:
        try:
            aux = altair.Osvdb.get(altair.Osvdb.code == vul.osvdb)
            vul.os = aux
        except altair.Osvdb.DoesNotExist:
            pass
        except Exception, e:
            raise e
    return render_template("port_detail.html", port=port, vulnerabilities=vuls)
It can be noted that in this function the database that is accessed is a different one. In this case, the database is from Altair, and the file altair.py is the one containing the peewee models for this database. Some counts are done in order to provide a summary table with the total number of vulnerabilities found.

### 4.5 Deployment

The system built in this project has a lot of dependencies. Deploying it is not as easy as getting the source code and executing it. Apart from the direct dependencies of the code, all the tools that have to be used need to be installed on the host system. This is a lot of work since not all tools are easy to install.

In order to solve this problem, or setback, an alternative is being provided. This alternative consists on a virtual machine with the system previously installed on it.
The chosen virtual machine is an **Ubuntu Server**. This choice is motivated by the fact that since the system does not have a common graphical interface, it does not need an Operating System with graphical interface. Instead, the choice of a server like distribution motivates the fact that the system is a small web server. Using this setup, the user can mount the Ubuntu Server with the system on a virtual machine, and access the system through a browser.

The creation process of the custom Ubuntu Server is the following. First of all, a clean Ubuntu Server distribution is downloaded from the official source the usand run inside a virtual machine. The next step is placing the source code of the system on the virtual machine. The next step would be configuring the source code to match it with the usernames and passwords presents on the system, since it needs a user to access the database and a user and password to run tools with root privileges. The following step would be installing all the code dependencies. The dependencies are most of them **Python** libraries, which makes them easy to install. However, there are quite a few, and will be listed now.

- Peewee
- Pexpect
- Celery
• RabbitMQ
• Flask
• Beautiful Soup
• MySQLDb

Once all the code dependencies are solved, the next step is installing all the tools that the system has to use. If a user would like to add new tools, s/he would need to install them first. An easy solution to this problem would be using a distribution that already had all this tools pre-installed on them, like **Kali Linux**, but this would condition the usage of the virtual machine. Choosing a server-like distro instead gives the user the possibility of using the virtual machine for other things than only using the penetration testing system.

Installing the tools in the operating system can be challenging, since most of them are not in official repositories. This means that the source code of the tool has to be downloaded from one of the known sources and then the dependencies of each tool have to be resolved manually.

Once all the tools are installed on the virtual machine, the system is almost ready to run. In order to be able to run the system on an operating system without graphical interface a script has been created that runs the necessary tools that need to run concurrently to the web server. Running this script and accessing the server via a web browser would be the last step to start using the system.

All this process has already been done and the virtual machine has been created. Together with the source code of the system, a virtual machine disk image will be provided. This disk image can be opened with any virtualisation software and the system tried out. In the image provided the pre-established user names and passwords will be **ptas**.
Chapter 5

Test Cases

The system has been used in two different scenarios, in order to demonstrate all parts of the system.

The first scenario is a virtual machine with the vulnerable Metasploitable distribution. This test case aims to demonstrate that the system can discover and list vulnerabilities. To do that a distribution with known vulnerabilities has been chosen, to make sure that there are some vulnerabilities to find.

The second scenario has been chosen to demonstrate a part of the system that cannot be tested with the first scenario. The scenario chosen is the Renfe website. This scenario will let the system discover multiple machines and information about the employees at Renfe, since this website has lots of documents online, and one of the tools used on the system uses web engines to find these documents.

5.1 Test Case 1 - Metasploitable

Metasploitable is a vulnerable VMware virtual machine built by the Metasploit team. It is intended for trying exploitation frameworks in a safe and legal environment. This virtual machine contains a Linux system with several vulnerabilities and it can be used to test our system.

First of all, a virtual machine has to be set up with Metasploitable. Once it is set up, the system can start analysing it. The first step is to create a Network for Metasploitable,
and to create an Audit with the IP of the virtual machine containing Metasploitable as target. Once this is done, the penetration testing can begin.

Once the testing ends, the results can be analysed. First of all, the employees section should be empty, since the address used is a private address and there’s no resources directly associated with the virtual machine in our test case. However, since theHarvester searches the internet in order to find email addresses, it could happen that the system finds email addresses that end with the target. This email addresses are not valid ones, but garbage found on the Internet.

Then, the important results on this test case are in the machines section. Here, the system correctly identifies a single machine, and if the machine is selected, a list of open ports can be explored. Together with the open ports, it can be seen that the system has correctly obtained the Operating System and its CPE, and some known vulnerabilities from the Altair database can be obtained using the CPE. As it can be seen, the Metasploitable machine has several open ports, and probably most of them have some vulnerabilities. In our case, the system scans all HTTP ports with Nikto in order to find vulnerabilities in the web servers listening on those ports. In this test case it can be seen that there are two web servers with some vulnerabilities. If we access one of this ports, the vulnerabilities found by Nikto will be displayed. This test case demonstrates that the system works up until the second stage of the penetration testing process.

![Figure 5.1: Employees - Metasploitable](image)
Chapter 5. Test Cases

Regarding the third stage of the process, the Metasploit module `dbautopwn` has been used on the Metasploitable virtual machine and it has been unsuccessful. This means that the module is not as useful as it seemed at the beginning, since it cannot even find vulnerabilities in a system that has been built in order to have lots of them. This means that both the intelligent exploitation and the automatic exploitation using the `dbautopwn` module are not possible on this system.
5.2 Test Case 2 - Renfe.com

This second test case is thought to assess whether the first stage of the penetration testing process is correctly done by the system. In order to demonstrate it, an actual web site will be used. In this case, the chosen web site is the web site from the Spanish railway company called Renfe. Since this is a public web site and we do not have permission to perform a complete penetration testing on it, performing it would be illegal. That is why only the first stage of the penetration testing process will be done in this test case.

After setting up the system creating the Network and the Audit, the Information Gathering phase is launched against www.renfe.com.

Once the audit has finished working, the results can be analysed.

First of all, the Employees section. In this test case it can be seen that several usernames and email addresses have been found. This is so because the website chosen to do the test case publishes lots of documents on the internet. This is why the metadata extractor is able to obtain so many different usernames.

The usernames could be used by a cybercriminal, together with the email addresses, to perform a Social Engineering attack. The usernames could also be used to perform a bruteforce attack on the administrative interface of the website. Usually, this kind of information leakage is disregarded as important, since at first sight does not seem that
dangerous. However, one of the easiest and most common ways to attack a system is through its weakest link, the user behind it, and a Social Engineering attack aims at attacking this link. Studies show that Social Engineering attacks are faster and more efficient than highly technological ones. This is an important thing to have in mind when performing an audit, and it should be included in the report handed to the company.

The next thing that can be explored in this test case is the machines found. In this case lots of machines have been found. This is so because this is an actual website, and the system will find DNS machines, virtual hosts, and machines related to the target.

We can see in the image that some of the machines found have been positively finger-

![Figure 5.5: Machines - Renfe](image)

printed, and the operating system and the Web Application Firewall they use have been identified. Also the open ports on the machines have been identified, and since they are machines exposed to the Internet, they have a much lower number of ports, if compared with the Metasploitable machine.
Chapter 6

Final Analysis

Once the project has been finished, the system built and tested, it is now time to do some conclusions over the work done, and some possible future work.

First, the problems found during the project will be explained, and how where they solved, then an economical analysis of the cost of the project will be done, followed by the conclusions extracted from building the system. After that, some possible future lines of work will be described, in order to improve the work already done, and the knowledge gained by doing this project will be also specified.

6.1 Technical Problems

When building the system described in this project, several problems where detected. The first problem that was detected, while researching about automating tools, was that intelligently automatizing the exploitation and post-exploitation phases of the penetration testing process was a really difficult task. No APIs are available to help solving this problem and no other security framework was doing it at the moment. There were frameworks that launched lots and lots of different exploits, hoping that one of them worked, but there was not enough time in the project’s lifespan to accomplish that.

This problem was a major one, since it reduced the effectiveness of the system. However, since no other framework studied accomplished that, it was decided that the system would center itself on the first stages of the penetration testing process instead.
Another major problem found during the analysis and design phases of the project was the amount of security tools available. There are lots and lots of tools out there, and to build a complete framework, most of them should be used. At least all the aspects of each phase should be covered, but this is again a really harsh job. Performing a penetration test is not an exact science, and even though there are really good guides, it depends a lot on the knowledge and creativity of the tester. Moreover, since everyday new vulnerabilities are found, and new technologies are created, the tools whose aim is to protect and secure these technologies need to be updated constantly. And of course, if a new technology is created, a new security tool that scans this technology has to be created. This means that creating the definitive scanning framework is impossible, it has to be update continuously. Due to the amount of security tools available and the fact that a complete framework should be constantly updated, a few tools where selected to represent each stage of the penetration testing process. This may give the impression that the system has few tools, but finding a new tool, learning how the tool works properly and then building a parsing function for that tool is a time-consuming task.

Finally, the fact that a clear standard for vulnerabilities does not exist was also a problem. Since different tools use different enumeration systems and establishing a relationship between them is really complex, classifying vulnerabilities in a correct and simple way was nearly impossible. In this project, this problem has been solved by using the OSVDB system, since it is the system used by the two vulnerability assessment tools used by the system. If new tools are added that do not use the OSVDB system, and instead use the CVE one, the project’s system will only need some minor modifications to correctly work with the new system.

6.2 Economical Analysis

This project lasted for 6 months. Two major roles can be defined here, since the project is separated in two differentiate parts. The first part would be all the research done, and the analysis and design of the system. The second part would be the implementation and testing. The initial planning of the project was followed, meaning that the first part lasted for 2 months and the second part was 4 months long. The first part should have been done by a security analyst, since it is a really focused
work, with a really big stress on security.
The second part could be done by a junior Python developer because the design is provided, as well as the tools that need to be used and the database model of the system.

- Security Analyst
  - 250 hours
  - 35 € / hour
  - 8750 €

- Junior Python Developer
  - 500 hours
  - 15 € / hour
  - 7500 €

This project did not need any special hardware components, nor any software license. The system could be executed with any desktop computer. This fact makes the project budget’s only factor the salary of the developer and the analyst. Both salaries combined would make the budget for the project be 16250 €.

6.3 Future Work

The development of this project has finished, but the system is not as perfect as it could be. There are some improvements that could be done, that would make the system even more interesting. Some of them will be listed now.

First of all, the most important thing this project needs and deserves is to widen the tool set. The tools chosen for this project are a representative and sufficient set for showing purposes. In order to make the system useful for a security expert, more tools should be added.

Since adding new tools is an easy process, it should not be really difficult. The most important aspect of this improvement is the fact that a security expert with knowledge on
penetration testing would make this improvement really faster than a regular developer
with no prior knowledge about penetration testing.

Another interesting improvement that could be easily implemented is the following one.
Starting from the deployment mode created with the virtual machine, a cloud-based
service could be offered. This would mean upgrading the user system, adding extra
security layers. This upgraded product could be offered to companies as a service. This
service could be useful for system administrators, because they could have an easy way
to test if their system had some security problems. On top of that, the system could be
launched automatically at fixed time intervals and then get a notification if the results
of an audit where different from the previous one.
The cloud-based system would need to have some issues under consideration, like the
availability of the service and the efficiency of the system, which are facts that where
not considered on this project. Probably some multiprocessing should be implemented
in order to improve the efficiency.

Other functionalities that could be included in a close future would be the ability for
user to see which tools where executed and where, and whether they discovered some-
thing new, they crashed, or they executed normally without adding new information.
This would simply mean building a logging system that was available from the user’s
perspective. Another good improvement would be adding a diff function to the system,
both in the logging system and in the results between two different audits. This would
make it really easy for the tester using the system if new vulnerabilities were found in
the system under study.

6.4 Conclusions

In this project, a lot of research about penetration testing has been done. This research
has allowed the building of a system that helps with the first steps of performing pene-
tration tests.
The system that has been built is a functional one, it does what it was expected for it to
do, with the only drawback of having less tools than the desired ones. Several problems
where found during the development of this project, some of them were fixed, and some
were not, but a lot has been learned from them.
The deployment method chosen for the project has opened new ways of exploiting it, opening the system to other markets that were not initially planned. This new markets could offer a good way of making money out of the project, which is another aspect of the system that was not planned.

Personally, I have learned a lot of things while making this project. Besides from the fact that I learned to program in Python, there were more important things that I learned. The field of computer science where this project resides was relatively unknown to me before doing it. Learning how the penetration testing process worked was one of the most interesting things that I did. Lots of tools were discovered, and a lot of practise using some of them was gained. Besides from the penetration testing process, I learned that not everything is easy to automate, there are resources that still do not exist that are needed to fully automate the penetration testing process, and even when this is accomplished, the existence of a security professional is still needed, because most of the vulnerabilities present in web site or networks need the perspicacity that only a human being can have.
Bibliography

[1] INTECO. *Pentest: Recolección de Información (Information Gathering).*


