ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

School of Information Sciences and Technology
Department of Informatics
Athens, Greece

MSc in Information Systems

Master Thesis

Vulnerability Detection Techniques

Angelos David
F3311803

Athens, February 2020

Angelos David
Vulnerability Detection Techniques
February 2020
Supervisor: Dr. Theodoros Ntouskas

Athens University of Economics and Business
School of Information Sciences and Technology
Department of Informatics
Athens, Greece

Abstract

The focus of this Master's thesis project is automated penetration testing. A penetration test is a practice used by security professionals to assess the security of a system. This process consists of attacking the system in order to reveal flaws. In order to perform one, some tests need to be done using several tools. But these tools are the same most of the times making it repetitive and tiring. Automating the process of penetration testing brings some advantages, the main advantage is reduced costs in terms of time and human resources needed to perform the test.

The goal of this thesis is to investigate these repetitive steps and create a tool that automates everything that can be done safely without the risk of damaging the system.

The outcome of this thesis project reveals that this tool is able to provide the same results as a standard initial penetration procedure, avoiding all the costly tasks.

Περίληψη

Αυτή η διπλωματική εργασία θα επικεντρωθεί στην αυτοματοποίηση της διαδικασίας της δοκιμής διείσδυσης (penetration testing). Η δοκιμή διείσδυσης είναι η πρακτική που χρησιμοποιείται από τους ειδικούς της ασφάλειας για να αξιολογήσουν την ασφάλεια ενός συστήματος. Αυτή η διαδικασία αποτελείται από την δοκιμαστική εισβολή στο πληροφοριακό σύστημα με σκοπό να αναδειχθούν κενά ασφαλείας. Προκειμένου να πραγματοποιήσουμε μια δοκιμή, κάποιοι έλεγχοι πρέπει να πραγματοποιηθούν χρησιμοποιώντας διάφορα εργαλεία. Αλλά αυτά τα εργαλεία είναι τα ίδια στις περισσότερες δοκιμές, καθιστώντας τες επαναλαμβανόμενες και κουραστικές. Η αυτοματοποίηση της διαδικασίας φέρει κάποια πλεονεκτήματα, με κύριο τη μείωση του κόστους σε όρους χρόνου και ανθρώπινου δυναμικού που απαιτούνται για αυτή.

Ο στόχος της διπλωματικής αυτής είναι η διερεύνηση αυτών των επαναλαμβανόμενων βημάτων και τη δημιουργία ενός εργαλείου που αυτοματοποιεί τα κομμάτια τα οποία μπορούν να γίνουν με ασφάλεια, χωρίς τον κίνδυνο να βλάψουν το πληροφοριακό σύστημα.

Το αποτέλεσμα της διπλωματικής εργασίας αποκαλύπτει ότι αυτό το εργαλείο μπορεί να παρέχει τα ίδια αποτελέσματα με μια τυπική διαδικασία ελέγχου διείσδυσης, αποφεύγοντας έτσι τις κοστοβόρες ενέργειες.

Contents

List of Figures

List of Acronyms

API Application Programming Interface

CMS Content Management System

CRLF Carriage Return Line Feed

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

DNS Domain Name System

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IP Internet Protocol

JSON JavaScript Object Notation

NVD National Vulnerability Database

OSINT Open-source Intelligence

OSVDB Open Source Vulnerability DataBase

OWASP Open Web Application Security Project

PCI DSS Payment Card Industry Data Security Standard

SAMM Software Assurance Maturity Model

SDL Security Development Lifecycle

URI Uniform Resource Identifier

VM Virtual Machine

XML Extensible Markup Language

XSS Cross Site Scripting

1 Introduction

A penetration test is a practice used by security professionals to assess the security of a system. Many checks need to be conducted using several methods to assess it. The aim of these tests is to find whether the target system is vulnerable to an attack in a certain scenario and can be exploited in order to gain control of it. There are various types of penetration testing, depending on what assets need to be tested (e.g. a network, a single machine, a web application). This thesis focuses on network-based penetration testing, which is one of the most common types of security testing. The main reason for choosing network-based penetration testing is that this testing involves several repetitive tasks that can be performed remotely via a network connection, so automating them is desirable.

The aim of automating the penetration testing is to minimize the costs in terms of time and the people needed to carry out the test. Automation drawbacks include restricted pivoting, stability issues, generation of false positives and less intelligent analysis of potentially sensitive data.

1.1 Problem Statement

There are many tools that can help a security professional do his/her job. These tools help by automating certain actions needed to obtain information about or exploit the target system. Nevertheless, such methods are not adequate to perform a successful penetration test. A high level of knowledge and a high imagination are required in order to succeed. However, the methods that people use are always the same in these first steps, and they have a natural order to be used in. This fact makes the implementation of these methods very tiring, as it is 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 things.

Doing a penetration test must be more about focusing on the creative parts of it rather than having to do the first boring and repetitive steps and that is why the program that will be developed in this project aims at fixing it. Through automating most of the techniques that need to be used in the penetration

testing process, a security expert may focus solely on thinking out of the box and being innovative about the device s/he is investigating. All the tools that need to be launched and all the interaction between them should be automatic. Accomplishing this will make the penetration test faster, more reliable and cheaper as a well-defined set of rules is being followed.

1.2 Goals of Thesis

The main objective of this thesis is to analyze in detail how penetration testing is conducted and to develop and implement a tool that automates the initial steps in the testing process, by eliminating most of the repetitive steps that a penetration test needs to go through.

This tool should be ideal for use in production environments, should automate everything that can be done safely and without the risk of interruption of service (although it may produce some traffic for some periods of time), and should recognize techniques that the human tester might use.

To develop this system, a lot of research is needed in order to understand which steps can be automated and which tools can be used in each step. Then, with the process of penetration testing in mind, a system will be designed and developed that gathers the tools, executes them and integrates their results. For security professionals to benefit from this system, the results should be clear and easy to understand.

After the consumer has implemented countermeasures, the steps that led to the discovery of a flaw should be possible to reproduce in order to verify that the problem has actually been solved. This would allow the customer to easily evaluate the effectiveness of the solutions without the need for a professional security tester, therefore this professional would only be required for the penetration test itself.

1.3 Thesis Structure

This thesis will be structured in 7 different chapters. First of all, this introduction. Following the introduction, the importance of the penetration testing will be discussed, the process of a penetration testing will be defined and detailed and some solutions to the same problem will be analyzed. After, the

system will be discussed, starting with its analysis and design and ending with its implementation. After the system has been described, a test case will be reproduced and its results will be displayed in order to demonstrate how the system works. Finally, some evaluation and future work will be specified and some conclusions will be made.

Background

2

In this chapter an overview of the key elements needed to fully understand the rest of this thesis is provided. Basic concepts of security testing and the main motivations behind the decision of performing a penetration test will be presented. The integration of security testing in the secure software development lifecycle will be covered. Afterward, the standard penetration testing process will be described. Understanding this part is essential for the development of a tool to automate the process.

2.1 Basic Concepts

Security testing validates software system requirements related to security properties of assets that include *confidentiality, integrity, availability*. These security properties can be defined as follows[1]:

- Confidentiality is the assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- Integrity is provided when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- Availability guarantees timely, reliable access to data and information services for authorized users.

Security requirements can be defined as *positive requirements*, specifically defining the expected security functionality of a security mechanism, or as *negative requirements*, stating what the program should not be doing. For example, for the security property authorization as positive requirements could be “User accounts are disabled after three unsuccessful login attempts.”, while a negative requirement could be formulated as “The application should not be compromised or misused for unauthorized financial transactions by a malicious user.”.

An *asset* is a data item, or a system component that has to be protected. In the context of security, an asset has assigned one or multiple security properties.

A *fault* is a textual representation of what goes wrong in a behavioral description. Since faults can occur in dead code - code that is never executed -, and because faults can be obscured by further faults, a fault does not necessarily lead to an error. On the other hand, a fault often results in an error. A fault is not necessarily related to security properties but is generally the cause of errors and failures.

A *vulnerability* is a particular type of fault. If the fault is related to security properties, it is called a vulnerability. A vulnerability is always related to one or more assets and their corresponding security properties. An exploitation of a vulnerability attacks an asset by violating the security property associated to it. Since vulnerabilities are always associated with the protection of an asset, the security relevant fault is typically correlated with a mechanism that protects the asset. A vulnerability either means that the responsible security mechanism is completely missing, or the security mechanism is in place but is implemented in a faulty way.

An *exploit* is a concrete malicious input that makes use of the vulnerability in the system under test and violates an asset's property. Vulnerabilities can often be exploited in different ways. One concrete exploit selects a specific asset and a specific property, and makes use of the vulnerability to violate the property for the selected asset.

A *threat* is the potential cause of an unwanted incident that harms or reduces the value of an asset. For instance, a threat may be a hacker, power outages, or malicious insiders. An attack is defined by the steps a malicious or inadvertently incorrectly behaving entity performs to the end of turning a threat into an actual corruption of an asset's properties. This is usually done by exploiting a vulnerability.

Security aspects can be considered on the network, operating system and application level. Each level has its own security threats and corresponding security requirements to deal with them. Typical threats on the network level are distributed denial-of-service or network intrusion. On the operating system level, all types of malware cause threats. Finally, on the application level threats typical threats are related to access control or are application type specific like Cross-Site Scripting in case of web applications. All levels of security can be subject to tests.

Security testing simulates attacks and employs kinds of penetration testing that aims to compromise a system's security by playing the role of a hacker

attempting to attack the system and exploit its vulnerabilities. By identifying risks in the system and creating tests driven by those risks, security vulnerability testing can focus on parts of a system implementation in which an attack is likely to succeed.

2.2 Why perform a penetration testing

There are several reasons why an organization should hire a security professional to perform a penetration test. The main reason is that security breaches can be extremely costly. A successful attack can result in direct financial losses, damage to the reputation of the organization, trigger fines, etc. It is possible to identify security vulnerabilities with a proper penetration test and then take countermeasures before a real attack occurs.

Another reason for conducting penetration testing is that it will pressure the system operator to keep the system up-to-date on the latest vulnerabilities. New bugs and security issues are being discovered all the time. So, an organization, in order to maintain an updated level of security, may use periodic penetration testing.

The result of the penetration test helps an organization prioritize its risks. A specific security breach causes a certain damage to the organization. Depending on the severity of the identified issues, a mitigation strategy can be properly prepared with a greater focus on more critical issues.

Since a penetration test simulates a real attack, it is a good chance to evaluate the preparation of the organization's technical staff in such situations. For example, if the testers are able to compromise the system without anyone noticing, it is a clear indication that more effort should be put into security awareness and handling of incidents.

Penetration tests may also be required for compliance with industry standards and regulations. A regularly conducted penetration test is needed in order to achieve compliance. Common compliance frameworks include ISO 27001[2], NIST[3], FISMA[4], HIPAA[5], Sarbanes-Oxley[6] or the Payment Card Industry Data Security Standard (PCI DSS)[7], which requires annual as well as ongoing penetration testing (in case of system changes). By conducting regular penetration tests of the environment, the organization demonstrates

information security due diligence and can avoid hefty fines resulting from non-compliance.

2.3 The Penetration Testing Process

The aim of a penetration test is to asses the exposure level of the system being tested, and to determine whether there are ways to break into the system. In addition to the actual testing phase, a few operations need to be performed in order to conduct a valuable and legitimate test, which will be described in this section. The process of a professional penetration test can be divided into four main phases: planning, preparation, attack and reporting.

2.3.1 Planning

The planning phase involves an initial discussion with the customer (owner of the system being tested) aimed at establishing an agreement with the penetration tester(s). In this phase, the two parties define the scope of the test, the people responsible for the different tasks, the actions permitted to the testers and the identifying test timelines. A team is established, and contact information (for emergency) is exchanged. Management consent and written permission are needed before moving to the next stage.

2.3.2 Preparation

Before starting the actual penetration testing, a preparation takes place according to the agreement established during the planning phase. If more than one penetration tester is involved in the testing, then the work is organized and divided within the team. The appropriate tools are selected and configured accordingly, based on the tasks which need to be executed. This phase requires the testers to take into account the integrity and stability of the system under test. This is a critical aspect when deciding what actions will be taken during the test.

2.3.3 Attack

This phase involves the actual testing and closely resembles the steps taken by an attacker who is not authorized to access the system, and whose goals

are of a malicious nature. Every action taken during the test phase must be logged so that it will be possible to analyze the history in case unexpected situations arise. Communication with the customer is also critical in specific situations where the systems owner's approval is required by the penetration tester before taking any action. The attack process involves several different steps, as described in the sections below. Some of these steps are repeated over time as new pieces of information are gathered allowing the tester to fill in earlier gaps or to explore new areas of the system being tested.

Target identification

Target identification consists of gathering information about the system under test such as available domains, IP addresses, internal resources, security policy, etc. The importance of the target identification phase depends on the amount of information available to the penetration testing team at the beginning of the test. It is necessary to identify the target, particularly in the context of an external penetration test, i.e. when the tester has no initial access to internal resources. Useful information can be found using a variety of techniques, such as exploring a website gathering information from search engines or doing social engineering.

Port scanning

Port scanning is the first part of the process of penetration testing involving an active interaction with the system under test. This consists of scanning the network to find out which hosts are available, which ports are open, and which services are running. Normally a tool is used to perform this task like Nmap[8].

Enumeration

Once the penetration tester has developed an overview of the hosts and services that are part of the system being tested, it's time to identify those that are most likely to be vulnerable. Enumeration consists of collecting information on the services in the system in addition to the port scan results. Examples of such information include the application version in use, well-known bugs, password lockout policy for a particular service, etc. Such

knowledge helps the tester to assess the weakest point(s). In this step, the tester's expertise is of great help, while tools can also be used to assist the tester.

Penetration

Penetration is the act of exploiting a weakness that has been identified in the system under test. An exploit is the means by which a penetration tester (or an attacker) takes advantage of a flaw within the system, resulting in a behavior that the developers never intended. The goal of the exploitation is to gain access to a certain resource, for example by obtaining a remote shell used to control a machine over the network. Examples of common exploits are buffer overflows, SQL injections, configuration errors, etc.

Since exploits are likely to cause temporary or permanent damage to the system under test, it is the penetration tester's responsibility to determine whether it is acceptable to use a certain exploit. Maintaining good communication with the customer usually helps the tester to make these decisions. As described in chapter 2.3.1 the tester is not normally permitted to perform actions that are potentially dangerous to the stability and integrity of the system being tested.

In contrast to what happens in a penetration test, stability issues rarely affect the penetration phase of the hacking process. Generally, a hacker is not concerned with the possibility of service interruption due to the adoption of aggressive exploits, unless the use of such exploits would increase the probability of detection.

Escalation

When a vulnerability is successfully exploited, the access gained to a resource is often limited. For instance, the penetration tester could gain access to a low-privileged user account, but higher privileges are needed to perform certain operations. The escalation phase consists of further exploiting a resource to increase the influence of the tester on the compromised machine.

Getting Interactive

The fact that a host in the system under test is compromised does not necessarily mean that it can be easily controlled. An interaction mechanism is needed for the penetration tester to perform operations on the compromised machine in the same way as an administrator would. Occasionally, exploits provide the tester with an interactive interface directly (e.g. a shell to remotely control the resource), but when this is not possible an additional phase to gain interactive access (graphical or command line based) is needed.

Pillage

Pillaging takes place when (limited) access is gained to the system being tested, and consists of collecting information about the resource and being compromised and potentially other network entities (e.g. routers or hosts). The goal of this phase is to expand the penetration tester's influence on the system and possibly identify additional vulnerabilities without the need to exploit them. For example, the tester might extract credentials from local databases, read the users' passwords in their hashed form, analyze firewall configurations, etc.

Cleanup

A professional penetration tester must not leave anything on the system that was installed during the test. Every altered configuration must also be restored to its original state. The purpose of the cleanup phase is to avoid introducing additional vulnerabilities in the system under test. The goal of this phase is different from that of a hacker. A hacker is concerned with removing all traces of his/her presence in the target system to avoid being detected and identified. However, a hacker might be interested in leaving a backdoor, i.e. a mechanism to later regain the same access level without the need for exploiting the system again.

2.3.4 Reporting

The final phase of a penetration test is the report of the test results. The report includes a summary of the vulnerabilities that were found during the test how the vulnerabilities could be exploited and recommendations on how to fix them. From the customer's point of view, merely having a list of the identified issues does not provide much value. Therefore, organizing a meeting is often preferred where the content of the report can be discussed and the penetration testers can explain clearly to the customer what really happened during the penetration test. Moreover, the severity of the vulnerabilities that were discovered can be discussed and defined with the customer. The severity indicates a vulnerability's level of danger and is "calculated" on two factors: the likelihood that a vulnerability will be exploited, and the impact that a possible exploitation could have on the company. The penetration tester knows only the technical extent, but the consumer should estimate the impact for their company that a specific security breach would have.

2.4 Related Work

In this chapter, two existing solutions similar to the project under development will be studied. These two solutions are both security frameworks which aim to automatically scan web servers. These two frameworks are *Golismero*[9] and *OWASP OWTF*[10].

Golismero

Golismero is a free software framework for security testing. It is oriented towards web security and contains many well-known security tools such as *XSSer*, *theHarverster*, *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 for the rest of the tools. Once it has finished, it merges the results and displays them.

It has integration with known vulnerability standards such as *CVE*, *CWE* and *OWASP*, and allows plug-ins to be built to incorporate new tools into the system.

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

This framework is aimed at 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. These 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.

OWSAP OWTF

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

The framework is *OWASP Testing Guide*-oriented, meaning that it classifies its findings as closely as possible to the testing guide provided by *OWASP*[11]. It also reports its findings on the fly in an HTML page, 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 system. However, it needs the tools to be installed in order to run them. It is easy to control and easy to run, using a command line tool for it, and it provides some examples of its usage with several examples. It stores the information obtained on a database and extracts data from it in a parsable format as well.

This system, thanks to having *OWASP* behind it, has a great community of developers that are constantly upgrading it.

System Design

3

This chapter describes the various phases that took place during the design of the automated penetration testing tool. At the beginning of this part, a few important issues had to be considered (these are described in section 3.1) that determined the objective of the tool that will be created. In section 3.2 an analysis will be provided about the tools selected, the technologies used and the different phases of this tool. In section 3.3, the architecture of the tool will be explained. In section 3.4, a brief comparison of the tool with similar existing tools will be presented.

3.1 Approach

The main objective of this project is to automate the process of penetration testing process in order to help security experts with their work. This main objective creates some key points that also need to be noted. These key points will be addressed in this section.

- The tool 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 easily integrate new tools.
- The results the system has found should be presented in an easily understandable way.

Therefore, these main points are the objectives this program needs to cover in order 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 tool that only runs the *Nmap*[8] tool won't be of any use to a penetration tester, since s/he can run *Nmap* directly instead. The goal is to cover most of the parts of a penetration test, beginning with the information gathering phase and using the information obtained to progress through the different steps and to create a report simple to understand. Since the process is linear, this tool will also be linear, starting with the information

gathering step and once enough information is collected, it will be used to start the vulnerability assessment phase, and so on.

3.2 Analysis

The main question that needed to be answered at the beginning of the design phase was what tools should be selected. Each tool had its advantages and disadvantages that will be described in this section. Another thing that should be considered is what technologies are more suitable to be used in order to create a tool like this. In the end, the different phases of a penetration test will be related to the tool under construction.

3.2.1 Tools

The most important part of this system will be the tools it uses. This will be the main strength, since it will gather more information as it integrates more tools. While choosing tools to add to the program, there are some aspects to consider. First of all, since the aim of this tool is to be as automatic as possible, the tools should be as automatic as possible, so they should not require constant user interaction. This is an important aspect, as it discards some of the tools that were considered to be used. There exist several frameworks and complex tools that use graphical interfaces to interact continuously with the user. This type of tools or frameworks is not recommended for this program even though they are generally more advanced and powerful. This is because the interaction with the user should be minimal.

Such minimal user interaction can be easily achieved with command-line tools or external tools that can be launched via the command-line and need no interaction. This does not mean that the tools need to be simple ones. For example, *Nmap* is a command-line tool and is a powerful tool, capable of scanning a whole network and discovering all of its machines and services that reside on those machines' open ports. Another example would be *theHarvester*[12], a command-line tool that uses search engines to discover usernames, email addresses and virtual hosts related to a target.

Another interesting aspect regarding the tools to be used is the ability to analyze their results. The tools included in the program should be ideally in an easy-to-parse output format. Usually, this kind of tools has XML and JSON output reporting, which are some of the easiest ways to parse the results.

There exist parsing libraries in almost all current programming languages. Parsing the tools will be one of the system's key points, as it is the part where a tool's output is collected and parsed. When you want to add a new tool to the program, you need to build and add a parser for this tool.

There are tools made by security companies in the market currently that require high fees to use them. All the tools used in this tool will be free. However, due to it being expandable, proprietary tools could be added by the users afterward.

With all these considerations 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. The information gathering step can find things like machines and ports, but it can also find usernames, email addresses and services residing on the ports found. It is really important to have this wide range of information on the system because it gives the security expert lots of possible entry points.

3.2.2 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 work. However, it would be better if the tool is in the form of a website, so the user will be able to use it from anywhere, without having to carry always the appropriate equipment in order to perform a scan. So the tool will consist of a back-end doing all the scanning using the tools and the analysis of the results and a front end which will give the opportunity to the user to choose the tools to be used and showing the results in a nice way.

After taking all these into consideration a server written in *Node.js* and a front-end written in *Angular* seems like a good idea. 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, *Node.js* is lightweight and efficient, and taking in mind that the tools that will be used may need a lot of power to perform

a scan, it would be better to have as much free computational power it's possible. Also having the ability to use JavaScript on both front-end and back-end adds an advantage for development. Moreover, the *Node.js* package ecosystem, *npm*, is the largest ecosystem of open source libraries in the world, so a lot of packages that will make the development easier can be found. After some research, interesting libraries can be found that are really useful for the project. Last but not least, *Node.js* uses an asynchronous architecture, which JavaScript can handle well. This is very important when executing the tools, because it is impossible to know how long each tool will take and an asynchronous style is needed.

One of these packages is *Express* which is a minimal and flexible *Node.js* web application framework that provides a robust set of features to develop web and mobile applications. This package makes it really simple to code and deploy an API that is able to manage and respond to HTTP requests. Another good point towards *Node.js* is that it has a module called *'child process'*. This module lets the user launch bash commands in a controlled way. It also retrieves the output of the command and presents it as a String. The functionality provided by this module is clearly useful for the system under construction. Last but not least packages that parse different types of resources into JSON make the reading job faster, easier and more consistent.

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

3.2.3 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 to be used will be listed here, as well as some explanations on the problems found during each phase analysis.

Information Gathering

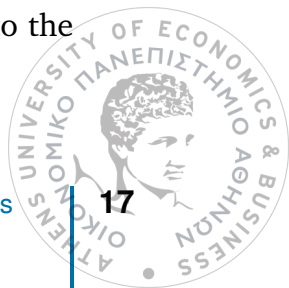
At this stage of a penetration test, there are lots and lots of tools that could be used. There are lots of areas that need to be covered, and lots of information to be found about a target. An important part of this stage is that no prior knowledge is required for the target. This means that all the 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 stored. 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, *Nmap*[8] is one of the most important penetration testing tools. Clearly, due to its high usefulness, this tool had to be used. *Nmap* is able to discover interesting things such as the operating system of the machine and the opened ports that this machine has. In addition to discovering the ports, it also reveals the service that is listening at each port. All of these things make it an effective tool to use as all the knowledge that it collects is really useful and establishes the attack vector for the next phases. In this system, *Nmap* will be used to discover the operating system of the machine, its active ports, the versions and other information of the services running in each port.

Another area that needs to be analyzed is the enumeration of the URIs of a server. In order to find different URIs of the web server *Gobuster*[13] will be used. With the use of a wordlist[14], this tool brute forces the web page and returns all the URIs that exists. This will give a better understanding of the contents of the server that is being tested. This tool can also be used to discover subdomains, but for this project, *Gobuster* will only be used to discover the URIs, this is so because another tool will be added that covers the other aspect of *Gobuster*.

Another aspect that this phase has to cover is the identification of the web technologies used on a website. These technologies include content management systems (CMS), blogging platforms, statistic/analytics packages, JavaScript libraries, web servers, and embedded devices.

Search engines can provide lots of information about a target. The information collected from search engines can usually 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 *Sublist3r*[15]. This tool enumerates subdomains using many search engines such as *Google, Yahoo, Bing, Baidu* and *Ask*. It also enumerates subdomains using *Netcraft, Virustotal, ThreatCrowd, DNSdumpster* and *ReverseDNS*.
The second tool that will be used is called *theHarvester*[12]. This tool uses search engines to directly search for 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.

Vulnerability Assessment

This penetration testing step also has lots of tools available. Since there are many different vulnerability types, there have to be tools to cover them. Most of these tools, however, require some kind of prior knowledge about the target. For example, the tester needs to know that the target has a *Wordpress* web server to launch a *Wordpress vulnerability scanner*. Another example would be *SQL Injection scanning*. This type of scanning needs a vulnerable link in order to launch all the known attacks. These 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 *Wapiti*[16]. This tool will be one of the tools used in this phase of penetration test. It works like a fuzzer, scanning the pages of the deployed web application, extracting links and forms and attacking the scripts, sending payloads and looking for error messages, special strings or abnormal behaviors. It supports a variety of attacks, including *SQL injection, Cross Site Scripting, local file inclusion* and *remote file inclusion, CRLF injection* and others. When finished it presents all its findings with the appropriate link to the web server and some information about the things it discovered.

Another such tool is *Nikto*[17]. It scans web servers to find potentially dangerous files or CGIs, checks out of date server versions and scans server configuration items such as multiple index files or HTTP server options.

Nikto, like *Wapiti*, when 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.

Nmap will also be used in this phase too. *Nmap* has a feature that scripts can be imported and executed through it in order to automate a wide variety of networking tasks. Some scripts that will be used are for well known vulnerabilities, like the *POODLE attack*[18], the *Heartbleed attack*[19] and the *DROWN attack*[20]. Also, another script that will be used is *nmap-vulners*[21]. Some of the information that *Nmap* gathered from the information gathering phase, like services running on ports and their versions, will be crossed checked with the *National Vulnerability Database (NVD)*. This will show some well-known vulnerabilities for the system being tested according to the services that are running on it providing their ids inside the *Common Vulnerability and Exposures (CVE)*. This id can be used to find more information about a particular vulnerability, for example on the MITRE website.

However, there is more than one way of classifying vulnerabilities and different entities have different databases. The most common is the *CVE* mentioned above, but another one is the *Open Source Vulnerability Database (OSVDB)*. Since there is more than one system, it happens that different tools offer different information. In our case *Nmap* uses *CVE*, but *Nikto* provides O*SVDB* ids. The desirable case would be to be able to convert *OSVDB* ids to *CVE* ids to use the same classification throughout the tool, 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 tools in this step, another scanning tool will be used. The tool is called *WPScan*[22] and it is used to scan *Wordpress* websites. It scans for outdated plugins and themes and can obtain default users. It crawls the website to find usernames used and the use this data to perform a bruteforce scan to find the credentials. For this scan, a wordlist[23] must be provided. Another feature provided by *WPScan* is the ability to use its vulnerability database[24]. When using it *WPScan* is able to retrieve vulnerability data in real time, providing information like which versions of the plugins are vulnerable to the exploit, in which version was it fixed and the *CVE* id to

search for extra information. One restriction though is that this feature is not free. But 50 requests can be done per day free, which are sufficient to showcase this tool.

The last tool that will be used in this step is *Joomscan*[25]. This tool enables scanning of *Joomla* installations, while leaving a minimal footprint with its lightweight and modular architecture. It can *enumerate the components and detect known vulnerabilities, firewall, misconfigurations and admin level shortcomings* that can be exploited by adversaries to compromise the system.

Exploitation

The number one tool for exploitation is *Metasploit*[26]. It provides the whole environment for detecting targets and exploiting them. However, our system aims to be automatic and the automatic use of *Metasploit* isn't that simple. As it has been stated previously, there is not a clear standard on vulnerabilities. Because of 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 hard. One possible solution might be to launch all known exploits on target independently, but that would be extremely time consuming and potentially dangerous to cause some problems at the system being tested. However, there exists a module for *Metasploit* called *db_autopwn*[27].

db_autopwn is a module that feeds from the *Nmap* output and launches all the exploits it knows. 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 tool'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. Taking all this in mind, it was decided to leave this module out of the tool.

Post exploitation

Once a pen tester has access to a system, s/he still has a bit of work to do. The access point sometimes isn't that interesting and the tester has to move from one machine to another before s/he discovers interesting things. 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 dependent, 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. Moreover, the system should be intelligent enough to know which machines could contain sensitive information inside the network. All these things are really complex to automate and would require Artificial Intelligence. This is obviously out of the scope of this project.

Therefore, 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 stage where a report of all the findings is generated by the tool. 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 to show the results. Since Angular makes it easier 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 it easier to find details and is much more concise than a report of 200 pages. The information will be provided depending on what tools were chosen to be executed, and the tester will be able to select which information s/he wants to see.

All this information gathered by the system is also stored in output files from each tool. This fact gives the possibility of outputting a PDF report of all the findings, but it will not be a priority of this project. The web interface is believed to be much more intuitive and easy to understand the way of showing the results.

3.3 Architecture

Tools Integration

The only thing all the tools the system uses have in common is that they are all command-line tools. This could lead to a problem in the sense that for each tool that needs to be used, a custom class should be created. 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 will be a generic class that will be able to call any command-line tool with root privileges. This module is called *child_process* and it will spawn a child with the execution command it will be provided. It also calls the tools with root privileges because certain tools need these privileges to be used. So, it will receive a command as a parameter, it will execute it and it will return the result.

This tool integration has two different parts. The class that launches the tools that the user has chosen to be executed, which will be called custom and the class which will determine which tools will be executed automatically based on the information gathered in the previous steps, which will be called auto.

For each tool a different class will be created, which will contain the different options that can be used for this tool (execute the same tool with different parameters) and its parser, that would parse the output returned from the previous class. A different parser for each tool cannot be avoided to be created, cause every tool has a different output format, so different strategies have to be used.
One of the most common and simple parsing types is the JSON file. This will make it really easy to obtain the information in later stages. So, when a tool finishes its execution, its parser function processes the output and creates a

JSON object. This object will have the tool name as key.
With this system, calling any tool is really simple, and the only thing that has to worry the user when adding a new class for a new tool is to contain the command to be executed and its own parsing function.

The next step in the design of the system is the step where the scans are launched. In order to do that, the system receives an array with the tools that the user selected to run and then use its class to launch them. The part of the system that chooses which tools are going to run is called handler.

The execution time of these scans can be really long, so it would be a good idea to show at which stage is the tool at every moment. Therefore, it seems right, each time a tool is executed or finished, an appropriate message to be emitted.

Web interface

A big part of this tool is the web interface, which will show the results of the system, by interactively showing the outputs of the tools. It will start with a screen with all the tools shown and separated by category. Each tool would have a checkbox, and the user can create a scenario of a scan and choose here the tools suited for his/her needs. Also, an input will exist, where the user will be able to specify the target. Under the target input, the user will be able to specify a proxy where all the traffic will pass through. Under all these, an interactive *MITRE matrix*[28] will be shown and whenever a user chooses a tool the categories in the matrix that this scan falls will be highlighted. When the tools are chosen and the target is specified there will be a button to start the scan. Also, the user has the choice to select, from the navbar, whichever tool is needed, or choose the custom scan.

When the scan is started a loading screen will be shown which will be updated in live time displaying the tool that is currently running. When the scan finishes, two blocks are shown. The first one is the summary of the scan, containing the phases/tools that were executed and the second, is the important one, which shows the output of the scan with all the important information found. If any known vulnerabilities exist, a link will be provided with information about it. The output depends highly on the type of the scan the user selected. If the user executes only one tool, the output will be this

tool result. If the user selects more tools or automated scan, then a good amount of information will be displayed.

3.4 Comparison with other tools

The goal of this section is to analyze the differences between similar automation penetration tools (summarized in section 2.4) and the process followed by the tool described above.

The main difference between these approaches is that this tool uses the scanners specified and created for each occasion. In other words, the existing automated penetration approaches would scan a website for common vulnerabilities in a general way, like SQLi and XSS vulnerabilities scan and misplaced or misconfigured files scan, without taking into consideration any variables of the specific website.
On the other hand, the tool would execute the same scans for SQLi and XSS vulnerabilities and misconfigurations, just like the other tools, but it would also identify the technologies running on the website. This enables it to launch the appropriate tools, specially created to test each technology, and provide more accurate scans. For example, if WordPress CMS is detected on the website, then a scan using the *wpscan* tool would run, which will be able to provide even more specific vulnerabilities about the theme, the components and the plugins used, something that the other tools could not detect.

Moreover, a lot of the existing tools can only be launched via terminal, like *Golismero* (see 2.4), while some, like the tool described in this thesis and *OWASP OWTF* (see 2.4), provide a web interface making it easier to configure and manage a scan. Also, with a web interface provided it is easy to configure the tools on a remote server, with the software and hardware needed and make the scans remotely, without having to carry always a machine capable to perform this type of scans.

	Golismero	OWASP OWTF	Tool
Add more tools with plugins	✓	✓	✓
Interactive Report	✓	✓	✓
Web Interface		✓	✓
Use of situational tools			✓

Tab. 3.1: Comparison Table

Implementation

4

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

Tools integration

As it has been said in the design section, a generic class that will be able to execute all the tools has to be created. This class is called *execCommand* and it is stored in the file *command.js*. This class uses node package *child_process* that has some simple functions that run the command line tool.

```
const { exec } = require('child_process');

exec(command, { cwd: cwd }, (err, stdout, stderr) => {

    if (err) {
        console.error("Exec error: ", err);
        reject("Command failed: " + command);
        return;
    }
    if (stderr) {
        console.log("Command stderr: " + stderr);
        reject(stderr);
        return;
    }
    if (stdout) { // the command finished successfully
        resolve(stdout);
    }
});
```

The function exec spawns a shell. The command that will be executed is given as the first parameter. The function then has to wait until the process finishes. If the command runs but produces an error, then this error is stored in the *stderr* variable. If the command is executed successfully then the output is stored in the *stdout* variable. If any other error is produced it is stored in the *err* variable. Once the process has finished, the output is returned to the

function that called the exec.
The timeout established for the exec 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 (around fifteen minutes), and the default timeout for the spawn function would be not enough.

The other main part of our program is the tool classes. Each tool has each own class which contains the available commands (different type of scans) for this tool and the parsers. Not all classes will be explained here, since most of them have the same logic. Some examples will be described.

One of the examples is *Nmap*. Since the *Nmap* class is quite long, it will be explained in parts. First of all, the command class, that was analyzed earlier should be imported, in order to use it.

```
const command = require('../Utils/command');
```

As stated before, each tool can have more than one type of scans. So when a tool is called it must determine which command to run.

```
function nmapTool(url, type) {
    switch (type) {
        case "default":
            nmapTool_default(url)
                .then(commandOutput => resolve(commandOutput
                    ))
                .catch(err => reject(err));
            break;
        case "vuln":
            nmapTool_vuln(url)
                .then(commandOutput => resolve(commandOutput
                    ))
                .catch(err => reject(err));
            break;
        case "heartbleed":
            nmapTool_heartbleed(url)
                .then(commandOutput => resolve(commandOutput
                    ))
                .catch(err => reject(err));
        case "poodle":
```

```
            nmapTool_poodle(url)
                .then(commandOutput => resolve(commandOutput
                    ))
                .catch(err => reject(err));
        case "drown":
            nmapTool_drown(url)
                .then(commandOutput => resolve(commandOutput
                    ))
                .catch(err => reject(err));
    }
}
```

As seen in the code, the function receives as parameters the target to be scanned (the variable *url*) and the type of the scan. The right scan is then called. The default scan function will be explained further.

Each scan function has two parts. The first one is the part where the command is executed and the second one is the parsing of the output.

```
function nmapTool_default(url) {
    defaultScan(url)
     .then((commandOutput) => {
         if (commandOutput !== 'success') {
             reject(commandOutput);
         }
         toJson(file)
             .then((json) => {
                 getInfo(json, type)
                     .then((nmapInfo) => {
                         resolve(nmapInfo);
                     })
                     .catch((err) => {
                         reject(err);
                     });
             })
             .catch((err) => {
                 reject(err);
             });
     })
     .catch((err) => {
         reject(err);
```

```
    });
}
```

The default scan is called with the target as parameter. Then if there is an error while the tool is being executed, the error message is returned. If the command was executed successfully the output is sent to the parser.

The first part of the function will be explained here. This *Nmap* scan tries to identify the operating system of the machine and the versions of the services running on each port. As referred, the parameter of the target is given. Then the output will be stored to a file, in order to read it and process it in the next steps.

```
const defaultScan = (url) => {
    command.execCommand(config.NMAP + ' -oX outputs/
        nmapOutput_default.xml -O -sV ' + url, 'nmap')
        .then((commandOutput) => {
            resolve(commandOutput);
        })
        .catch((err) => {
            reject(err);
        });
};
```

As it can be seen in the code above, the *execCommand* is being called and as parameters there is the command that will be executed and the name of the tool. In the command the *config* is something that has not been explained. The *config* is a file that contains all the paths of the executables of each tool, and some other paths like wordlists that some tools use. So, in this example the *config.NMAP* is the path of the *Nmap* executable file. When the command finishes its execution the result is stored in the file that was given as parameter to the command.

Now for the second part, the output, that was stored to a file, is sent to the parser for process.

```
function getInfo(json, type) {
        const ports = json.host.ports;
        const address = json.host.address.addr;
        const name = json.host.hostnames[0].name;
```

```
        const os = json.host.os.osmatch[0].product.osfamily
           + ' ' + json.host.os.osmatch[0].product.osgen;
        const openPorts = [];
        const vulns = {};
        ports.forEach(element => {
            if (element.state.state === "open") {
                openPorts.push(element.port.portid);
                if (typeof(JSON.stringify(element.script))
                   !== 'undefined') {
                    const scriptString = jsonEscape(JSON.
                       stringify(element.script));
                    const scriptObj = JSON.parse(
                       scriptString);
                    vulns[element.port.portid] = parseVulns(
                       scriptObj.vulners);
                }
            }
        });
        resolve({ 'ports': ports, 'openPorts': openPorts, '
           vulnerabilities': vulns, 'address': address, '
           name': name, 'os': os });
}
```

The parser looks for all the ports found by Nmap in the host. Also, important information is stored, like the address and the name of the host and its operating system. After that, for each port, it finds its number, the service that is listening to that port and its state. Once it has all this information, a JSON object is being created and sent back as an answer.

Not all tools provide a file output or the output that they provide is not in a parsable format. In this case, the parsing function has to play with *strings, delimiters, string split() function* and *includes() function* provided by JavaScript. A simple example is the *Joomscan* tool, which does not provide a file output. A part of the parsing function is the following one.

```
var array = fs.readFileSync(file).toString().split("\n");
var json = new Object();
var key = "Joomla Scan";
json[key] = [];

for (var i = 0; i < array.length; i++) {
```

```
    if (array[i].includes('FireWall Detector')) {
        key = "Firewall Detector";
        i++;
        json[key] = [];
        while (!array[i].includes('Detecting Joomla Version
           ')) {
            if (array[i] != '') {
                if (array[i].startsWith('[++] ')) {
                    json[key].push(array[i].split('[++] ')
                       [1]);
                } else {
                    json[key].push(array[i]);
                }
            }
            i++;
        }
        i--;
    } else if (array[i].includes('Detecting Joomla Version')
       ) {
        key = "Joomla Version";
        i++;
        if (array[i].startsWith('[++] ')) {
            json[key] = array[i].split('[++] ')[1];
        } else {
            json[key] = array[i];
        }
        while (!array[i].includes('Core Joomla Vulnerability
           ')) {
            i++;
        }
        i--;
    } else if (array[i].includes('Core Joomla Vulnerability
       ')) {
        ...
    }
}
```

This type of output requires some creative string manipulation in order to obtain the part that is useful. The usage of the *includes()* function is a must, and playing with String delimiters also helps in obtaining the interesting parts. 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 in information.

Before explaining the handler class, where the system calls the appropriate tools based on the user's choices, there is a preparation preceding. This is the handler function, where the request from the frontend is being received. There, the system checks if there are any problems with the target provided by the user, if it is valid, if it is provided with the wrong way, if it redirects, etc. and it will try to fix it. Here is the part that checks for redirects.

```
command.execCommand('curl -v -L ' + url + ' 2>&1 | egrep "^<
    (Location:)" | tail -1; echo "";', 'redirect')
    .then((commandOutput) => {
        if (commandOutput.includes('Location:')) {
            const redirectUrl = commandOutput.split('
               Location: ')[1];
            resolve({'redirect': true, 'redirectUrl':
               redirectUrl});
            console.log('Redirected to: ' + redirectUrl);
        } else {
            resolve({'redirect': false})
            console.log('No redirection');
        }
    })
    .catch((err) => {
        reject(err);
    });
```

And here is the part checking if the URL has the appropriate format.

```
const cleanUrl = (url) => {
    let cleanUrl = url;
    cleanUrl = cleanUrl.replace(/[^\x20-\x7E]/g, ''); //
       remove non ASCII chars
    cleanUrl = cleanUrl.replace(/(^\w+:|^)\/\//, ''); //
       remove protocol in front of the url
    cleanUrl = cleanUrl.endsWith('/') ? cleanUrl.slice(0,
       -1) : cleanUrl // remove the slash at the end of the
       url
    return cleanUrl
```

```
}
```

Also, this function is responsible to wait for all the tools to finish executing, receive the scan output and return it to the frontend.

Then, in order to launch the tools, the *handler* needs to be called. There, the tool classes are imported in order to be called. As it can be seen, it receives the type of the scan and then launches the appropriate function.

```
switch (tool) {
    case "nmap":
        nmap.nmapTool(url, 'common')
            .then((nmapInfo) => {
                resolve(nmapInfo);
            })
            .catch((err) => reject(err));
        break;
    case "whois":
        whois.whoisTool(url, type)
            .then((whoisInfo) => resolve(whoisInfo))
            .catch((err) => reject(err));
        break;
    case "xss":
        wapiti.wapitiTool(url, 'xss')
            .then((wapitiInfo) => resolve(wapitiInfo))
            .catch((err) => reject(err));
        break;
    case "wpscan":
        wordpress.wordpressTool(url, type, proxy)
            .then((wordpressInfo) => resolve(wordpressInfo))
            .catch((err) => reject(err));
        break;

        ...

    case "auto":
        pentest.autoScanTool(url, proxy)
            .then((autoScanInfo) => resolve(autoScanInfo))
            .catch((err) => reject(err));
        break;
    case "custom":
```

```
            custom.customScanTool(url, toolsSelected, proxy)
                .then((autoScanInfo) => resolve(autoScanInfo))
                .catch((err) => reject(err));
            break;
}
```

If a single tool is selected, like *Nmap*, a function like the one that was explained earlier is called. Here, it is more interesting to see the *custom* and the *auto scan* functions. If the *custom scan* is selected, then the function will get as a parameter a list containing the tools that the user selected to run. The appropriate tools will be executed in parallel in order to make the scan as fast as possible, and then all the outputs will be parsed by each tool's parser and will be saved in an object, assigning the tool name as the key for each one.
If the *automatic scan* has been chosen, then the program will follow a certain procedure. The information gathering tools will be executed first. The open ports will be discovered and the services running in each port will be enumerated. After this information is found, common vulnerabilities will be searched for each one of the services. If the system has a web application running, more scans will automatically start. The technologies running on the system will be enumerated, like the version of web servers, libraries, CMS, etc. and if there is CMS like Joomla or WordPress the appropriate scans will be initialized. All the plugins, components and information about the sites will be gathered and the tool will try to find vulnerabilities for each one of the findings. If in the findings, a login page exists, then the tool will crawl the website to find usernames in order to try and bruteforce their credentials. In the meantime, the website is being crawled for other vulnerabilities, like SQLi, XSS, security misconfiguration and others. After all the scans are completed , all the outputs will be parsed by each tool's parser and will be saved in an object, assigning the tool name as the key for each one, as mentioned in the custom scan.

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 it will give the opportunity to the user to make a scan that suits his/her needs. In order to

do that an API has been created with JavaScript. The API will be detailed now, as well as the results of the scans.

First of all, the node app has to be created. This app will be used to specify the routes and the type of requests that the API will accept.

```
const express = require('express');
const app = express();

app.use((req, res, next) => {
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Headers", "Origin, X
        -Request-With, Content-Type, Accept");
    res.setHeader("Access-Control-Allow-Methods", "GET, POST
        , PATCH, DELETE, OPTIONS");
    next();
});

app.post("/api/scan/:tool", (req, res, next) => {
    ...
});
```

This shows how the server works. The *setHeader* functions declare what types of headers are permitted, like what methods the server accepts. Then there is an API function listening for scan requests. The POST method is the method allowed when asking for this service. The method post is the one that separates the behavior depending on the method used to access it. If the method is GET, then this function will do nothing. If the method is POST, it means that the scan form has been filled and that a new scan has to start. Also, the path variable is the route needed to follow in order to get this function. At the end of the path the *:tool* is noticed. This is not a static path, but a dynamic path, where the tool is a variable that changes depending on what tools have been chosen.

When a user opens the web app, the first thing s/he sees is the scan page. The screen has an input where the target will be specified, as well as an input to specify a proxy and then a list with all the tools that the user can select to run, categorized. Under the checklist, there is the *MITRE matrix*[28].

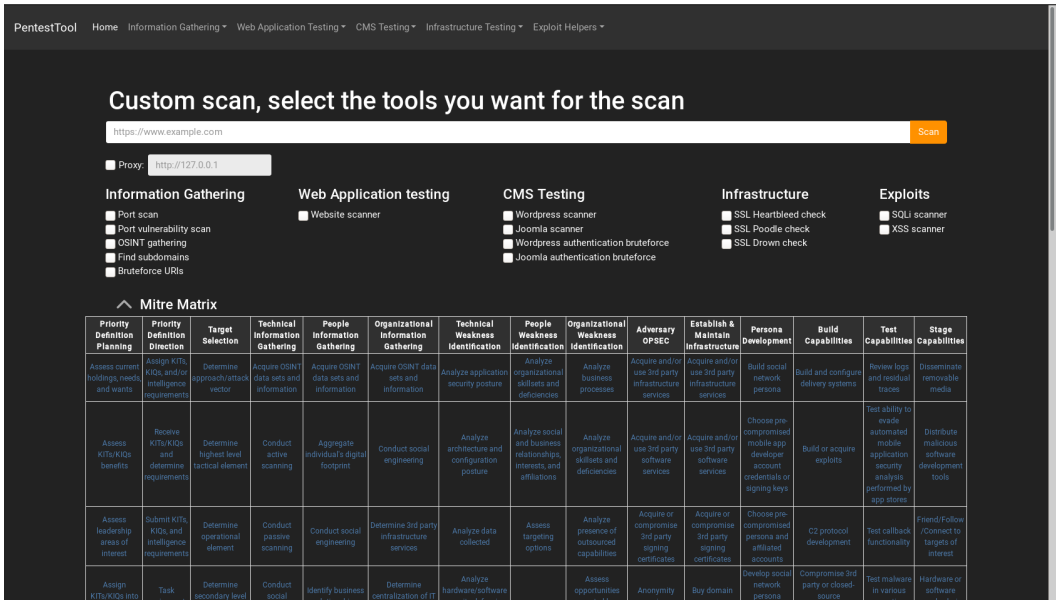


Fig. 4.1: Scan page

When the user selects a tool, the cells, that a scan with this tool will be covered, are highlighted. Every cell is also clickable, and the user can learn more information by clicking one.

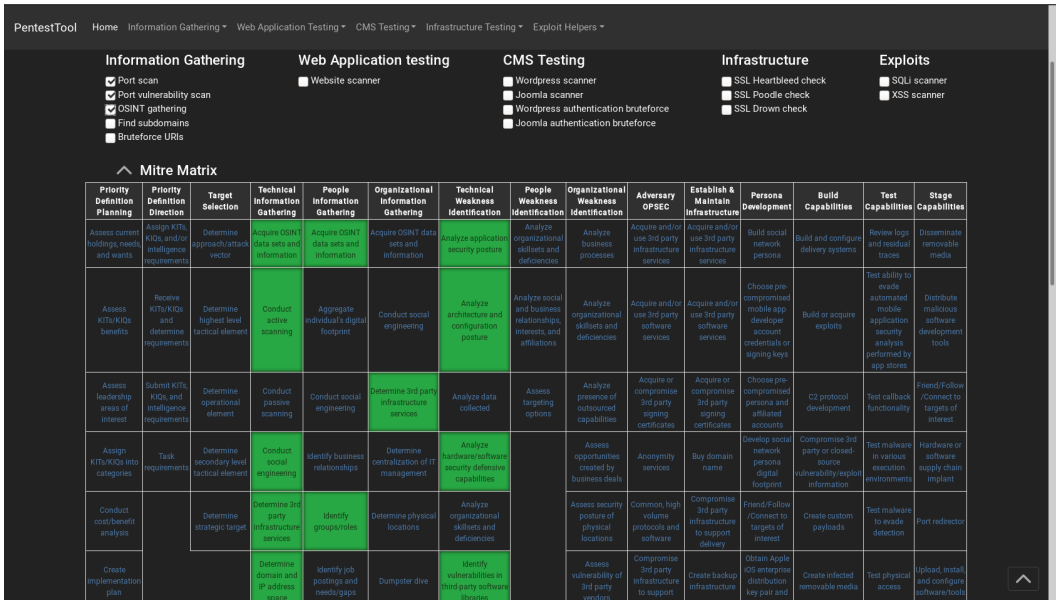


Fig. 4.2: Scan page with selected tools

If the user just sets a target and clicks the Scan button, then the auto scan will begin.

There is also the choice to select each tool separately from the dropdown menus on the navbar on top of the screen. After the user has selected the tools that s/he wants and presses the *Scan* button, then a loading screen appears, informing about the stage of the scan. This is achieved by the use of a *socket*, where the backend emits information every time a tool is started or finished, and the frontend is always listening for messages.

```
getMessage() {
        return Observable.create((observer) => {
            this.socket.on('progress', (message) => {
                observer.next(message);
            });
        });
    }
```

In order to have a more organized code, the frontend is separated into components. To avoid having a huge HTML file reading and showing the result of the scan, each tool has its own component with HTML file and is loaded only when it is necessary.

```
<app-joomla-output *ngIf="mode=='joomscan'">
</app-joomla-output>
<app-wp-output *ngIf="mode=='wpscan'">
</app-wp-output>
<app-nmap-output *ngIf="mode=='nmap'">
</app-nmap-output>
<app-sublist3r-output *ngIf="mode=='sublist3r'">
</app-sublist3r-output>
<app-gobuster-output *ngIf="mode=='gobuster'">
</app-gobuster-output>
<app-whois-output *ngIf="mode=='whois'">
</app-whois-output>
<app-whatweb-output *ngIf="mode=='whatweb'">
</app-whatweb-output>
<app-wapiti-output *ngIf="mode=='wapiti' || mode=='sqli' ||
    mode=='xss'">
</app-wapiti-output>
```

As shown each component is being called only if it was selected.

Let's take a better look at the *Nmap* output HTML file (in the above code it is the app-nmap-output component).

```
<table class="table table-striped table-dark">
    <thead class="thead-dark">
        <tr>
            <th>General Info</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Name: {{ outputObj['name'] }}</td>
        </tr>
        <tr>
            <td>Address: {{ outputObj['address'] }}</td>
        </tr>
        <tr>
            <td>Os: {{ outputObj['os'] }}</td>
        </tr>
    </tbody>
</table>
<tabset>
    <tab heading="Ports" *ngIf="outputObj['ports']">
        <table class="table table-striped table-dark" *ngIf
           ="outputObj['ports'].length > 0">
            <thead class="thead-dark">
                <tr>
                    <th>Port</th>
                    <th>State</th>
                    <th>Service</th>
                    <th>Product</th>
                    <th>Version</th>
                </tr>
            </thead>
            <tbody>
                <tr *ngFor="let port of outputObj['ports']">
                    <td>{{port.port.portid}}</td>
                    <td>{{port.state.state}}</td>
                    <td>{{port.service.name}}</td>
                    <td>{{port.service.product}}</td>
```

```
                    <td>{{port.service.version}} <span *ngIf
                        ="port.service.version">({{port.
                        service.extrainfo}})</span></td>
                </tr>
            </tbody>
        </table>
    </tab>
    <tab heading="Vulnerabilities" *ngIf="
        vulnerabilitiesPorts.length > 0">
        <tabset type="pills">
            <tab heading="Port {{port}}" id="{{port}}" *
                ngFor="let port of vulnerabilitiesPorts">
                <div *ngFor="let key of
                    vulnerabilitiesPortsServices[port] |
                    keyvalue">
                    <table class="table table-striped table-
                        dark" *ngIf="outputObj['
                        vulnerabilities'][port][key.key]">
                        <thead class="thead-dark">
                            <tr>
                                <th>CVE</th>
                                <th>URL</th>
                            </tr>
                        </thead>

                        <tbody>
                            <tr *ngFor="let vuln of
                                outputObj['vulnerabilities'][
                                port][key.key]">
                                <td>{{vuln.cve}}</td>
                                <td> <a href="{{vuln.url
                                    }}">{{vuln.url}}</a></td>
                            </tr>
                        </tbody>
                    </table>
                </div>
            </tab>
        </tabset>
    </tab>
</tabset>
```

The *outputObj* variable is the object that contains the scan result returned from the server. Then, depending on the result, the appropriate HTML elements are created. One interesting thing that can be seen is the usage of if conditions and for loops inside the HTML code. Angular provides these built-in functions to execute code inside the HTML files which makes frontend coding easier. In this case, the *if conditions* are used to check if the output contains some information. If the information exists, then the HTML components will be created, otherwise it will be omitted. The *loop condition* makes it easy to create the same HTML element, using a few lines of code.

```
<tr *ngFor="let port of outputObj['ports']">
```

This line of code iterates over the list of ports and 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, all the necessary information is being displayed in a table layout. For each vulnerability, the CVE id is provided, as well as a link with more information about this vulnerability.

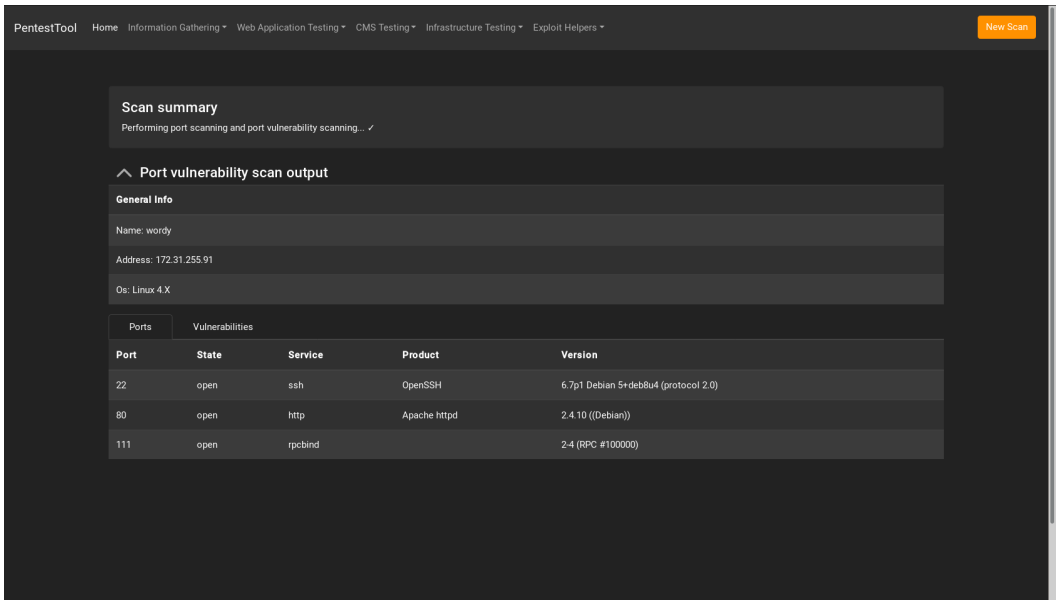


Fig. 4.3: Nmap output

Each tool has an HTML file output like the one described above. Let's take a look at an auto scan to see how the results from more than one tool are displayed.

On the top of the page there is a card with the scan summary. There, all the scans run will be displayed and a check on the right side will indicate

that this scan was successful. Below this card, the result will be displayed. It is categorized by scan in order to have a better image of the scan and not being chaotic. Each category has a dropdown button, so only the information necessary for the user will be displayed, hiding the other information. Also, in order to make it easier and clearer to read, each scan will be categorized in tabs and the user can choose from there what information to show.

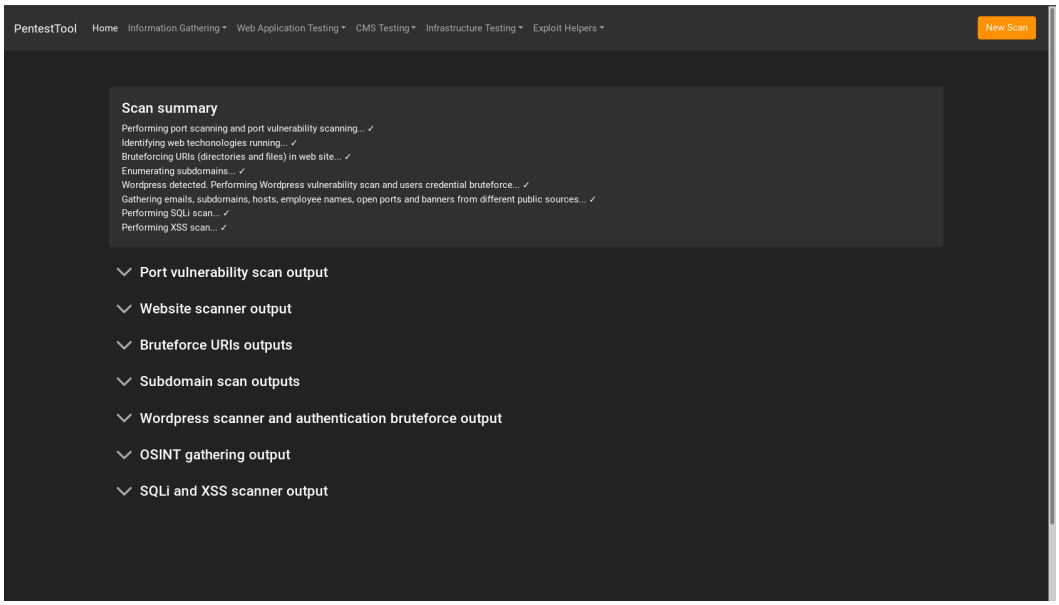


Fig. 4.4: Auto scan output

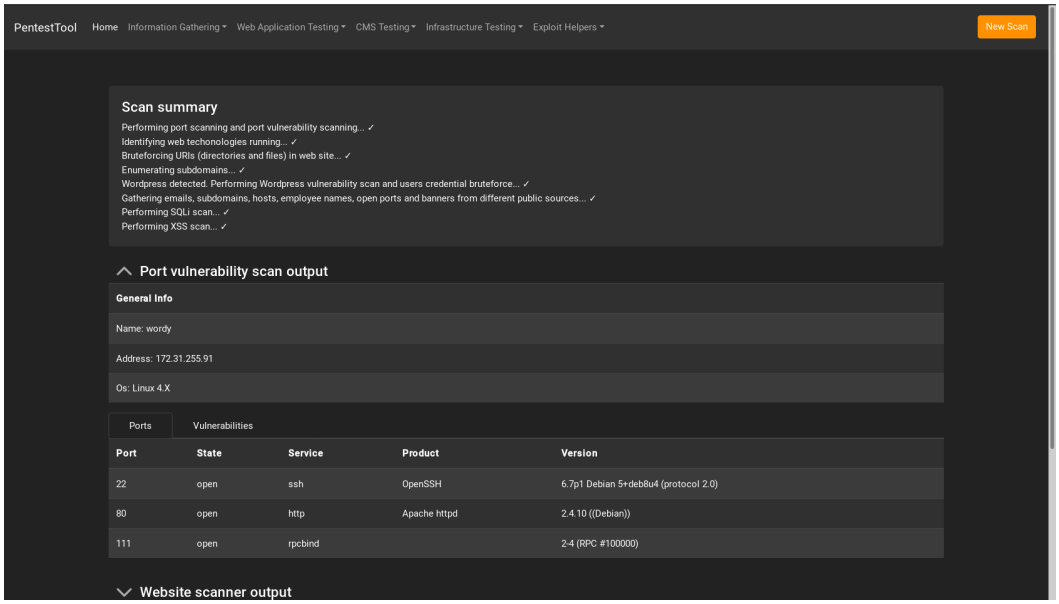


Fig. 4.5: Auto scan output with expanded Nmap output

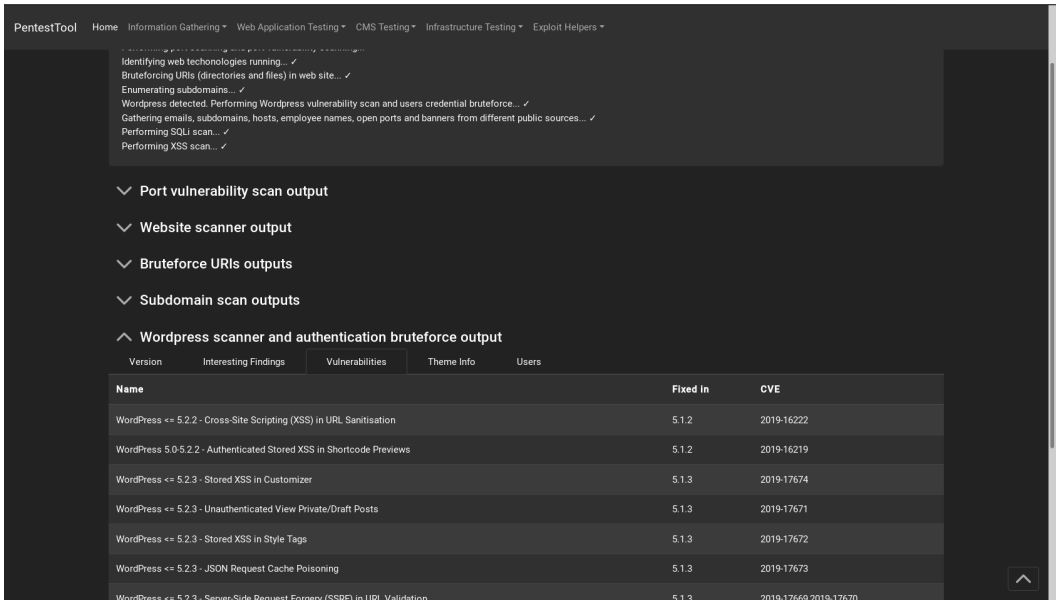


Fig. 4.6: Auto scan output with expanded Wordpress output

Deployment

The system built in this project has a lot of dependencies. Deploying it is not as simple as getting and running the source code. 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, an alternative is being provided. This alternative consists of a virtual machine having the system installed on it beforehand.

The virtual machine chosen to install the tool is a Debian based. This choice is due to the fact that the tools are made for Debian Linux based systems, so there will be no problem with installing and running them. Also a Debian machine comes without any extra packages that would bloat our machine.

Using this setup, the user can mount the Debian OS with the system installed on a virtual machine, and have access to it through a browser.

The creation process of the Debian system is the following. First of all, a clean Debian distribution is downloaded from the official source[29] and install it on a virtual machine. The next step is placing the source code of the system on the virtual machine. Afterward, the code dependencies should be installed.

Most of them are Node.js libraries, which makes them easy to install. Once all the code dependencies are installed, the next step is installing all the tools that the system has to use. If a user wants to add new tools, s/he would have to install them first. A simple solution to this problem would be to use a distribution, such as Kali Linux[30], that already has all these resources preinstalled on them.

It can be challenging to install the tools in the operating system, as most are not in official repositories. This involves downloading the tool's source code from one of the known sources, and then manually resolving each tool's dependencies. Once all the tools on the virtual machine are installed the system is almost ready to run. Last step is to run the web server and then it will be accessible via a web browser locally.

Results

5

This chapter presents the results of this thesis project. This test aims to demonstrate that the system can discover and list vulnerabilities. As it is not legal to test this tool on real machines a virtual machine with known vulnerabilities has been chosen, to make sure that are some vulnerabilities to find.

The virtual machine that was chosen for the test is the *HA: Wordy*[31] from *vulnhub.com*. It is intended for trying penetration testing tools in a safe and legal environment. This virtual machine contains a Linux system hosting a web page with several vulnerabilities and it is a good choice to test the tool created.

First of all, a virtual machine has to be set up. Once it is set up, the system can start analyzing it. Once the testing ends, the results can be analyzed. Firstly, the *OSINT gathering scan*, which searches for hosts and user emails, should be empty cause we used a virtual machine. However, since *theHarvester* searches the internet in order to find this type of information, it could happen that the tool finds hosts and emails that have the same name as the target. These email addresses are not valid ones for our target, but garbage found on the internet. If the email addresses were real, they could be used by an adversary to perform a *Social Engineering attack*. Usually, This king 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. So, this is an important thing to have in mind when performing a test, and it should be included in the report.
Also, the *Subdomains scan* should not produce any output, as we are using a virtual machine. Indeed, the output is empty.

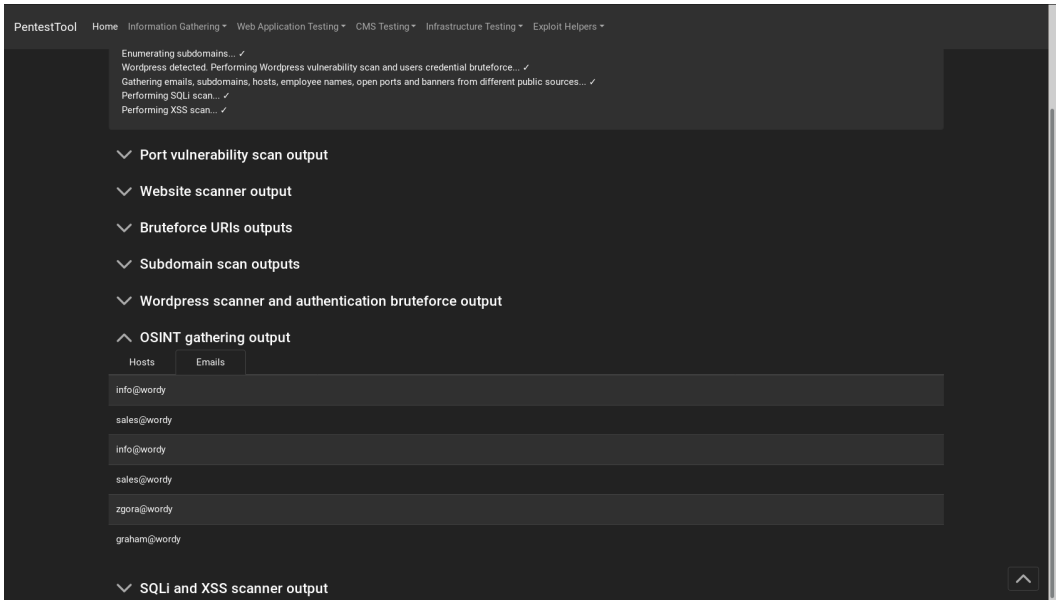


Fig. 5.1: OSINT gathering result

Then, the important results on this test case are in the other sections. Here, the system correctly identifies general information about the machine and a list of open ports can be explored.

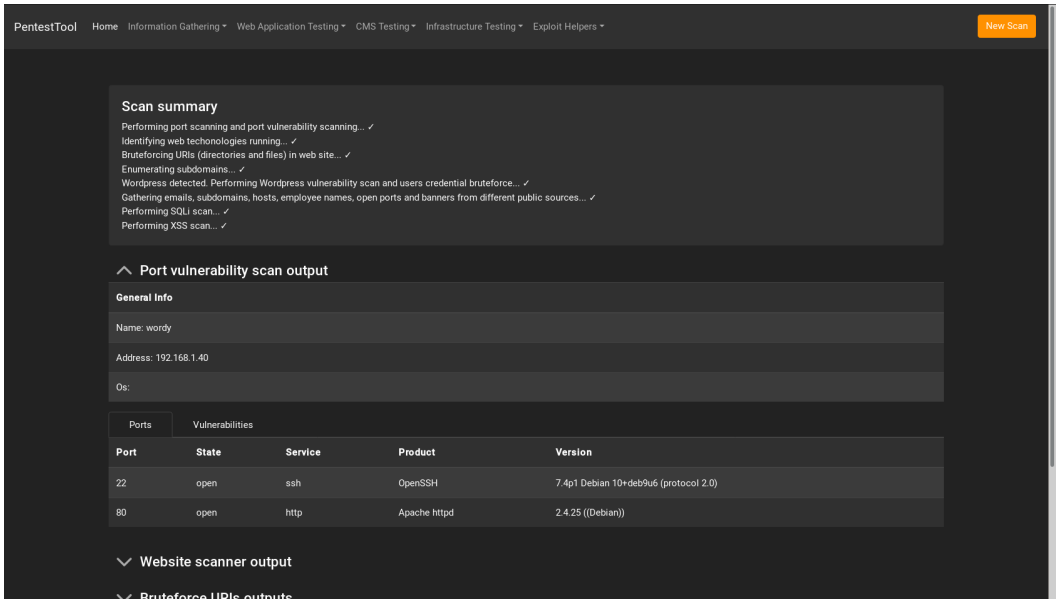


Fig. 5.2: Nmap port scan result

Together with the open ports, it can be seen that the system has obtained some known vulnerabilities. The virtual machine has a lot of known vulnerabilities.

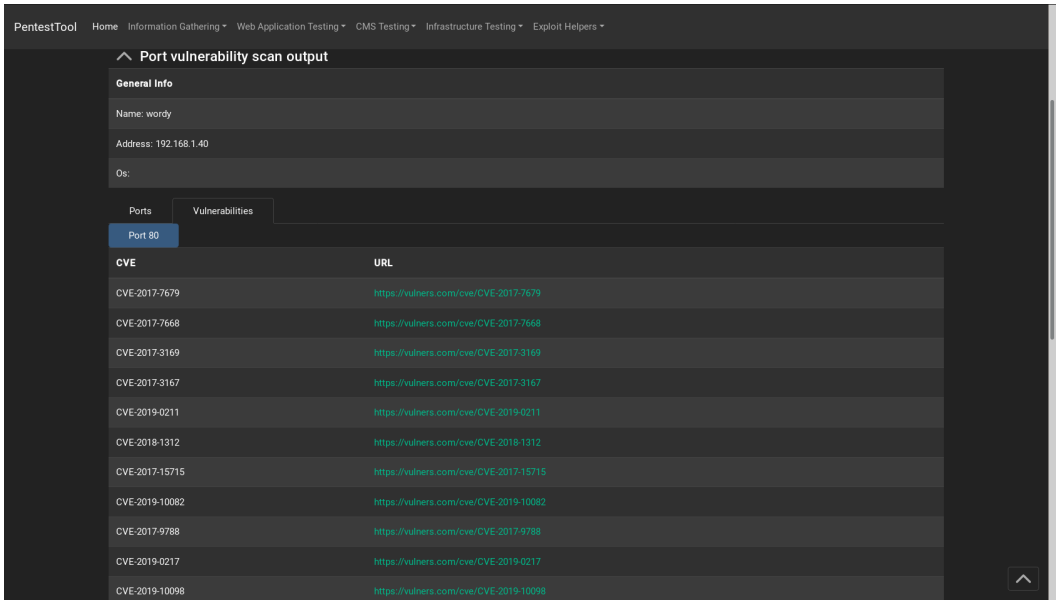


Fig. 5.3: Nmap vulnerability scan result

The website scanner also found interesting information about the machine. The more important one is that the CMS is Wordpress.

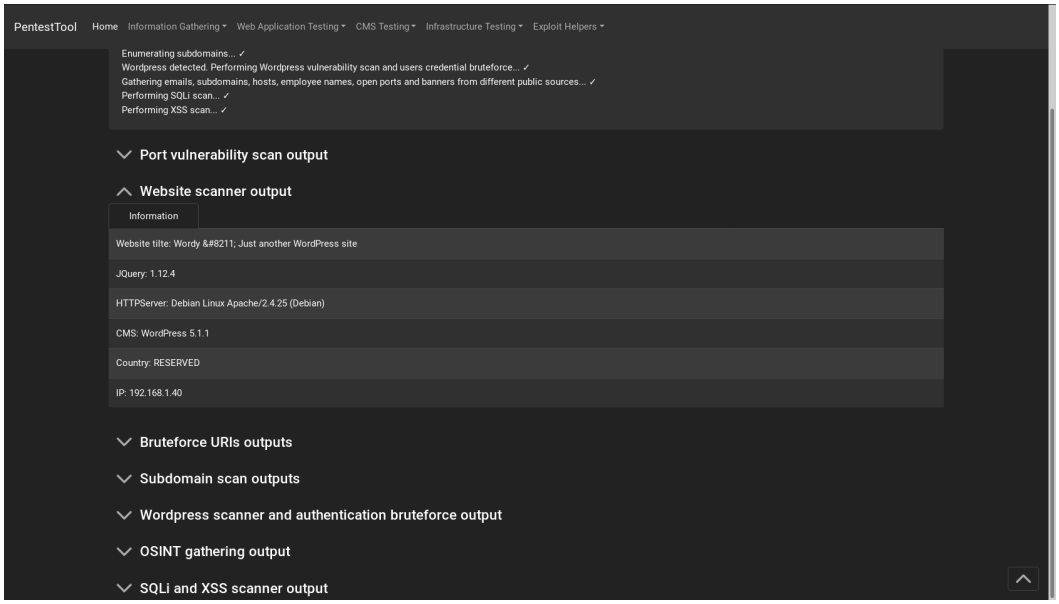


Fig. 5.4: Website scan result

As a result, the *Wordpress scan* should be able to find more stuff. Indeed, it enumerated the website components, identified interesting findings, found information about the theme and also found the users and its passwords.

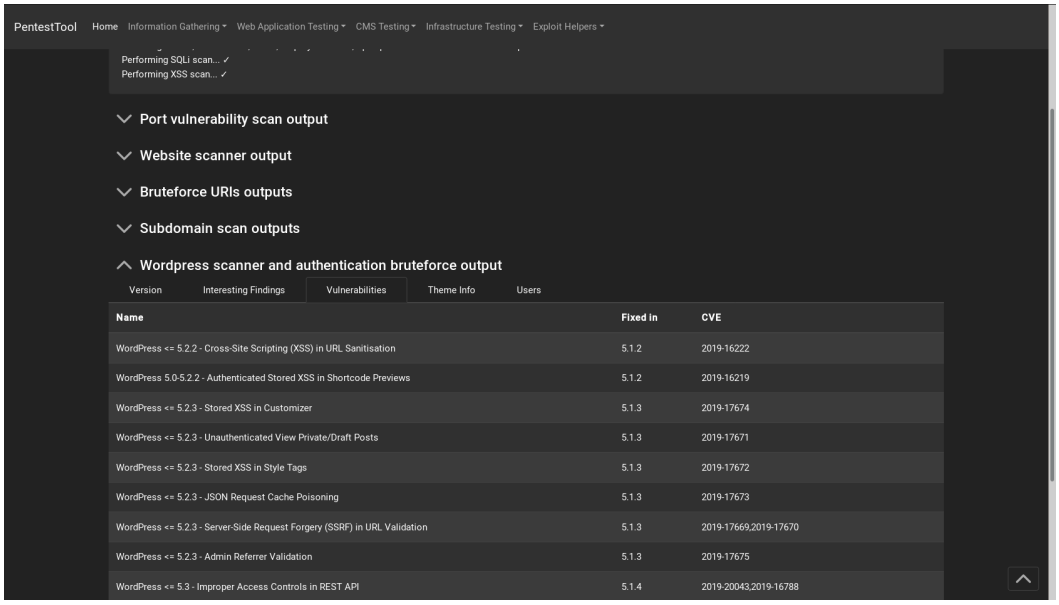


Fig. 5.5: Wordpress scan result

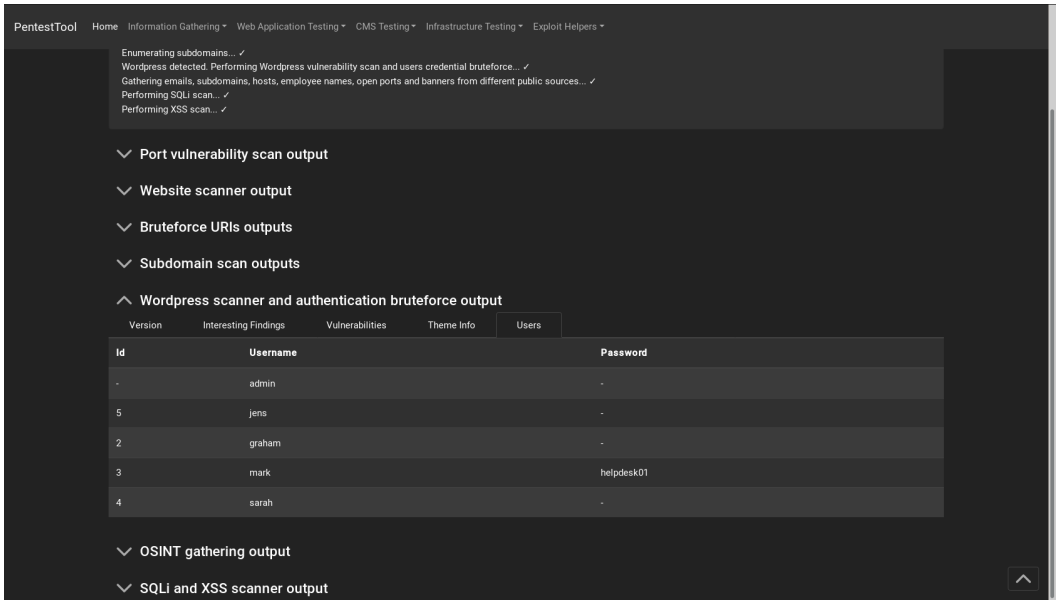


Fig. 5.6: Wordpress credentials bruteforce result

Since the webpage is really simple and does not have any inputs the *XSS* and the *SQLi scans* were not able to find anything. The last one is the *bruteforcing of URIs* that managed to find some, which might be interesting.

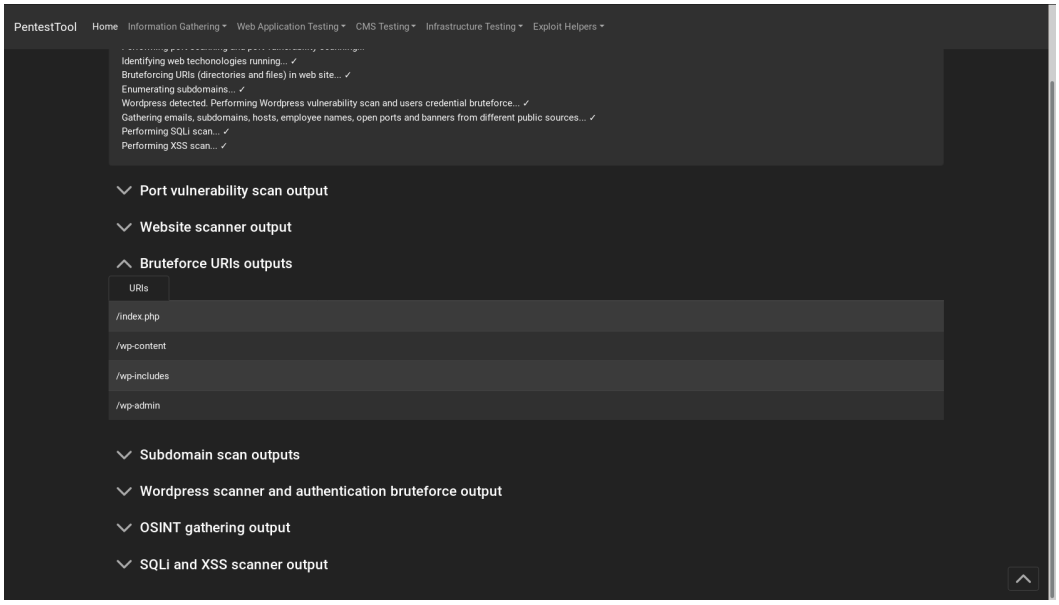


Fig. 5.7: Website URIs bruteforce result

6 Evaluation and Future Work

Once the project has been completed, the system has been built and tested, now is the time to make some conclusions about the work done, and some possible future work. First, the problems faced during the project will be explained, and how they were solved. Thereafter, some possible future work lines will be defined to enhance the work already done, and the knowledge gained from doing this project will also be defined.

6.1 Technical Problems

There were several problems detected when building the system described in this project. The first problem that was detected while researching on automating tools was that it was a really difficult task to intelligently automate the post-exploitation phases of the penetration testing process. There are no APIs available to help solve this problem and at the moment no other security framework is doing so. There were projects launching lots and lots of different exploits, hoping one of them would succeed, but there was not enough time in the lifespan of the project to accomplish this.

This problem was a major one, as it reduced the system's effectiveness. Since no other framework studied achieved this, it was determined that instead the system would focus itself on the first stages of the penetration testing process. 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 aspects of each phase should be covered, but this is a really rough job. Performing a penetration test is not an exact science, and although there are really good guides, it is still heavily reliant on the tester's knowledge and creativity. Furthermore, as new vulnerabilities are found every day and new technologies are developed, the tools that aim to protect and defend these technologies must be constantly updated. So obviously, if a new technology is developed, a new security tool has to be created that tests this technology. This means that it is impossible to create the definitive scanning system, it must be continuously updated. Due to the amount of

security tools available and the fact that a complete framework should be constantly updated, a few tools were selected to represent each stage of the penetration testing process. This may give the impression that the program has few tools but it's a time-consuming task to find a new tool, learn how the tool works properly, and then create a parsing function for that tool.

Finally, a problem was also the fact that there is no clear standard for vulnerabilities. Because different tools use different enumeration systems and establishing a relationship between them is actually complex, it was almost impossible to classify vulnerabilities in a correct and simple way. This problem has been solved in this project by using the *CVE* system as it is being used by most tools used by the system.

6.2 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.

Firstly, the most important thing that this project needs and deserves is to expand the tool collection. The tools chosen for this project are a representative and sufficient collection for showing purposes. Further features should be added in order to make the system valuable to a security expert.

Since adding new resources is an easy process it should not be very hard. The most important aspect of this improvement is the fact that a security expert with experience in penetration testing would really make it easier than a standard developer with no prior knowledge of penetration testing.

Another interesting improvement that could be easily implemented is the following one. Starting from the virtual machine where this tool was created, a cloud-based service could be offered. This would involve updating the user system and adding extra layers of security. This updated product could be offered as a service to companies. This service could be useful for system administrators, because they could have an easy way to test if their system had some security problems. In addition, the system could be automatically activated at fixed time intervals and then receive a notification if an audit results vary 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 were not considered on this project.

Other functionality that could be included in the near future would be the ability of the user to see which tools were executed and where, and whether they discovered anything different or crashed. This would simply mean building a logging system that was available from the user's perspective.

Conclusion

7

In this project, a lot of research about penetration testing has been done. This research has enabled the construction of a system that assists in the first steps of performing penetration tests.

The program that has been developed is a functional one, it does what it was supposed to do, with the only downside being to have fewer tools than the ones it desired. During the development of this project, several problems were discovered, some of them were solved, and some were not, but a lot was learned from them.

Personally, I have learned a lot of things while making this project. Besides the fact that I used and learned to program in *Node.js* and *Angular*, two technologies that are widely used, there were more important things. Learning how the process of the penetration testing works was one of the most interesting things. Lots of tools were discovered, and a lot of practice using some of them was gained. Besides from the penetration testing process, I learned that it is not something easy to automate and even when this is accomplished, a security expert is needed, because most of the vulnerabilities present in web servers or networks need the perspicacity that only a human being can have.

Bibliography

[1] *Committee on National Security Systems. 4009 - National Information Assurance (IA) Glossary*. URL: https://www.dni.gov/files/NCSC/documents/nittf/CNSSI-4009_National_Information_Assurance.pdf (visited on Jan. 28, 2020) (cit. on p. 4).

[2] *ISO/IEC 27001:2013*. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:27001:ed-2:v1:en (visited on Feb. 21, 2020) (cit. on p. 6).

[3] *Framework for Improving Critical Infrastructure Cybersecurity - National Institute of Standards and Technology (version 1.1)*. URL: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf (visited on Feb. 21, 2020) (cit. on p. 6).

[4] *S.2521 - Federal Information Security Modernization Act of 2014*. URL: https://www.congress.gov/bill/113th-congress/senate-bill/2521 (visited on Feb. 21, 2020) (cit. on p. 6).

[5] *Health Insurance Portability and Accountability Act*. URL: https://www.hhs.gov/hipaa/for-professionals/privacy/index.html (visited on Feb. 21, 2020) (cit. on p. 6).

[6] *SARBANES-OXLEY ACT OF 2002*. URL: https://legcounsel.house.gov/Comps/Sarbanes-oxley%20Act%20Of%202002.pdf (visited on Feb. 21, 2020) (cit. on p. 6).

[7]*PCI DSS Quick Reference Guide Understanding the Payment Card Industry Data Security Standard (version 3.2.1)*. URL: https://www.pcisecuritystandards.org/documents/PCI_DSS-QRG-v3_2_1.pdf?agreement=true&time=1582124712479 (visited on Feb. 19, 2020) (cit. on p. 6).

[8]*Nmap (verion 7.60)*. URL: https://nmap.org/ (visited on Jan. 28, 2020) (cit. on pp. 8, 13, 17).

[9]*Golismero Project (version 2.0)*. URL: http://www.golismero.com/ (visited on Jan. 28, 2020) (cit. on p. 11).

[10]*Offensive Web Testing Framework (OWTF) (version 2.1a)*. URL: https://owtf.github.io/ (visited on Jan. 28, 2020) (cit. on p. 11).

[11]*OWASP Testing Guide (version 4.0)*. URL: https://www.owasp.org/images/1/19/OTGv4.pdf (visited on Jan. 28, 2020) (cit. on p. 12).

[12]*theHarvester - Subdomain and emails harvesting (verion 3.1.1dev3)*. URL: https://github.com/laramies/theharvester (visited on Jan. 28, 2020) (cit. on pp. 14, 18).

[13]*Gobuster - Directory/File, DNS and VHost busting tool (verion 3.0.1)*. URL: https://github.com/OJ/gobuster (visited on Jan. 28, 2020) (cit. on p. 17).

[14]*Gobuster Common Wordlist (version 2.3)*. URL: https://github.com/aboul3la/Sublist3r (visited on Jan. 28, 2020) (cit. on p. 17).

[15]*Sublis3r - Fast subdomains enumeration tool for penetration testers (version 1.0)*. URL: https://github.com/daviddias/node-dirbuster/blob/master/lists/directory-list-2.3-medium.txt (visited on Jan. 28, 2020) (cit. on p. 18).

[16]*Wapiti - The Web-Application Vulnerability Scanner (version 3.0.0)*. URL: https://github.com/inc775/wapiti-3.0.0 (visited on Jan. 28, 2020) (cit. on p. 18).

[17]*Nikto web server scanner (version 2)*. URL: https://cirt.net/Nikto2 (visited on Jan. 28, 2020) (cit. on p. 18).

[18]*Nmap POODLE script*. URL: https://nmap.org/nsedoc/scripts/ssl-poodle.html (visited on Jan. 28, 2020) (cit. on p. 19).

[19]*Nmap Heartbleed script*. URL: https://nmap.org/nsedoc/scripts/ssl-heartbleed.html (visited on Jan. 28, 2020) (cit. on p. 19).

[20]*Nmap DROWN script*. URL: https://nmap.org/nsedoc/scripts/sslv2-drown.html (visited on Jan. 28, 2020) (cit. on p. 19).

[21]*Nmp NSE script based on Vulners.com API*. URL: https://github.com/vulnersCom/nmap-vulners (visited on Jan. 28, 2020) (cit. on p. 19).

[22]*WordPress Vulnerability Scanner (version 3.7.4)*. URL: https://wpscan.org (visited on Jan. 28, 2020) (cit. on p. 19).

[23]*Common Passwords Wordlist*. URL: https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt (visited on Jan. 28, 2020) (cit. on p. 19).

[24]*WPScan Vulnerability Database*. URL: https://wpvulndb.com/ (visited on Jan. 28, 2020) (cit. on p. 19).

[25]*OWASP Joomla Vulnerability Scanner Project (version 0.0.7)*. URL: https://wiki.owasp.org/index.php/Category:OWASP_Joomla_Vulnerability_Scanner_Project (visited on Jan. 28, 2020) (cit. on p. 20).

[26]*Metasploit Framework (version 4.17.0)*. URL: https://www.metasploit.com/ (visited on Jan. 28, 2020) (cit. on p. 20).

[27]*db_autopwn plugin of metasploit*. URL: https://github.com/hahwul/metasploit-autopwn (visited on Jan. 28, 2020) (cit. on p. 20).

[28]*MITRE PRE-ATT&CK Matrix*. URL: https://attack.mitre.org/matrices/pre/ (visited on Jan. 28, 2020) (cit. on pp. 23, 34).

[29]*Debian Downloads (vesrion 10.x)*. URL: https://www.debian.org/distrib/ (visited on Feb. 24, 2020) (cit. on p. 41).

[30]*Kali Linux Downloads (version 2020.1)*. URL: https://www.kali.org/downloads/ (visited on Feb. 24, 2020) (cit. on p. 42).

[31]*HA: Wordy VM (version 13 Sep 2019)*. URL: https://www.vulnhub.com/entry/ha-wordy,363/ (visited on Jan. 28, 2020) (cit. on p. 43).