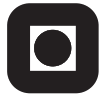

NTNU – Trondheim
Norwegian University of
Science and Technology

Design and implementation of a non-aggressive automated penetration testing tool

An approach to automated penetration testing focusing on stability and integrity for usage in production environments

Fabio Viggiani

Master in Security and Mobile Computing
Submission date: May 2013
Supervisor: Danilo Gligoroski, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem description

The topic of this thesis project is automated penetration testing. A penetration test is a method of evaluating the security of a computer system or network by simulating an attack from malicious outsiders and/or insiders. Several procedures carried out during penetration tests can be easily automated. However, automated tools have many limits and manual testing is required to cover enough test cases to certify the security of a system/network.

The thesis project addresses the problem of automated penetration testing limitations by studying the differences with manual testing. This includes understanding where and how human reasoning is needed and what trade-offs need to be considered when choosing between manual and automated testing procedures. The purpose of the thesis project is to improve the way penetration tests are automated, with a focus on penetration tests performed in production environments, i.e. systems with high availability requirements.

By understanding and following the logic used in manual testing as well as the current level of automation used by security companies, the work carried out during the project will focus on making automated tools behave more efficiently.

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. Automating the process of penetration testing brings some advantages, the main advantage being reduced costs in terms of time and human resources needed to perform the test. Although there exist a number of automated tools to perform the required procedures, many security professionals prefer manual testing. The main reason for this choice is that standard automated tools make use of techniques that might compromise the stability and integrity of the system under test. This is usually not acceptable since the majority of penetration tests are performed in an operating environment with high availability requirements.

The goal of this thesis is to introduce a different approach to penetration testing automation that aims to achieve useful test results *without* the use of techniques that could damage the system under test. By investigating the procedures, challenges, and considerations that are part of the daily work of a professional penetration tester, a tool was designed and implemented to automate this new process of *non-aggressive* testing.

The outcome of this thesis project reveals that this tool is able to provide the same results as standard automated penetration testing procedures. However, in order for the tool to completely avoid using unsafe techniques, (limited) initial access to the system under test is needed.

Acknowledgements

I would like to thank everyone who supported me during this thesis project.

In particular, I would like to thank Marcus Murray for welcoming me in Truesec and giving me the opportunity to learn from highly knowledgeable people and become part of an exciting organization with an amazing philosophy. I am also very grateful to everyone else in Truesec, for their friendliness, openness, and helpfulness.

I would also like to thank Professor Gerald Q. Maguire Jr. for his constant support during this project, and his willingness to share his unlimited knowledge.

Contents

Problem description **i**

1 Introduction **1**

1.1 Problem Statement . . . 2

1.2 Goals of the Thesis . . . 3

1.3 Structure of the Thesis . . . 4

2 Background **5**

2.1 Why perform penetration testing . . . 5

2.2 The penetration testing process . . . 6

2.2.1 Initiation . . . 6

2.2.2 Preparation . . . 7

2.2.3 Testing . . . 7

2.2.3.1 Target identification . . . 7

2.2.3.2 Port scanning . . . 8

2.2.3.3 Enumeration . . . 8

2.2.3.4 Penetration . . . 8

2.2.3.5 Escalation . . . 9

2.2.3.6 Getting interactive . . . 9

2.2.3.7 Pillage . . . 9

2.2.3.8 Clean up . . . 9

2.2.4 Reporting . . . 10

2.3 Tools for penetration testing . . . 10

2.3.1 Metasploit Framework 10
2.3.2 Nmap 12
2.3.3 Wireshark 12
2.3.4 Cain & Abel 13
2.3.5 Medusa 13
2.3.6 Gsecdump and msvctl 13
2.3.7 Burp Suite 14
2.4 Related work 15
2.4.1 Fast-Track Autopwn 15
2.4.2 Core Security's Impact 16
2.4.3 Immunity's Canvas 17
2.4.4 Nessus 17
2.4.5 Summary of related work 18

3 Method 21

4 Safe Penetration Testing 23
4.1 Safe penetration testing techniques 23
4.1.1 Environment observation 23
4.1.2 Hosts and services overview 24
4.1.3 Identification of well-known vulnerabilities 25
4.1.4 Techniques specific to Windows domains 26
4.1.5 Web applications 28
4.1.6 Resource Sharing 28
4.1.7 Default and guessable credentials 29
4.1.8 Remote information gathering 29
4.1.9 Eavesdropping 30
4.1.10 Client-side attacks 30
4.1.11 Extending the scan range 30
4.1.12 Expanding 31
4.2 Comparison with standard automated tools 33

5 Design **35**
5.1 Initial considerations . . . 35
5.2 Approach . . . 36
5.2.1 Structure . . . 37
5.2.2 Platform independence . . . 37
5.2.3 Extensibility . . . 37
5.2.4 Tracking and storage . . . 38
5.2.5 Customer perspective . . . 38
5.2.6 System state change and reproducibility of checks . . . 39
5.3 Architecture . . . 39
5.3.1 Actions . . . 40
5.3.2 Vulnerability Checks . . . 41
5.3.3 Knowledge Base . . . 41
5.3.4 Tracker . . . 43
5.3.5 Decision Engine . . . 43
5.3.6 Report Generator . . . 44
5.3.7 Customer Implementation . . . 44
5.3.8 Penetration Tester GUI . . . 44
5.3.9 Customer GUI . . . 44
5.3.10 Database . . . 45
5.4 Application scenario . . . 45

6 Logic **47**
6.1 The Penetration Test Life Cycle . . . 47
6.2 Individual Steps . . . 50

7 Implementation **53**

8 Results **57**
8.1 State of the application . . . 57
8.2 The testing environment . . . 58

8.2.1 Configuration 59
8.2.2 Test Execution and Results 60
8.3 Analysis of test results 69

9 Conclusions 71
9.1 Conclusion 71
9.2 Future work 72
9.2.1 System State Change 72
9.2.2 Additions 73
9.2.3 Extensibility 73
9.2.4 Risk Definition 73
9.2.5 Efficiency 74
9.2.6 System Virtualization 74
9.3 Required reflections 75

References 77

A Autopwn Results 81

B Nessus Executive Summary 85

List of Figures

5.1 Architecture of the proposed automated penetration testing tool. . 40
5.2 UML class diagram of the main classes in the knowledge base. . . 42
5.3 Scenario showing the different components of the testing application. 45

6.1 Example of an automated penetration test. 49

8.1 Nessus scan policy adopted during the test. 62
8.2 Configuration of the penetration testing tool. 64
8.3 Hosts and services in the knowledge base. 65
8.4 Manually adding an account to the knowledge base 66
8.5 Network shares and programs collected from a remote machine. . 67
8.6 Example of vulnerabilities reported by the tool. 68

List of Tables

6.1 List of steps that the penetration tester can select and execute. . . . 51

8.1 Configuration of the virtual machines used in the test. 59

List of Acronyms and Abbreviations

DCO	Domain Controller
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
GUI	Graphical User Interface
HTTP(S)	HyperText Transfer Protocol (Secure)
IDS	Intrusion Detection System
IP	Internet Protocol
IPC	Inter-Process Communication
IPS	Intrusion Prevention System
IT	Information Technology
JRE	Java Runtime Environment
LDAP	Lightweight Directory Access Protocol
LSA	Local Security Authority
MAC	Media Access Control
MBSA	Microsoft Baseline Security Analyzer
NIC	Network Interface Controller
OS	Operating System
PCI DSS	Payment Card Industry Data Security Standard
PDF	Portable Document Format

PTGUI Penetration Tester Graphical User Interface

RPC Remote Procedure Call

SAM Security Accounts Manager

SMB Server Message Block

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Manager Protocol

SQL Structured Query Language

SSH Secure Shell

TCP Transmission Control Protocol

UML Unified Modelling Language

URL Uniform Resource Locator

VOIP Voice Over Internet Protocol

XML Extensible Markup Language

Chapter 1

Introduction

This thesis focuses on automated penetration testing. A penetration test is a practice used to assess the security of a computer system by acting as a malicious attacker trying to gain access to the system. The outcome of the test reveals whether the system is vulnerable to an attack in a certain scenario. There exist several types of penetration testing, depending on what assets need to be tested (e.g. a network, a single machine, a web application). This thesis will focus on network-based penetration testing, 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, therefore it is desirable to automate them.

The purpose of penetration testing automation is to reduce the costs in terms of time and people needed to perform the test. The time (and human resources) that is saved can be used to provide a broader testing range (see for example [14]). Disadvantages of automation include limited pivoting*, generation of false positives, stability issues, and less intelligent analysis of potentially sensitive data.

The thesis project is carried out at Truesec AB, an IT security company based in Stockholm. An important aspect of this thesis project is to understand Truesec's needs in the context of penetration testing automation and to provide a solution to their current problems.

*A pivot attack consists in compromising one machine and launching a new attack from that machine, to reach other areas of the network.

1.1 Problem Statement

There exist a number of tools that can be used to perform automated penetration testing. Some of these tools are described in section 2.4. In certain situations, these tools perform well and minimize the amount of manual work needed to perform a penetration test. An obvious question is: "Why are automated penetration testing tools not used by all security professionals?". The answer is that most tools that automate the penetration testing process involve the active use of *exploits*[*] (see section 2.2.3.4).

Exploiting a vulnerability can often cause a system or service to crash or fail to perform its legitimate purpose. This makes penetration testing a risky practice, since many tests are performed in a production environment[**]. The stability and integrity of the target system are extremely important in most situations and one cannot simply accept that the automated tools may cause the system to stop functioning. Therefore, penetration testers often prefer manual testing, so that the human tester maintains control of the testing process and thus can assure that only safe techniques are used.

Another reason why penetration testers might prefer not to use automated tools is that the aggressiveness and intrusiveness of such tools are not only dangerous to the reliable operation of the system, but often an aggressive and intrusive action is not even needed to compromise the system. By investigating Truesec's procedures and results it emerged that in most cases the testers do not need to utilize risky exploits as automated tools would. Instead, basic and conceptual mistakes in the system under test allow the attacker to take full control of the system, confirming that aggressive and sophisticated attacks are generally unnecessary.

An important aspect to consider is the value of the outcome of the test from the customer's perspective. A final report, containing the results of the test, includes a list of vulnerabilities and should lead the customer to take a set of mitigation actions in order to increase the security of their system. The standard automated tools target known vulnerabilities that in most cases can all be addressed by the same mitigation technique: implementing a patch management policy. Finding a large number of these vulnerabilities does not increase the value of the testing. In practice, the most critical vulnerabilities are often the more basic ones, and these

[*] Within the context of this thesis, the term *exploit* refers to leveraging a software flaw/bug as opposed to exploiting weaknesses in the system. For instance, attempting a login using default account credentials is not considered an exploit.

[**] The term production environment refers to a phase in the System Development Life Cycle characterized by the need for a very high availability, since a failure in this phase could potentially cause severe damage to the business, as the business may be very dependent upon the correct and timely operation of the system.

can be discovered by less aggressive tests that usually result in a useful feedback to the customer.

Another limitation of standard automated tools is that network dependencies are usually not considered. For example, a cracked password that is used to access one machine could be used to access several different services in the system. Although it is quite straightforward to implement such behaviour, this knowledge is normally not exploited by an automated tool, but is readily exploited by a human tester.

Nowadays, a penetration tester who cares about the integrity and stability of the target system is obliged to conduct most of the work manually, resulting in high costs. The currently available automated tools do not take into consideration the risk to the system under test, thus a different approach to automating penetration testing is needed to address these issues.

1.2 Goals of the Thesis

The main goal of this thesis is to examine in depth how non-aggressive penetration testing is conducted and to design and possibly implement an automated tool that utilizes this approach. The purpose is to evaluate a different approach to automated penetration testing that unlike standard automated tools focuses on *methods and techniques that preserve the integrity and stability of the system under test*. The question that the thesis tries to answer is whether such an approach can provide the same results as standard automated penetration testing procedures, but without making use of techniques that could cause damage to the system under test.

This new tool should be suitable for use in production environments, should automate everything that can be done safely and without risk of service interruption (although it may degrade service for some periods of time), and should identify more dangerous techniques that might be used by the human tester. As mentioned in section 1.1 this non-aggressive approach is often sufficient to compromise the system and can provide the customer with valuable information that could be used to increase the level of security realized by their system. In situations where this non-aggressive approach is not sufficient to compromise the system, the tool would serve as an initial step in the testing process, by eliminating repetitive manual testing. Dependencies in the system under test should also be exploited to extend the compromised area and to safely harvest additional useful information.

Additionally, the tool should provide the customer with an *interactive* report of the testing results. After countermeasures have been adopted by the customer,

it should be possible to reproduce the steps that led to the discovery of a vulnerability in order to verify that the problem has actually been solved. This functionality would allow the customer to easily evaluate the effectiveness of the solutions without the need for a professional security tester, hence this professional would only be needed for the penetration testing itself.

Furthermore, the penetration tester should be able to manually influence the behaviour of the automated tool when needed. It should be possible to select between different levels of automation and manually provide the software with additional information, when available (e.g. IP address of a host in the network that was not discovered automatically). This characteristic would give the penetration testing tool the flexibility needed to fill potential gaps that could be left behind in the automated process.

1.3 Structure of the Thesis

Chapter 1 has described the problem and the specific goals of this thesis. Chapter 2 provides the background necessary to understand the problem and the specific knowledge that the reader will need to understand the rest of this thesis. The standard penetration testing process is explained, common tools are briefly described, and some efforts to automate the testing procedures are analysed. In chapter 3, the methodology and the steps followed during this thesis project are explained. A description of safe penetration testing techniques based on the observation of the tests carried out by Truesec is presented in chapter 4, and the main differences with an aggressive approach are analysed. In chapter 5, the design of the architecture of the new automated tool is presented, while the definition of the logic that follows the safe penetration testing approach is illustrated in chapter 6. The main aspects deriving from the implementation of the new tool are presented in chapter 7. Chapter 8 describes the current state of the implemented application, as well as the virtual scenario that was set up to test the new tool, and the results of these tests. Finally, chapter 9 reports the conclusions of this thesis project, and suggests possible future improvements and extensions to the new tool.

Chapter 2

Background

This chapter gives an overview of the main elements needed to fully understand the rest of this thesis. Section 2.1 presents the main motivations behind the decision of performing a penetration test. In section 2.2 the standard penetration testing process is described. Understanding this part is essential for the development of a tool to automate the process. Several tools, utilities, and frameworks are then presented. Some are part of the common toolkit of a penetration tester (section 2.3), while others (section 2.4) aim at automating the penetration testing process. The automated tools described in this chapter behave in a way that normally does not comply with the idea of non-aggressive testing introduced in this thesis. The main issues will be described in the appropriate sections.

2.1 Why perform 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 may lead to direct financial losses, harm the organization's reputation, trigger fines, etc. With a proper penetration test it is possible to identify security vulnerabilities and then take countermeasures before a real attack takes place.

A penetration test is generally performed by people external to the organization responsible for the system under test. Consequently, the testers operate with a different point of view of the system's resources and may be able to identify issues that were not readily visible to internal operators.

Another reason for performing penetration testing is that it can be a forcing

function to cause the system operator to keep the system up-to-date with respect to the latest vulnerabilities. New bugs and security issues are frequently discovered. An organization may use periodic penetration testing to maintain an updated security level.

The result of a penetration test helps an organization to prioritize their risks. A specific security breach produces a certain damage to the organization. Depending on the severity of the issues that are identified, it is possible to appropriately plan a mitigation strategy with a stronger focus on more critical issues.

Since a penetration test simulates a real attack, it is a good chance for assessing 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 on security awareness and incident handling.

Penetration tests may also be required for security compliance. For example the Payment Card Industry Data Security Standard (PCI DSS) requires penetration testing to be performed at least annually and after any significant upgrade or modification to the system [21].

2.2 The penetration testing process

The purpose of a penetration test is to evaluate the level of exposure of the system under test and to determine whether ways to break into the system exist. In order to properly perform a valuable and legitimate test a few operations need to be performed in addition to the actual testing phase, as described in this section. The process of a professional penetration test can be divided into four main phases: initiation, preparation, testing, and reporting.

2.2.1 Initiation

The initiation phase involves an initial discussion with the customer (owner of the system under test) 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 that the testers are allowed to take, and the test scheduling. A team is set up and (emergency) contact information is exchanged.

2.2.2 Preparation

Before starting the actual penetration testing, a preparation takes place according to the agreement established during the initiation phase. If more than one penetration tester is involved in the testing, then the work is organized and divided within the team. Depending on the tasks that need to be executed, tools are chosen and configured accordingly. This phase requires the penetration testers to take into consideration the integrity and stability of the system under test. As discussed in section 1.1 this is a critical aspect when establishing what actions will be taken during the test.

2.2.3 Testing

This phase contains the actual testing and closely resembles the *hacking process**. Every action taken during the testing phase must be logged so that it will be possible to analyse the history in case unexpected situations arise. The communication with the customer is also important in specific situations where the penetration tester needs the approval of the system's owner before taking an action. The testing process involves several different steps, described in the following sections. Some of these steps are repeated over time when new pieces of information are gathered that allow the tester to fill in earlier gaps or to explore new areas of the system under test.

2.2.3.1 Target identification

Target identification consists in gathering information on 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. Identifying the target is essential, especially in the context of an external penetration test, i.e. when the tester has no initial access to internal resources. Useful information can be discovered with a number of different techniques, such as probing a website, gathering information from search engines, or performing social engineering [15].

---

*In the context of this thesis, the term *hacking process* refers to the steps taken by an attacker who is not authorized to access the system and whose goals are usually of a malicious nature.

2.2.3.2 Port scanning

Port scanning is the first part of the penetration testing process that involves an active interaction with the system under test. It consists of probing the network for the purpose of finding which hosts are present, what ports are open, and what services are running. A tool is usually used to perform this task (see for example nmap - described in section 2.3.2).

2.2.3.3 Enumeration

Once the penetration tester has built an overview of the hosts and services that are part of the system under test, it is time to identify those that are most likely to be vulnerable. Enumeration consists of gathering information about the services in the system in addition to the results of the port scan. Examples of such information are the version of the service in use, well-known vulnerabilities, password lockout policy for a specific service, etc. This knowledge allows the tester to identify the weakest point(s). The experience of the tester is of great help in this phase, although tools can also be used to support the tester.

2.2.3.4 Penetration

Penetration is the act of exploiting a weakness that has been identified in the system under test. As described in [8] an *exploit* is the means by which a penetration tester (or an attacker) takes advantage of a flaw within the system, resulting in a behaviour 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 1 the tester is usually not allowed to perform actions potentially dangerous to the stability and integrity of the system under test, hence the concept of non-aggressive penetration testing described in this thesis.

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.

2.2.3.5 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 in further exploiting a resource to increase the influence of the tester on the compromised machine.

2.2.3.6 Getting interactive

The fact that a host in the system under test is compromised does not necessarily mean that it is easy to control it. An interaction mechanism is needed for the penetration tester to perform operations on the compromised machine in the same way an administrator would. Sometimes, exploits directly provide the tester with an interactive interface (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.

2.2.3.7 Pillage

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

2.2.3.8 Clean up

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 clean up phase is to avoid introducing additional vulnerabilities in the system under test. The goal of this phase is different from a hacker's perspective. 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.

2.2.4 Reporting

The final phase of a penetration test is to report the results of the test. The report includes a description of the vulnerabilities that were encountered during the test, how it was possible to exploit them and suggestions on how they could be fixed. From the customer's perspective, simply receiving a list of the issues that were identified does not provide much value. Therefore, it is often preferred to organize a workshop where the content of the report can be discussed and the penetration testers can clearly explain to the customer what really happened during the penetration test. Another advantage of a follow-up workshop is that the *severity* of the vulnerabilities that were found can be discussed and defined together with the customer. The severity indicates the level of danger of a vulnerability and it is based on two factors: the likelihood that a vulnerability will be exploited and the damage that a possible exploitation may have on the business. The penetration tester only knows the technical severity, but the customer should estimate the consequences that a specific security breach would have on their business.

2.3 Tools for penetration testing

This section presents some of the most common tools used by security professionals when performing penetration testing. These tools help the testers perform specific tasks and are therefore not considered to be automated tools. During the design phase of this thesis project, the inclusion of some of these tools was considered as part of the automated application proposed in this thesis.

2.3.1 Metasploit Framework

Metasploit [7] [8] is an *exploitation* framework. It provides several tools, utilities, and scripts to execute and/or develop exploits against targeted remote machines. Referencing the penetration testing process explained in section 2.2, the Metasploit framework is usually used to cover the phases of Penetration, Escalation, and Getting interactive; although tools are included in the framework to assist the penetration tester during other phases as well.

The Metasploit framework is an open-source software development kit. This framework was acquired by Rapid7 in 2009 and served as basis for two commercial editions released later: Metasploit Express and Metasploit Pro. A statement on Rapid7's website illustrates the commitment of the company to always maintaining the Metasploit framework as open-source software and that it will be free to download and use. This will be an important consideration when deciding how independent from other software the tool designed during this thesis project should be.

When using Metasploit to exploit a remote target, a penetration tester needs to collect information about the target in advance. This information can be collected using tools (e.g. Nmap) and it is used to identify potential vulnerabilities in the target machine. Metasploit includes a large number of exploits for several different applications, protocols, and operating systems that can be launched once a vulnerable service has been spotted. It is up to the penetration tester to select (manually or with the help of software tools) the appropriate exploit matching the vulnerability that has been identified. When executed, the exploit will leverage the vulnerability of the target machine and, if successful, will allow the tester to remotely execute arbitrary code. The code that is executed on the target machine is called a *payload*. Several payloads are included in the Metasploit framework. A suitable payload needs to be selected depending on the desired result and on the environment (e.g. OS type and version of the remote machine). For instance, a payload can provide the tester with a *reverse shell*, i.e. an interface to the remote machine's kernel that connects back to the tester's machine allowing the tester to interactively execute commands on the compromised machine. The exploit must be configured to use a certain payload before being launched. In order to avoid detection from Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), the payload can be *encoded* in one of several possible forms before the exploit is executed. This entire procedure can be automated using an appropriate tool (see section 2.4.1).

The possibility of combining any payload with any exploit gives Metasploit flexibility and modularity, extending the scope of the framework and facilitating payload and exploit development. Being open-source, it is possible for anyone to write new exploits and add payloads to the framework. These characteristics have made Metasploit a popular penetration testing framework and it is used by a large number of security professionals.

2.3.2 Nmap

Nmap [9] [10] is an open-source utility used to create a map of a computer network and to provide a list of hosts and services that exist in the network. Nmap is often used by professionals for performing security auditing, since the scanning of a network might reveal vulnerable services or configurations. However, this utility can also be used for tasks such as network monitoring and inventory. Its scalability properties make it an excellent tool for scanning large networks.

Nmap injects specially crafted packets as network traffic and by analysing the responses to these packets it derives several pieces of information about the network, such as what hosts are present, what services are running on those machines, the operating system installed, whether firewalls are in use, etc. Nmap is a powerful utility that gives the user great flexibility (with over 100 command-line options). This results in a rather complex program with several different tasks executed behind the scenes. As described by Gordon "Fyodor" Lyon in [10] the phases that take place during a normal scanning process are the following:

1. Script pre-scanning
2. Target enumeration
3. Host discovery
4. Reverse-DNS resolution
5. Port scanning
6. Version detection
7. OS detection
8. Traceroute
9. Script scanning
10. Output
11. Script post-scanning

Selected parts of this process will be used later in this thesis to gain knowledge about the environment during the initial phase of the automated penetration test.

2.3.3 Wireshark

Wireshark [11, 12] is an open-source network packet analyser, i.e. software that captures packets from a network and tries to display their contents. Wireshark can be used for several purposes, such as learning network protocols, debugging new protocols, examining network problems, and identifying security issues.

From a penetration tester's perspective, listening to network traffic can provide important information revealing security vulnerabilities or serve as a basis for different types of attacks. For example, clear-text data sent from web forms or services to applications can contain sensitive data or reveal a lack of input validation. Wireshark can also be used to analyse the protocols utilized by different machines as they communicate via the network in order to find inconsistencies that can be exploited.

2.3.4 Cain & Abel

Cain & Abel (or simply Cain) [2] is a password recovery tool for Microsoft Windows. It utilizes several techniques to recover secrets, including eavesdropping, brute-forcing, cached passwords, recording VOIP conversations, and cryptanalysis techniques. Cain targets weaknesses in protocol standards, authentication methods, and caching mechanisms.

2.3.5 Medusa

Medusa [3] is an open-source login brute-forcer based on Linux. The characteristics that distinguish this software from other brute-forcing tools is that Medusa is flexible, modular, and allows thread-based parallel testing.

The flexibility comes from the fact that the target information can be adjusted in detail. The user can specify the target host(s), account(s), and password(s) separately. It is therefore possible to flexibly refine the attack depending on the lockout policy (the rules that define the number of tries allowed before blocking an account) and on the password policy (rules to create a password compliant with the minimum strength requirements).

Medusa supports several authentication mechanisms organized in modules. Modules allow users to easily extend the list of services that Medusa can target without changing the core of the application. Examples of services that are supported are MS-SQL, HTTP, SSH, SMB, and telnet.

2.3.6 Gsecdump and msvctl

Gsecdump [4] and msvctl [5] are two tools that can be used together to escalate privileges on a compromised Microsoft Windows machine. The purpose of the escalation phase in a penetration test was explained in section 2.2.3.5. The technique that these two tools use is referred to as *pass the hash*. It consists in

extracting *password hashes* of users (both local and domain users) that have an *active logon session* and use these password hashes to run commands with these users' privileges.

Gsecdump is used for the first phase of this escalation technique: collecting password hashes. In particular, it extracts *non-salted* password hashes from the Security Accounts Manager (SAM) file. It also extracts Local Security Authority (LSA) secrets. Gsecdump works with both x86 and x64 architectures for Microsoft Windows. One limitation of this tool is that it requires local admin privileges to be able to extract the desired data. Therefore, a local admin user account must be compromised before escalating privileges using this tools. However, as shown by the results of the penetration tests performed by Truesec, it is common to encounter Windows domains using poorly designed account management mechanisms that, for example, assign local admin privileges to every user logged on to a machine. In such a situation, compromising *any* user account would lead to an easy privilege escalation.

Msvctl is the tool that performs the actual escalation. It injects the password hash in a process specified by the user, allowing the user to run that process with the privileges of the target account. Since the possible accounts that can be targeted include domain accounts, msvctl may be able to run processes with influence on the entire network. This means that once any machine in the network is compromised, an attacker can use this access and the information gained to launch further attacks directed against more critical network entities (e.g. the Domain Controller).

The tools introduced in this section provide a method to escalate privileges that does not require the passwords to be cracked and it is therefore considerably faster than a brute-force attack against a (possibly salted) password hash.

2.3.7 Burp Suite

Burp Suite [13] is a Java application designed to perform security testing of web applications. The suite consists of different components, described briefly in this section, that together constitute an integrated platform for web application security assessment.

Burp Proxy is the central component of the suite. It works as a web proxy server that lies as a man-in-the-middle between the penetration tester's web browser and the web servers under test. The proxy allows the tester to intercept, analyse, and manipulate the HTTP/HTTPS traffic that flows in both directions. The web browser used by the penetration tester must be configured to use the

proxy.

Burp Spider is a tool for web application crawling, i.e. browsing a web application in an automated and methodical manner with the purpose of building a complete map of the application. Starting from a user-provided URL, the Spider searches for every reference on that page (e.g. links and images), requests them and proceed recursively. This behaviour produces a map of the application containing all resources that are directly or indirectly referenced within the web application.

Burp Scanner is used to find security vulnerabilities in a web application. The scanning performed by this tool can be either passive or active. When passively scanning a web application, the Scanner simply analyses all responses received by the web server and tries to deduce vulnerabilities. During active scanning, specially crafted requests are sent to the web server and responses are inspected to recognize vulnerabilities.

Burp Intruder allows the user to customize HTTP/HTTPS requests for the purpose of launching automated attacks. A *base request* is initially prepared. The user can then specify how the base request will change in order to generate modified versions of the original request. This is an extremely handy functionality for testing for vulnerabilities such as SQL injection, Cross Site Scripting, and brute-force guessing of web directories.

Burp Repeater is a simple tool used to reissue individual HTTP requests multiple times. It works similarly to Burp Intruder, but it provides less flexibility and it is normally used to perform *stress tests*.

2.4 Related work

This section presents some efforts to automate the penetration testing process. The tools described will provide the basis for understanding the limitations of today's automated procedures for penetration testing in the context of production environments.

2.4.1 Fast-Track Autopwn

Fast-Track [8] is a python-based open-source project based on the Metasploit framework providing penetration testers with automated tools to identify and exploit vulnerabilities in a network. Fast-Track extends Metasploit with additional features and is composed of several tools concerned with different aspects of

the penetration test: MSSQL server attacks, SQL injection, Metasploit Autopwn Automation, Mass Client Side attacks, additional exploits not included in the Metasploit framework, and Payload generation.

Within the context of this thesis, the most interesting Fast-Track tool is Metasploit Autopwn Automation (or simply Autopwn). Autopwn aims at automating the procedure that a penetration tester would follow when trying to exploit a remote network using the Metasploit framework. As explained in section 2.3.1, this procedure consists of gathering information about the target(s), identify a vulnerability, select an exploit to leverage that vulnerability, configure a payload to be executed in case of successful exploitation, optionally encode the payload to avoid detection, and finally launch the exploit. Autopwn automates the entire procedure by running an nmap scan (see section 2.3.2) and, based on the scan result, unleashing every possible exploit that matches the characteristics of the target machine.

Autopwn provides an extremely high level of automation and depending on the quality of the exploit database, it can be very effective. However, there is an obvious drawback. This tool is excessively visible (i.e., detectable) and aggressive and it is therefore likely that the system under test will be subjected to temporary or permanent failures.

2.4.2 Core Security's Impact

Core Security's Impact is a commercial application for automated penetration testing developed by Core Security Technologies. This GUI-based software aims at easing the job of corporate security administrators who want to perform penetration testing on their systems. Core Impact automates all phases of a penetration test, from information gathering to report generation.

The basic concept corresponds to the procedure used by the majority of automated penetration testing tools: the software scans a range of hosts in a network, looking for vulnerabilities for which it has suitable exploits. Additionally, after the vulnerability exploitation, Core Impact is able to install *agents* on the compromised machines that provide different levels of remote access. These active agents can launch additional tests from the new location, allowing the penetration tester to move from host to host within the system under test.

The exploits used by Core Impact are constantly updated and available to customers who purchased the product. The available exploit database contains a large number of up-to-date exploits giving Core Impact the ability to test a wide range of systems. The exploits and tools used by Core Impact are written in

Python and compiled at run-time. This gives experienced penetration testers the possibility to extend the application with their own custom additions. Another advantage of using this product is that it provides a wealth of information once the test is finished, including a summary of all activities and modules executed, details of every tested host in the network, and a description of the identified vulnerabilities.

Disadvantages of Core Impact include its high price and the lack of a command line interface. Core Impact presents the same issues that were mentioned in section 1.1, that derive from the active use of exploits. This characteristic is common to the majority of automated tools for penetration testing and will be analysed in section 2.4.5.

2.4.3 Immunity's Canvas

Immunity's Canvas is a commercial vulnerability exploitation tool developed by Immunity Inc. This software follows the same approach as Core Impact's, but provides a lower level of automation and lacks features such as pivoting and automated reporting. Advantages compared to Core Impact are a considerably lower price and the inclusion of a command line interface.

Canvas does not provide fully automated procedures for penetration testing. Instead, it is a support tool for penetration testers who can use it to gather information about the system under test and select appropriate exploits and actions among those provided by Canvas. Although Canvas is able to automate parts of the penetration testing process, the user of this software is required to have a substantial knowledge about penetration testing and system security.

In the same way as the other automated penetration testing tools described so far in this section, the use of exploits threats the stability and integrity of the system under test and many penetration testers are therefore reluctant to use this tool in a production environment.

2.4.4 Nessus

Nessus is a proprietary vulnerability scanner developed by Tenable Network Security. As opposed to the other tools described in this section, Nessus only aims to discover vulnerabilities on systems and does not exploit them. The software scans the specified hosts in the system under test and tries to match the information from the scan result with an extensive and constantly updated vulnerability database.

From the point of view of the stability and integrity of the system under test, the fact that Nessus does not exploit the vulnerabilities gives the penetration tester more confidence in the use of this tool. However, host probing can be a risky practice itself, depending on the technique used. An advantage of Nessus is that the user is able to select which types of scans the application is allowed to run. Therefore, the penetration tester can adjust the behaviour of the scanner and assure that only safe techniques are used. Nessus can be extended with additional plug-ins or custom scripts, thus the penetration tester can adapt this tool to the specific system under test.

If configured properly, Nessus may be suitable for use in production environments. However, the approach of identifying security issues based on a database of well-known vulnerabilities limits this tool to detection of only well-known issues.

2.4.5 Summary of related work

By studying the behaviour of the tools described in this section, a common approach to automated penetration testing emerged. The procedure followed by these tools consists of three main phases:

1. **scan** the hosts in the system under test in order to gather as much information as possible;
2. **identify vulnerabilities** by matching the results of the scan with entries in a vulnerability database; and
3. **exploit** a vulnerability to gain access to a certain resource.

Depending on the specific tool other phases may take place, however the basic behaviour always reflects the three steps mentioned above. In chapter 4 these tools' procedures will be compared with the actions manually performed by a penetration tester in a production environment, with the goal of understanding the differences that make manual testing the preferred solution in such environments.

As explained in section 1.1 the uncontrolled use of exploits is likely to cause service interruption in the system under test, therefore automated tools following this approach are not suitable for use in production environments. Moreover, security issues identified by matching system properties with well-known vulnerabilities often do not add substantial value to the results of the penetration test, since the majority of these issues can be fixed with the same solution, that is implementing a patch management mechanism in order to maintain all software in the system up-to-date. However, there might be an added value in detecting such

vulnerabilities. When the issue does not concern a specific product for which a relatively straightforward patching mechanism is possible, but instead involves generic components such as protocols and libraries, then the issue is not as easily fixable, and the detection of such an issue represents important feedback for the customer.

Chapter 3

Method

This chapter describes the different steps and the methodology followed during this thesis project. As explained in the previous chapters, the first task was to identify an open problem at Truesec and define a set of goals expected to solve this problem. A study of the generic penetration testing process and existing tools related to this project has then been carried out, in order to develop a solid background to be used as a starting point for the rest of the project.

An essential step was to understand the procedures that take place during a *safe* penetration test performed by Truesec employees. This step is extremely important because it is used to derive the differences between a safe penetration test and the standard aggressive penetration tests performed by most automated tools. In order to gather the necessary information, professional penetration testers at Truesec have been observed during their work. The procedures, the tools used, and the reasoning behind every decision have been noted. The observed process and a comparison with standard automated tools is presented in chapter 4.

In order to implement in software the concepts derived from the analysis presented in chapter 4, an architecture for a new tool has been designed (and is described in chapter 5) and the logic compliant with safe penetration testing procedures has been defined (and described in chapter 6). The application implementation is described in chapter 7.

In order to verify that the non-aggressive approach followed by the implemented tool is feasible in a real-world scenario, a virtual network was set up and the tool was used to assess the security of this virtual system. Other tools were used as well, and considerations were made based on the results of the different tools. This is described in chapter 8.

Chapter 4

Safe Penetration Testing

This chapter describes some of the techniques used during penetration testing of systems in production environments that are considered to be safe with respect to the stability and the integrity of the system under test. The methods illustrated here do not include sophisticated techniques corresponding to attacks that are unlikely to occur, but rather focus on compromising the system by taking advantage of more basic and conceptual mistakes. As mentioned in chapter 1, a large number of systems are vulnerable to these kinds of vulnerabilities, and the techniques explained in this chapter are usually sufficient to take over the entire system under test. However, more advanced methods are also required for a thorough penetration test.

4.1 Safe penetration testing techniques

The safe penetration test approach consists of gaining a form of (limited) access to one or more resources in the system and using that access to harvest (additional) sensitive information in a recursive way. Every step described in this section uses methods that generally do not harm the system under test. The order of these steps may vary depending on several factors, such as the type of testing environment and the priority of the different resources.

4.1.1 Environment observation

The first step in a safe penetration test consists in gathering as much information as possible about the testing environment. Depending on how the scope is defined (i.e. what resources are to be tested) and the amount of

information that the tester is given at the beginning of the penetration test, it may be valuable to perform a research of publicly available information about the organization. This information, usually discovered via the web, may reveal details about the infrastructure to be tested (IP addresses, DNS domain(s), etc.) along with the names and e-mail addresses of key personnel.

Assuming that the tester is able to physically connect to the network to be examined, the initial phase includes collecting all the information that is readily available, such as:

- local IP address(es);
- network subnet mask;
- network address space;
- domain names;
- default gateway; and
- address of DNS servers.

This knowledge allows the penetration tester to derive an initial overview of the network and to plan and appropriately configure the techniques and tools to be used in the following phases of the penetration test.

A crucial piece of information that needs to be derived during this initial phase is the platform that the system under test is based on (e.g. a Microsoft Windows Domain or a UNIX/Linux-based infrastructure). This knowledge determines some of the testing techniques that the tester will be able to use.

4.1.2 Hosts and services overview

In order to obtain an overview of the hosts and services that are part of the system under test, a limited scan of the network is usually performed at the beginning of the test. Some techniques do not require this step to be performed first, but a scan is often the preferred starting point of a penetration tester, because acquiring an overview of the system usually accelerates the rest of the process.

There exist several different types of scanning tools, such as Nmap [9], that can be applied. The different types of scans differ in the technique that is used. These differences will determine aspects such as speed of execution; probability of detection; and filtering, effectiveness, level of intrusiveness, and local privileges needed to run the scan. Within the context of a safe penetration test, the most important aspect to consider is the intrusiveness of the scan, as this determines the probability of a host or service experiencing a failure.

A common choice for non-aggressive scanning is *TCP SYN scanning* [10]. This technique, also called *half-open* scanning, consists in beginning to establish a TCP connection [16] by sending a SYN message to the target host. Depending on the response received from the remote machine (either a SYN/ACK or a RST), the scanning tool is able to determine whether the targeted port is open and that there is a process listening to this port. If there is a positive response from the host (SYN/ACK), then the scanning tool interrupts the connection establishment by sending a RST message. It is important to send this RST as otherwise this scanning could cause a system crash (due to the exhaustion of memory due to the transport control block which the system allocated when the SYN was received) or could reduce the ability of the system to serve legitimate users (due to limits on number of outstanding TCP open request queues and other resources that are allocated when the SYN is received). Hence the name *half-open* scanning.

A simple port scan such as a TCP SYN scan provides information about what TCP ports are open on which hosts. By using a database containing information about well-known services, nmap or another tool using a TCP SYN scan may be able to *guess* what type of service is running on a certain port (e.g. SMTP at TCP port 25, HTTP at TCP port 80 or 8080, or DNS at TCP port 53). Although these guesses are often correct, a penetration tester should not rely on such assumptions. More advanced techniques, such as service enumeration and version detection, are needed for a more accurate scan. In delicate situations, the use of these techniques may not be possible due to the risk of service failures. A simple scan, however, gives the penetration tester a sufficient overview to start building a map of the system under test.

In cases where well-known ports for common services are found to be open, the penetration tester may decide to dig a little deeper and execute a *version detection* scan. This technique is more aggressive than TCP SYN scanning and it is used only when the tester is rather confident that the targeted service is robust. As explained in [10], version detection scans involve interacting with the remote services, e.g. by connecting and sending additional probes specific to the service, and analysing the responses in order to determine the version of the service and other information.

4.1.3 Identification of well-known vulnerabilities

If the penetration tester was able to perform a service detection scan, well-known vulnerabilities and software bugs can easily be spotted at this point and these will be included in the final test report. As mentioned in chapter 1, a penetration tester operating in a production environment will generally *not* exploit

these vulnerabilities, in order to preserve the stability and integrity of the system. The identification of such vulnerabilities is sufficient to provide the customer with valuable information. Sometimes, however, a penetration tester might identify and execute a *safe exploit*, i.e. an exploit for which there is certainty that the target system will not experience damage.

A different approach to detect well-known vulnerabilities consists in analysing the data resulting from the remote information gathering technique described in section 4.1.8. The information collected with this procedure reveals details about the applications running on the remote machine and allows the penetration tester to identify vulnerable software.

In order to detect a vulnerable application, details such as version and patch level must be known. The ability to determine whether a specific software instance is vulnerable may derive from the penetration tester's experience or can be gained from vulnerability databases, i.e. databases that maintain records of vulnerable software.

The Nessus vulnerability scanner described in section 2.4.4 can also be used to identify vulnerabilities, provided that the scanner is configured appropriately to safely run in a production environment.

4.1.4 Techniques specific to Windows domains

When the system under test is based on *Microsoft's Windows domains*, a number of techniques specific to this type of environment can be used by a penetration tester. The main target within a Windows domain is the *Domain Controller* (DCO). The DCO is a server that manages all accounts within the domain, their permissions, the authentication mechanism, and all operations concerned with authentication and authorization.

One of the first pieces of information that a penetration tester aims to acquire is the list of usernames for the domain accounts, in order to apply password guessing or brute-force password attacks; the goal is to gain a limited domain access that may be used as a starting point for privilege escalation. Sometimes it is possible to retrieve the usernames list directly from the DCO. One way to do this is to leverage a *null session authentication* vulnerability, that allows a user to anonymously authenticate to the Server Message Block (SMB) service (see section 4.1.6) on a remote machine. Cain & Abel, described in section 2.3.4, provides a functionality that uses a null session authentication to enumerate all the accounts listed in the DCO. This is only possible for DCOs running an OS older than Windows Server 2008.

Another way to enumerate the accounts from the DCO is to use Simple Network Management Protocol (SNMP) requests. As mentioned in section 4.1.7, SNMP does not provide any lockout policy. It is therefore possible to brute-force the authentication for SNMP requests (i.e., to learn a *community* string for either read or read & write access) and, if successful, a penetration tester is able to retrieve a great amount of information about the system, including the list of usernames. Using SNMP it may even be possible to enable a device to enter packet capture mode to collect traffic or to forward traffic from within the organizations' own network to the penetration tester.

Windows systems provide the possibility to enable a Guest account to access a certain machine with predetermined restrictions. Sometimes, if the Guest account is enabled, its credentials correspond to the default values and the tester has immediate (limited) access to the system. When this is not possible, the Guest account can be used to determine the lockout policy* of the system. By intentionally supplying incorrect passwords, it is possible to determine the maximum number of attempts that can be performed before the account is locked. It is usually not possible to discover this information by using other accounts, because it is not desirable to lockout user accounts in a system in production environment. A locked Guest account, however, is not likely to cause problems. Information about the lockout policy can be used to adjust brute-forcing tools and make sure that user accounts are not locked as a result of an attack.

Windows systems connected to a Windows domain possess a *computer account*. While *user accounts* are mapped to human users, a computer account identifies a *machine* within the network. When a computer account is reset, an easily guessable password derived from the computer name is assigned to the account. It may happen, especially in large environments, that the password for a reset computer account is not changed, as a result of negligence and/or incorrect account management. A penetration tester may attempt to login to the domain with default passwords for computer accounts.

Once the list of usernames for domain accounts has been gained, a penetration tester may attempt to login to the DCO or other systems using guessable passwords (e.g. the same password as the username, the name of the department, the employee's name or date of birth). As shown by J. Bonneau in [27], users tend to choose weak passwords. This can be exploited by a tester or an attacker to gain access to the system.

*The lockout policy specifies the conditions for account lockout, i.e. the disabling of an account when an incorrect password is supplied a certain number of times within a specified time period.

4.1.5 Web applications

The web applications identified in the system under test are tested individually. The security testing of web applications is a very broad subject [22] and a large number of techniques can be used by a penetration tester. During a network-based penetration test, although the focus is not primarily on web applications, compromising such applications may result in exposure of other resources on the network. Therefore, the penetration tester will always attempt to leverage web applications' vulnerabilities by using a small number of common techniques.

The first operation performed by a tester after having identified a web application running on a certain [host,port] pair is to access the application through a web browser. *Forced browsing* [23] and *path traversal* [24] attacks can then be used to access possible resources such as configuration files, backup and test folders, etc. These resources may contain sensitive data, e.g. credentials to access an external database or source code that can be analysed to identify software bugs. The *admin panel* interface of a web application may also be targeted using techniques such as login brute-forcing and/or password guessing.

Web services [26] are another interesting element to be examined during a penetration test. Exposed web services may allow unauthorized users to execute methods on the web server and may result in vulnerabilities such as command execution, information exposure, file uploading, etc. Although web services may not be directly referenced within a web application, the location of such services may be identified with techniques such as eavesdropping and code review (if the source code of an application using the service is available).

If the web applications in the system are part of the scope of the penetration test and the tester is required to perform a complete security assessment of such applications, several additional techniques may be used. Stuttard and Pinto [22] provide a detailed description of the procedures for penetration testing of web applications.

4.1.6 Resource Sharing

Network resource sharing is a network service that may contain vulnerabilities or provide a penetration tester with means to compromise the network. A very common protocol providing shared access to network resources and inter-process communication is the SMB protocol [25]. This protocol is mostly used by computers running Microsoft Windows, but implementations for UNIX-like and Linux systems exist as well.

A common security issue related to resource sharing services is *weak access control*, i.e. improper or incorrect configuration of restriction to resources. This allows users to read and/or write to shared resources to which they should not have access, and may result in exposure of sensitive information such as source code, configuration files, backup folders, etc. In order to detect such vulnerabilities, a penetration tester may try to access SMB services *without* providing a password or, if available, with credentials of low-privileged accounts.

The SMB protocol may also be leveraged to perform other operations, such as remote information gathering (see section 4.1.8).

4.1.7 Default and guessable credentials

It is quite common, especially in large environments, to find services that are accessible with default or easily guessable account credentials. A penetration tester may try to access exposed services, specific network entities (e.g. a printer, a router, a database), web-based authentication interfaces for Content Management Systems, firewalls, routers, etc.

Depending on the lockout policy, it may also be possible to attempt (limited) brute-force attacks to login interfaces. Additionally, SNMP does not provide any limitation on the number of authentication attempts.

4.1.8 Remote information gathering

If account credentials are available (either as a result of other techniques or because they were provided at the beginning of the penetration test), a penetration tester may attempt to remotely gather additional information about the machines accessible by the available account. This step may result in the identification of several security issues and is an essential part of the *expansion* technique described in section 4.1.12.

A set of scripts has been implemented by Truesec in order to facilitate this remote information gathering. These scripts require valid credentials and leverage the SMB protocol to execute commands on the remote machine and to collect the results. Safe commands such as *ipconfig*, *systeminfo*, and *tasklist* are executed on the remote machine in order to gather as much information as possible about the system. Scripts are also executed to gain important pieces of information such as the list of account usernames and the passwords in their hashed form, using tools such as gsecdump (described in section 2.3.6). As explained in section 4.1.12, the results of these scripts' execution are essential for expanding the influence on the

entire system.

When the accessible host is based on a Windows operating system, a remote *patch level check* can be executed. Such analysis reveals whether the system lacks important security updates that may result in exploitable vulnerabilities. Microsoft has developed the Microsoft Baseline Security Analyzer (MBSA), a software tool that performs remote security assessment based on security updates and system settings.

4.1.9 Eavesdropping

Eavesdropping, i.e. the act of passively listening to network communications and analysing the content of network packets, is a non-intrusive technique that a penetration tester may attempt in the hope of gaining sensitive information. For example, if network entities communicate using *clear-text protocols*, it may be possible to discover transmitted passwords that services use to authenticate over the network.

A penetration tester would primarily target the traffic generated by applications that are likely to communicate to a database and therefore provide credentials.

4.1.10 Client-side attacks

Client-side attacks consist in exploiting *people* in order to gain access to the system under test. The goal is to make a user internal to the target organization execute malicious software and/or reveal sensitive information. This can be achieved with *emails* containing malicious content in the form of URLs, PDF documents, etc. Techniques used to discover sensitive information through people are referred to as *social engineering* techniques [15].

4.1.11 Extending the scan range

The scans executed in the initial phase of a penetration test usually target a limited range of port numbers. One of the reasons for this choice is that by limiting the scan to the most common ports, the majority of the services available in the system are probably identified in a relatively short amount of time. These services can be used as a starting point for the other techniques described in this chapter. Moreover, services running on common ports are more likely to be stable and robust, thus version detection may be attempted on these services.

When all safe techniques have been applied to the discovered ports, it is possible to execute a new scan with a wider port range in order to possibly discover new services and apply again the same safe techniques. This gradual way of proceeding assures that the services that are most likely to be robust are tested first, and less common ports may not even require an external scan because information on these services could be obtained by other means, such as remote information gathering. A common schedule for port scanning is the following:

1. scan the top 100 most common ports (option *-F* in Nmap);
2. scan the top 1000 most common ports (default behaviour in Nmap);
3. if the tester has access to a host in the network, it is possible to list all listening ports *from the inside*, and then scan all these ports for *all* hosts in the network. It is likely that other hosts in the network run the same service on the same port; and
4. scan the entire port range (1-65535). This scan requires a long time to be completed and is therefore the last choice of a penetration tester.

4.1.12 Expanding

As mentioned at the beginning of this chapter, the process of a *safe* penetration test involves gaining limited access to resources and/or information within the system under test, and exploiting this knowledge to further compromise the system. This *recursive* procedure allows the penetration tester to prioritize the safe techniques during every step of the testing process.

One of the first actions that the tester performs once gaining access to a machine in the network is to search for sensitive data within the compromised machine. For instance, configuration files containing unencrypted passwords to access network services may be accessible. Moreover, a penetration tester might be able to leverage weak access control to network resources to access shared files, exposed resources, and sensitive information within the domain that may be accessible by the account used by the penetration tester.

In section 4.1.8, remote information gathering was briefly introduced. A set of commands and scripts are executed on a remote compromised machine and the results are collected by the penetration tester. The remainder of this section illustrates what information is gathered and what can be assessed from a security perspective.

As a result of the remote execution of gsecdump (section 2.3.6), the tester gains knowledge about the password hashes of all local and domain accounts that have an active logon session in the compromised remote machine. This

information is used to identify system *dependencies*, i.e. the re-use of accounts with the same credentials among different hosts in the network. For example, if password hashes have been collected from different machines and the same [username, password hash] pair is stored in multiple hosts, then a dependency has been identified, as the same account has access to different machines. While this is not a security issue in itself, it is common to encounter weak access control policies that assign high privileges to users who do not need them. This allows a penetration tester to compromise *multiple* network resources with only one user account. The collected password hashes can also be *cracked* offline with tools such as Cain & Abel, as described in section 2.3.4. Moreover, the hashes can be used to execute processes within the context of the account for which the password hash is available; Truesec has developed a tool called RunhAsh [6] that allows the penetration tester to inject the password hash in a process and impersonate the owner of this password hash when accessing network resources.

The remote information gathering also provides the penetration tester with a list of all processes running on the remote machine and *what ports they are listening on*. This information can be used to refine a port scan and target specific port numbers. Ports that are found to be open on one host are good candidates for ports scanning on *all hosts* in the network, since the same application, if running on multiple hosts, is likely to be listening on the same port.

A check for antivirus presence and update state is also part of the remotely collected information. A lack of an (up-to-date) antivirus may be exploited by a penetration tester.

Another check that is performed based upon the information gathered from the remote script execution is whether privilege escalation is possible. The penetration tester checks whether a low-privileged account is allowed to start and stop specific services which in turn operate with system administrative privileges, and whether write operations to the location of the service's binary file are permitted for this account. If these conditions are met, then the low-privileged account can easily escalate its privileges to that of a system administrator. This can be achieved by substituting the service binary file with a program that, since it is executed with administrative privilege, is able to alter the accounts' properties.

Additionally, information about the remote machine's network interfaces are collected. This can reveal network segments that are not directly accessible by the penetration tester, but can be reached through the compromised machine.

If any of the accounts collected from a host is found to have *domain admin privileges*, i.e. unrestricted access to all network resources (including the DCO), then by knowing its password hash a penetration tester is able to take over the *entire system* using the *pass the hash* technique described in section 2.3.6.

4.2 Comparison with standard automated tools

The goal of this section is to analyse the differences between an aggressive penetration test (carried out by most of the standard automated tools and summarized in section 2.4.5) and the process followed by a penetration tester manually testing a system in a production environment. Some of the techniques used during a safe penetration test were described earlier in this chapter.

The main difference between the two approaches is that *vulnerabilities derived from software flaws are not exploited* in production environments. However, most of these vulnerabilities can still be identified and reported. The scanning phase is common to both approaches and, although slightly different, it leads to very similar results that may reveal vulnerable exposed services in the system under test. A penetration tester does not necessarily need to exploit these vulnerabilities, but simply point them out to the customer. Some exploits, however, are considered safe to exploit. For instance, there exists an exploit that targets HP's network backup system Omniback (called HP OpenView OmniBack II Generic Remote Exploit) that is considered to be safe as the target system's memory is not affected by the exploit. A penetration tester may decide to leverage a safe exploit to gain access to the vulnerable resource.

In a production environment, an experienced penetration tester always applies the safest techniques first. The execution of every single step of the process is carefully contemplated according to the risks that the system under test would be subjected to. For example, as described in section 4.1.11, the scanning of the environment is gradually extended throughout the testing process. Moreover, more accurate types of scans (such as version detection scans) are not executed against unknown services. Instead, these services are analysed *from the inside* once access has been gained.

Since vulnerabilities due to software bugs are usually not exploited, the tester needs to leverage other security issues in order to gain access and start the expanding process described in section 4.1.12. Examples of targeted security issues are configuration errors, default account credentials, weak access control, and exposed sensitive information. According to the results of Truesec's earlier penetration tests, large infrastructures are very likely to present some of these issues and a penetration tester may be able to compromise the entire system *without* the need for risky vulnerability exploitation.

As described in section 4.1.12, a safe penetration test follows a *recursive* procedure that allows the testers to gradually extend their influence on the system by always preferring safe techniques. Limited access to the system is needed in order to follow this recursive procedure. However, Truesec's experience is

that obtaining restricted access to network resources is usually relatively easy, especially in large environments.

The elements mentioned in this section will serve as a basis to define the logic of the automated tool implemented as part of this thesis project. The logic reflects the behaviour of a penetration tester operating in a production environment, who can use the techniques described in this chapter. The definition of this logic is described in Chapter 6.

Chapter 5

Design

This chapter describes the different phases that took place during the design of the automated penetration testing tool. At the beginning of this part of the project, a few important issues had to be considered (these are described in section 5.1) that determined whether to start the implementation from the ground up or to develop the tool on top of existing software. In section 5.2 an approach to the problems described is derived. This provided the foundations for the first sketch of the architecture of the new penetration testing tool that will be described in section 5.3. A clear distinction was highlighted between the design of the *architecture* and the definition of the *logic* (chapter 6). The architecture should provide support for the desired functions, such as dynamic reporting, extensibility, possibility of reproducing specific steps of the testing process, etc. On the other hand, the logic defines the behaviour of the tool and determines what actions will be taken, how results will be interpreted, and the order of the different steps. The concept of non-aggressive penetration testing will be reflected in the definition of the tool's logic.

5.1 Initial considerations

The main question that needed to be answered at the beginning of the design phase was whether to start the implementation from scratch or base the tool on existing software. There are advantages and disadvantages for both approaches, as will be described in the remainder of this section.

The advantages of using existing tools and frameworks as a base for the new application are obvious. The main benefit is that a large part of the code need not to be rewritten. Implementing functions already existing elsewhere

(e.g. a scanner) requires a large amount of resources and may be unproductive. Moreover, using existing software that is mature results in more powerful capabilities. Many bugs in this software have already been fixed, whereas a new application is likely to present new bugs and this requires additional time for debugging.

However, the choice of using existing software presents several drawbacks as well. An essential aspect to consider is the license of the software to be used. Depending on the type of license, the use of the product and its inclusion in other software may be limited. Relying on other products creates a dependency that in some cases is not acceptable. Possible changes in the license or support for a product will affect any other application that depends on that product.

There are some desired characteristics of the automated penetration testing tool that need to be considered throughout the whole testing process. For instance, the possibility of reproducing specific operations requires the application to perform detailed *tracking* of the testing process in all its phases. When using existing software, such operations need to be implemented externally.

Another issue that needs to be considered is platform dependency. When performing network-based penetration tests, it is common to access compromised machines in order to harvest additional information, as described in section 4.1.12. This often requires the tester to install and/or execute software on a remote machine. Depending on the platform and other characteristics of the machine, the tester selects the appropriate tool to perform this operation. In order for an automated tool to follow the same logic, either a platform-independent solution must be available, or a set of implementations for different systems should be developed. This can be problematic when relying solely on existing software.

Taking into account all of the aspects described earlier in this section, a possible solution was derived. This solution is described in the next section.

5.2 Approach

In order to address the issues mentioned in section 5.1 and to support the desired characteristics of the tool to be implemented, a set of design decisions were taken, as described in this section. These decisions define the approach to the problem and will serve as the basis for the design of the entire architecture (described in section 5.3).

5.2.1 Structure

The first aspect introduced in the design of the tool is a conceptual separation between the core application and the implementation of the various *actions*. Actions are defined as single activities, or functions, used by the application to perform specific tasks (such as running a certain type of scan, attempt to login to a specific service, etc.). The core, on the other hand, contains the logic and the internal resources needed to carry out the penetration test, as described in section 5.3.

The separation between the core and the implementation of the actions gives the application greater flexibility. In particular, an action can be implemented from scratch when needed or existing software can be used (e.g. a script or a separate tool) to perform the requested operation. Therefore, there is no need to implement from the ground up every single unit of the tool. At the same time, however, the core of the application remains flexible and maintains control of the complete testing process. The application does not depend on specific software, but rather depends on the operations performed, regardless of the specific implementation.

5.2.2 Platform independence

The flexibility introduced in the previous section gives another advantage. Since the core application does not need to execute any action, but instead relies on external implementations, the core itself can be made *platform-independent*. Actions that cannot be made cross-platform can then have multiple implementations, and the application simply selects the appropriate implementation as needed, depending on the platform that the tool is running on.

5.2.3 Extensibility

Since the initial knowledge about safe penetration testing at the beginning of this thesis project was rather limited, a gradual implementation of the automated tool seemed to be the most suitable development plan. As a result, the structure of the application must allow developers to improve the logic and extend its capabilities in a flexible manner, when a new testing activity or function needs to be included in the application. Therefore, the designed tool must be easily *extensible*.

In order to increase the extensibility of the automated tool, an additional separation of functionality was introduced. As mentioned in section 5.2.1 the actions are implemented external to the core of the application. A set of actions

together with appropriate logic constitutes a *vulnerability check*, i.e. a test for a specific security issue. These vulnerability checks are controlled by logic that handles the flow of the entire penetration test.

Summarizing, the *central logic* determines the flow of the penetration test and performs *vulnerability checks*. Each vulnerability check uses *actions* to determine whether security issues are present in the system under test. These actions are implemented externally. As a result of using this mechanism, all of the different parts of the tool can be extended independently. For instance, a single action can be added to the list of actions, or a vulnerability check can be added using the same available actions. The logic can also be modified to include new actions and vulnerability checks.

The central logic of the application is further organized in *steps*. Steps take care of the execution of the different operations according to the information about the system under test that is currently available. The concept of *steps* is explained in more detail in chapter 6.

5.2.4 Tracking and storage

The application that has been designed includes a database where data for each penetration test is stored. This database must contain everything needed to derive conclusions about a test that has been performed, as well to allow the user to reproduce specific steps of the testing process. The penetration test is constantly tracked and records are stored in the database. Together with detailed information about the system, the stored data allows the automated generation of a test report. Section 5.3 describes in more detail the data that needs to be stored into the database.

5.2.5 Customer perspective

In chapter 1, it was mentioned that it should be possible for a customer (owner of the system under test) to reproduce the steps that led to the discovery of a vulnerability in order to verify that the problem has actually been (re-)solved. All of the information that needs to be known in order to provide this functionality needs to be stored in the database. However, a customer has a different perspective than a penetration tester. Therefore, it was decided to implement two separate Graphic User Interfaces (GUI) that provide two different perspectives on the database. The penetration tester's GUI must allow the tester to dynamically influence the behaviour of the tool and to inspect every single item of information that has been collected. On the other hand, the customer's GUI only shows a

detailed list of all issues that were found in the system (in the form of a dynamic report), but it allows the user to re-execute specific vulnerability checks and to compare the outcome to the earlier outcome stored in the database.

5.2.6 System state change and reproducibility of checks

This section addresses the issue of system state changes and the ability to reproduce vulnerability checks based on a static system state.

Vulnerability checks and actions interact directly with the system under test and the state of the system may change due to a particular series of interactions. However, the reproducibility of specific vulnerability checks mentioned in section 5.3.2 does *not* assume that the state of the system at the time of re-execution is the same as when the check was initially performed. The goal of reproducing a check is to determine whether the vulnerability that was detected is still present in the system, therefore the state of the system must change in order for the tool to verify that an issue has been solved.

At the same time, however, state changes may prevent the tool from reproducing a vulnerability check. In the simplest example, a specific host may be assigned a different IP address and all security issues associated with that host are no longer available for analysis. These types of changes in the system result in an incorrect behaviour of the automated tool, resulting in limitations and possible inconsistent functionalities.

The first version of the tool will not address these issues. A later improvement, however, is suggested as a future work in chapter 9.

5.3 Architecture

This section describes the architecture of the tool that was implemented during this thesis project. Figure 5.1 shows the architecture of the automated penetration testing tool.

As introduced in section 5.2.1, actions are defined as single activities, or functions, used by the application to perform specific tasks. Actions are composed of all the operations that involve *interaction* with the system under test, such as running a certain type of scan, attempting to login to a specific service, etc. As shown in figure 5.1, the definition of the actions is internal to the application. The definition specifies the operation performed, including the parameters needed to execute the action and the resources that are affected as a result of the

execution. However, the implementation of the action may be external to the tool. As explained in section 5.2.1, this allows the penetration testing tool to use external tools and scripts to perform certain actions and does not introduce any platform-specific requirements.

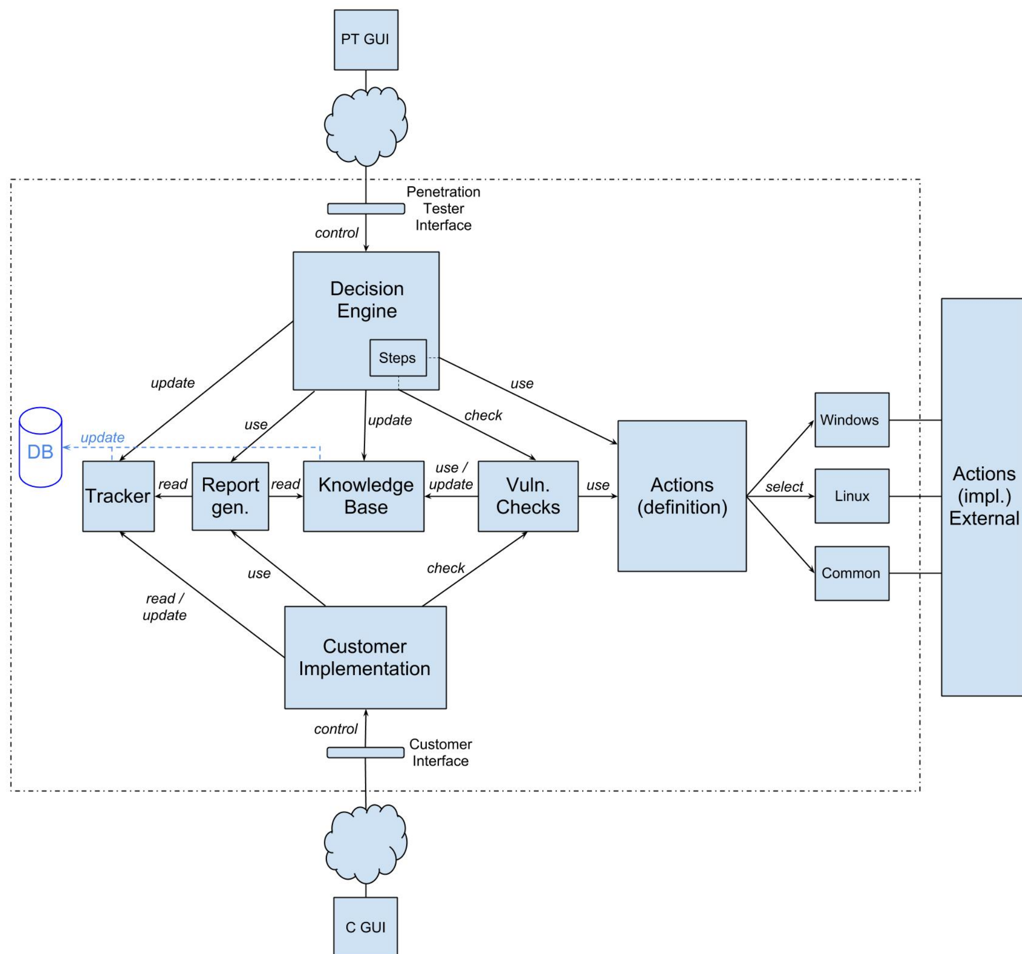


Figure 5.1: Architecture of the proposed automated penetration testing tool.

5.3.1 Actions

Since the core of the application is platform-independent, system-specific implementations of the actions need to be duplicated externally for all of the different target platforms. For instance, an action that utilizes Linux-specific commands must have a Windows counterpart that executes equivalent commands

in a Windows environment. When the automated penetration testing tool is launched, the host operating system is detected and the appropriate implementation for the actions is loaded. Additionally, the OS of the target system is detected and suitable actions are selected for execution.

In order to achieve the final goal of this thesis project, i.e. a *non-aggressive* tool for automated penetration testing, all actions are assigned a pre-determined *risk* that will be used by the controlling logic to decide whether a specific action is permitted in a certain user configuration.

5.3.2 Vulnerability Checks

As mentioned in section 5.2.3, *vulnerability checks* are used to determine whether a certain security issue is present in the system under test. Vulnerability checks contain a logic that uses *actions* to determine the outcome of the single assessments. Similarly to what was introduced for the actions, a *risk* is assigned to the vulnerability checks, the risk is calculated using the risk levels of the actions that are used in this vulnerability check. To simplify out calculations we assume that the risk associated with a vulnerability check is the **maximum** of the risk of any of the actions that will be applied to the system under test for this vulnerability check.

The most important characteristic of a vulnerability check is that it is *reproducible*, i.e. it is possible to execute the exact same operation at a future point in time. In order for this to be possible, the execution of each vulnerability check must be tracked and every useful item of information necessary to reproduce the vulnerability check must be stored for future use.

5.3.3 Knowledge Base

The knowledge base contains all of the data that needs to be stored in order to conduct a penetration test and generate the final report. It contains general information about the system under test, details of every host and user account that were detected during the test, etc. The content of the knowledge base is constantly updated with the results of actions performed and the results of the vulnerability checks, and the database is kept synchronized with it. The knowledge base utilizes the underlying database to store and retrieve information. The most important pieces of information stored in the knowledge base are illustrated in the UML diagram in figure 5.2.

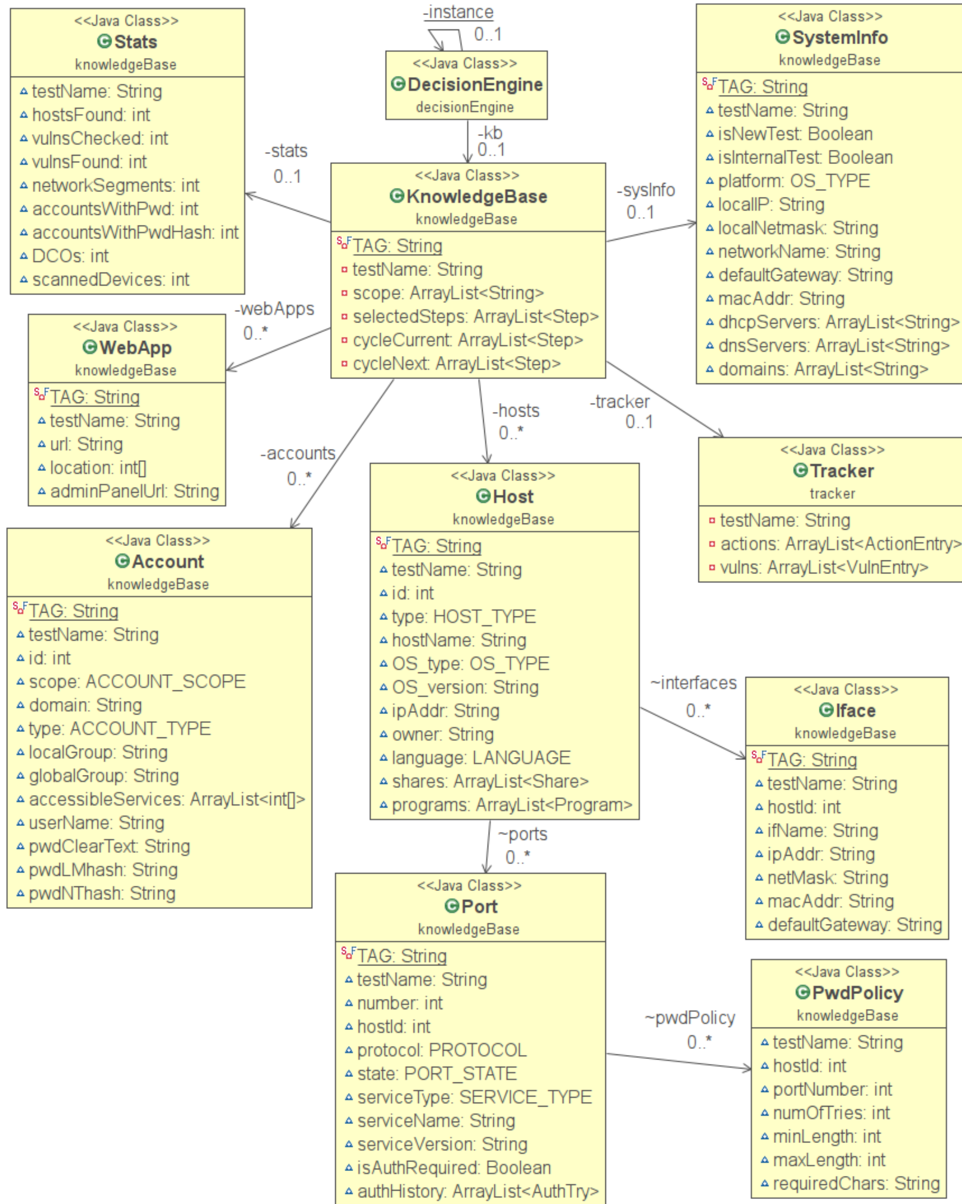


Figure 5.2: UML class diagram of the main classes in the knowledge base.

5.3.4 Tracker

The tracker behaves similarly to a *logger*, but with the important addition that it keeps track of reproducible vulnerability checks. For each vulnerability check, the tracker records the following pieces of information:

1. timestamp, used to produce a chronological history of the steps executed during the penetration test;
2. vulnerability tag, i.e. a unique identifier of the vulnerability check;
3. value of the parameters that were used during the vulnerability check (these can later be used to simulate the exact same scenario at the time of the execution);
4. result of the vulnerability check; and
5. additional notes.

The tracker, in the same way as the knowledge base, utilizes the underlying database.

5.3.5 Decision Engine

The decision engine contains the main logic of the tool and controls the flow of the penetration testing tool. The flow determines what actions and vulnerability checks are executed, what resources are involved, the ordering of the operations, the analysis of the risks, etc. All operations are organized in *steps*. The concept of non-aggressive penetration testing is realized by the decision engine. More details about the definition of this logic and the procedures followed during the penetration tests are explained in chapter 6.

The knowledge base and the tracker are updated by the decision engine following every operation executed during a penetration test, and these changes are stored in the underlying database. Therefore, the decision engine does not need to contain any test-specific data. Instead, a test will be loaded from the database, thus allowing the decision engine to control multiple penetration tests and execute these penetrations and their analysis at different points in time.

The decision engine is directly controlled by the penetration tester through the penetration tester GUI (see section 5.3.8). The penetration tester is able to configure the decision engine and tune the testing procedure to reflect the requirements of a specific environment, for example by specifying the maximum risk level that a vulnerability check may have in order to limit which vulnerability checks will be executed.

5.3.6 Report Generator

The report generator unit reads information from the knowledge base and the tracker, and generates a report that can be used to reproduce selected steps of the testing process. This component is essential from the customers' perspective, since it provides the results that are valuable to the customer. As shown in figure 5.1, the decision engine makes use of the report generator as well, allowing the penetration tester to refine the final report before it is delivered to the customer.

5.3.7 Customer Implementation

The customer implementation unit provides the customer with a report of the test execution (as generated by the report generator). When the customer selects a vulnerability to be reproduced from the report, the customer implementation unit retrieves the necessary information from the tracker and executes the appropriate vulnerability check.

5.3.8 Penetration Tester GUI

The penetration tester GUI (PTGUI) allows the penetration tester to interact with the decision engine. The implementation of the PTGUI resides external to the application (i.e. it is an separate application) and can access the decision engine remotely. This allows the penetration tester to run the main application on a server with sufficient resources (in terms of memory capacity and computing power), while the penetration tester is still able to control the penetration test remotely (for example, from their office).

In order to maintain a certain level of flexibility, the PTGUI provides the user with the possibility to manually add information to the knowledge base. For instance, the penetration tester may add a user account that was not detected automatically by the application.

5.3.9 Customer GUI

The customer GUI provides a graphical interface for use by the customer. It displays the final report, including the vulnerabilities that were identified in the tested system and allows the user to select vulnerability checks to be reproduced. Similarly to the PTGUI, the customer GUI can access the penetration testing application remotely.

5.3.10 Database

A database is needed in order to permanently store data about the different penetration tests. A mapping has been defined between the tables that constitute the database structure and the different units described in this section. Every change in the tracker or in the knowledge base triggers a corresponding update in the database. A penetration tester can load a test into memory from the database without losing other test data.

5.4 Application scenario

This section describes how the automated tool introduced in this thesis project can be utilized in practice. There are four components linked together to deliver the service to both the penetration tester and the owner of the system under test: the testing engine, the database, the penetration tester GUI, and the customer GUI. Figure 5.3 illustrates these components and how they should be connected.

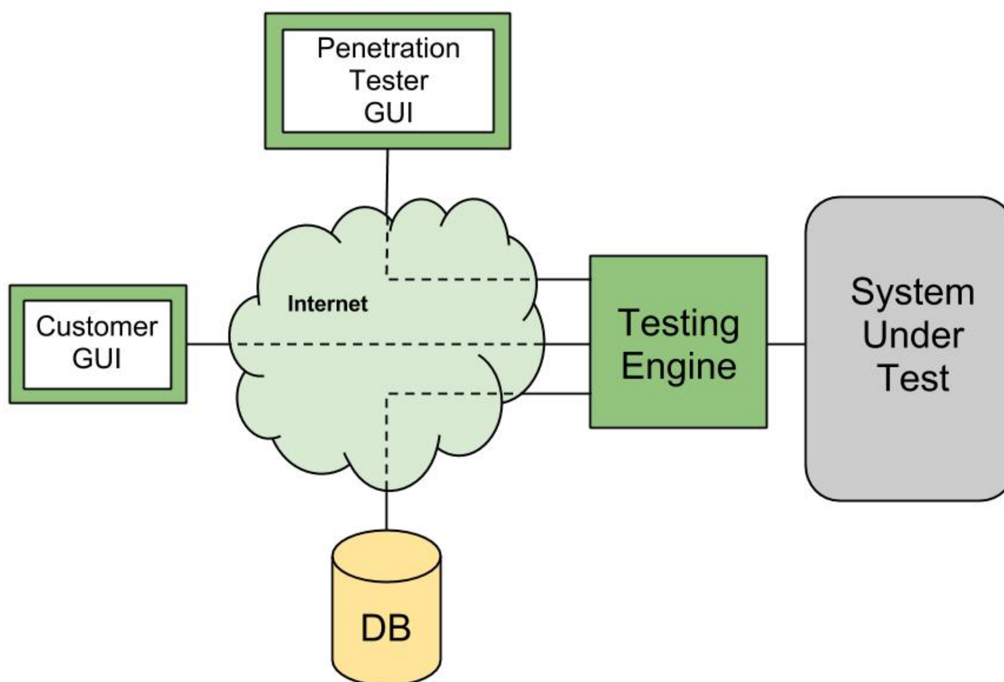


Figure 5.3: Scenario showing the different components of the testing application.

The testing engine must be connected directly to the system under test. Future

extensions of the application might support remote connections to the system under test. However, the current version of the engine only supports direct connection, since a number of operations executed during the test are based on the fact that the engine is part of the network to be tested.

The database, described in section 5.3.10, does not need to reside in the same location as the testing engine. The database can be located in any location that is reachable by the testing engine, provided that the engine is configured appropriately to connect to the database.

The penetration testing GUI (described in section 5.3.8) and the customer GUI (described in section 5.3.9) can connect to the testing engine via the Internet as well. The GUIs do not need to connect to the database, since the testing engine is the only component that communicates with the database.

The possibility to locate the different components at different locations gives the testing system great flexibility. Only the testing engine needs to be connected directly to the system under test. For instance, the database could be located at the penetration tester's office, while the tester could remotely perform the test from yet another location. The different components can also be placed on the same machine, when this flexibility is not needed.

Chapter 6

Logic

As mentioned in chapter 5, a clear distinction was made between the design of the architecture of the tool and the definition of the logic that defines the flow of the penetration testing process. This chapter illustrates how the automated tool is able to select the steps to be executed, how the overview of a penetration test is maintained, and what operations are actually performed on the system under test.

6.1 The Penetration Test Life Cycle

The main issue that had to be considered during the design phase was the fact that the information contained in the knowledge base, i.e. what the tool knows about the system under test, dynamically changes throughout the whole testing process; it is therefore impossible to assume at any moment that the available data is sufficient to evaluate the system entirely. In other words, when a certain operation is executed, there is no guarantee that the same operation will not be needed again on data that may only become available at a later point in time. As an example, consider the state of the knowledge base when a certain set of hosts have been discovered. One of the possible operations that the tool may execute is a check for accounts with default passwords. This operation would be executed on the currently available data, in this case all the hosts currently stored in the knowledge base. However, in a later phase of the penetration test, new hosts may have been discovered (as a result of other operations) and there would be a need to execute again this check for default credentials on the new hosts.

As a result of the analysis of this issue, the following requirements for the design of the logic were formulated:

1. all operations that are executed should be tracked;
2. it should be possible to check the history of the operations in order to avoid duplicates;
3. the decision on what resources are involved in a certain operation should be logically separated from the actual execution of the operation; and
4. the overall life cycle of a penetration test should be based on a looping process.

These requirements, if fulfilled, allow for a simpler implementation of the single operations (actions), but introduce an additional level of responsibility for keeping the overview of the system. The looping approach aims to solve the issues related to the dynamic nature of the knowledge base.

The first requirement was already met, since the architecture described in chapter 5 includes a *tracker* unit that is capable of maintaining a record of all operations performed. Part of the definition of the tracker also satisfies the second requirement, as functions to analyse the history of a penetration test are included in the implementation.

The concept of *steps* (briefly mentioned in section 5.2.3) was introduced in order to fulfil the third requirement. *Actions* and *vulnerability checks*, described in chapter 5, take care of the execution of single operations and evaluate whether the system is vulnerable to certain security issues. These units, however, do not maintain the overview of the system and must not be involved with the selection of the resources from the knowledge base. They accept "simple" parameters (e.g. an IP address, an account's credentials, parameters for a network scan) and simply perform their tasks on the specified resources.

The task of selecting the appropriate resources from the knowledge base is assigned to the individual steps. Each step contains the following information:

1. a unique tag identifier;
2. a description;
3. a risk, depending on the risks of all the actions and the vulnerability checks involved;
4. a sequence number, used to order the steps when carrying out the test; and
5. the implementation of the execution.

When a step is executed, no parameters are passed. It is part of the step's execution to exploit the current state of the knowledge base, select the relevant resources, and execute actions or vulnerability checks specifying the selected resources. However, the history of the penetration test is checked before executing further operations, in order to avoid duplicate executions. A list of steps that have

been implemented or may be included in future versions of the tool is presented in the following section (section 6.2).

Since the overall testing process should be organized in loops (see the fourth requirement above), the steps are assigned to *cycles*. Figure 6.1 illustrates an example of the process followed by the automated penetration testing tool that includes all the concepts introduced in this chapter. A description of the available steps is given in section 6.2.

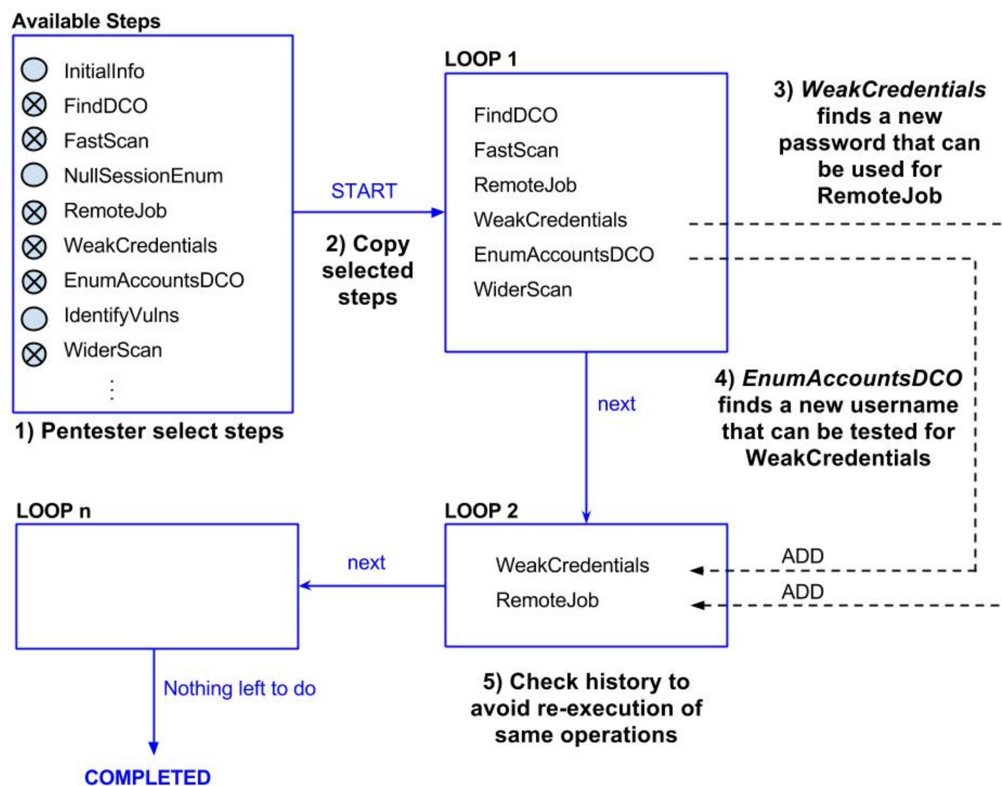


Figure 6.1: Example of an automated penetration test.

The first operation that a penetration tester (referred to as "pentester" in Figure 6.1) must conduct when using the tool introduced in this thesis project is to configure the tool for the specific test to be executed. This includes specifying all the steps that the software is allowed to execute among a list of available steps.

When the test is started, the selected steps are copied to the first loop of the test iteration and the tool starts to execute them one by one. Because of the issue described at the beginning of this chapter, steps that have already been executed may need to be run again. Therefore, the steps have the capability of

adding other steps to future cycles when needed. In the example of figure 6.1, the step *WeakCredentials* (responsible for attempting logins to services using default or easily guessable usernames and/or passwords) discovers a new password. The knowledge base is then updated with the new data. The new information, however, may be useful to other steps, in this case the *RemoteJob* step, which collects data from remote machines but needs valid account credentials in the knowledge base. The step is therefore added to the following cycle. In the same way, the *EnumAccountsDCO* step discovers a new username (by enumerating the domain users from the Domain Controller) and triggers the execution of the *WeakCredentials* step for the next cycle.

In every cycle and for every step, the history of the performed operations is checked in order to avoid re-executions. The iteration continues until no steps are added to the following cycle.

Although the proposed solution obviously presents some efficiency issues due to constant history checks, several components of the application are considerably simplified. Future versions of the tool, however, should consider scalability issues that could arise from this approach to planning the next cycle. The topic is briefly covered when we describe potential future work in section 9.2.

6.2 Individual Steps

The automated penetration testing tool implemented as part of this thesis project allows the user to individually select and execute single steps of the process, independent of the logic described in the previous section. The penetration tester can run specific steps one at a time. This feature closely resembles the behaviour of a manual penetration test, i.e. allowing the tester to carry out operations quite meticulously. Depending on the risk of the individual steps and the level of automation desired, the penetration tester may decide to either execute the steps individually or select a set of (or all) the available steps and run the tool using the logic described in the previous section.

Table 6.1 presents some of the steps that were either implemented during the thesis project, were defined but not implemented, or simply are suggested as a future extension. The list covers only a small part of a generic penetration test and was used as a starting point for the implementation of a new tool, which can later be extended.

Table 6.1: List of steps that the penetration tester can select and execute.

Tag	Risk [0-10]	Seq. Num.	Description
InitialInfo	0	1	Collects information that is readily available when connecting directly to the system under test, such as system platform, local IP address, network mask, network name, default gateway, MAC address, DHCP servers, DNS servers, domain names.
FindDCO	0	2	Finds the Domain Controller for the available Windows domains. Assuming that a DHCP server automatically configured the network interface used by the testing engine, the assigned DNS suffix is resolved. The tool then uses the obtained IP address to attempt a connection to the inter-process communication share (IPC$). If successful, the machine is assumed to be a DCO.
FastScan	4	3	Runs a *nmap* scan with options *version* (-sV) and *fast* (-F). Scans the 100 most common ports and attempt service detection.
Weak Credentials Accounts	1	4	Attempts connections to hosts and services in the knowledge base using default or weak credentials. The list of credentials is specified in external XML files.
Weak Credentials DataBase	1	5	Similar to *WeakCredentialAccounts*, but specific for database services.
EnumAccounts DCO	2	6	Attempts enumeration of user accounts from the Domain Controller.
Remote Collect	3	7	Uses the available accounts to connect to remote hosts, execute a set of scripts, download the results, parse them and store the data in the knowledge base. The collected data includes password hashes, shared resources with associated permissions, list of installed programs and their versions, interface configurations, etc. This step requires valid credentials, preferably with local administrator privileges.

Account Dependencies	0	8	Uses the data collected from the RemoteCollect step to determine whether account dependencies are present within the system. One possible vulnerability is confirmed if the same credentials are used for local administrators on different machines. Another issue that is checked is whether domain accounts with unnecessarily high privileges are used to access non-critical resources.
SharesAccess Control	0	9	Uses the data collected from the RemoteCollect step to determine whether the access control policies regulating shared network resources are configured properly.
Patch Management	0	10	Uses the data collected from the RemoteCollect step to determine whether an appropriate patch management policy is in use and-or whether vulnerable versions of installed software are present in the hosts. This step requires a *software vulnerability database*.
Indentify Vulns	0	11	Similar to PatchManagement, but makes use of scan results instead of remotely collected data.
Computer Accounts	1	12	Attempts to connect to network hosts by using computer accounts with default credentials (see section 4.1.4).
Guest Accounts	1	13	Attempts to connect to network hosts by using default guest accounts (see section 4.1.4).
LocalAdmin Escalation	4	14	When connected to a remote machine, this step checks whether privileges escalation is possible. As described in section 4.1.12, this happens if a low-privileged user is allowed to substitute service binaries that are executed with admin privileges.
Eavesdropping	0	15	Listens passively to network traffic in an attempt to intercept useful information such as non-encrypted credentials, requests to web services that are not directly linked to web applications, services communication over unusual port numbers, etc.

Chapter 7

Implementation

The programming language that was chosen to implement the tool introduced in this thesis project is Java. Several reasons led to this choice. The most obvious reason, derived directly from the definition of the architecture described in chapter 5, is platform-independence. Java runs on any platform where a Java Runtime Environment (JRE) is installed, making it possible to straightforwardly write a cross-platform application. Another requirement that was introduced in chapter 5 and that fits well with this choice is extensibility. The object-oriented nature of the Java programming language facilitates the implementation of software that is supposed to be later extended. As for the need to include external implementations of the actions in the application, Java provides the ability to execute arbitrary commands on the underlying system that is hosting the JRE. These external scripts and tools can therefore be invoked directly from the Java framework.

The biggest effort during this thesis project was the implementation of the Java framework as it constitutes the most important part of the automated tool. This framework includes all the internal components that were introduced previously and were illustrated in figure 5.1, i.e. the decision engine, the database, the tracker, the report generator, the knowledge base, the customer implementation, and the vulnerability checks. Some of the actions are also part of the internal framework, while others are implemented externally.

Once the framework was implemented, it became relatively easy to extend the application with additional features, thanks to the focus on extensibility maintained during the implementation. In order to conclude the implementation of a proof of concept that could be tested in a simulated environment, a few important features were added to the tool, e.g. some of the steps described in table 6.1 (for more details on the state of the application at the time of testing, see section 8.1). However, regardless of the operations that the tool can perform on the

system under test, the internal framework itself presents the following important characteristics:

1. It is relatively easy to extend the application, as described in more details in the rest of this section.
2. All the data collected from the system under test and the results derived from the interpretation of the data are re-usable. Since everything is maintained in the database, future extensions can manipulate and use the data that belongs to older tests. This also allows the penetration tester to implement functionalities on-the-fly when needed.
3. The penetration tester can manually add or modify data in the knowledge base at any time. This allows the penetration tester to instruct the tool based upon information that may be available to the tester in advance and/or to skip potentially risky steps and fill the gaps manually.

The last point is considered particularly important because it extends the possible applications of the tool. In particular, since there is the possibility to manually insert data into the knowledge base and only execute specific steps (automatically or individually) based on that data, the tool is suitable for the execution of *health checks*, in addition to penetration tests. A health check is generally considered to be different from a penetration test as complete access to the system is given to the tester, whose job is to assess the security of the system from the inside. For instance, during a health check, issues such as patch management, system configurations, and access control policies are taken into account, but the aspects related to *gaining access* to the system are usually unnecessary. The ability of the implemented tool to flexibly accept user-provided data and run with different automation levels makes it possible to use this tool for penetration tests, health checks, and any configuration in between.

The property of extensibility has been mentioned more than once, but the effort needed to actually introduce new elements to the application has not been illustrated yet. The following list describes in more details the changes and additions that a developer must introduce in the application in order to extend it with new functionalities, for example when adding a new step (such as one of the ones presented in table 6.1).

1. Since the database is constantly synchronized with the data stored in the Java objects within the application, the first change to make is to add the relevant new tables and/or columns in the database. This includes modifying the part of the code where all the tables for a new test are created.
2. Secondly, the new objects must be added to the knowledge base shown in figure 5.2, as well as all the attributes and methods needed to handle them (class constructor, getters and setters).

3. In addition to the simple methods implemented so far, the developer needs to implement the methods to *load* a specific test from the database, i.e. read the data from the database and populate the new Java objects.
4. Lastly, the new elements must be added to the GUI and all the methods defined in the interface between the GUI and the engine need to be implemented. These methods take care of the two-way communication between the two entities, i.e. updating the GUI when the data in the knowledge base is changed, and updating the knowledge base when the GUI (driven by the tester) requires manual changes.

Although it is not a straightforward procedure, these four tasks are quite easy to perform, the new code closely resembles already existing parts, and the time required to introduce the changes is very short. As part of the suggested work in section 9.2 a simplification of this procedure is suggested.

Chapter 8

Results

This chapter presents the results of this thesis project. In particular, the state of the implemented application at the time of testing is described, as well as the details of the testing environment and the results of the tests. These tests were performed using the tool introduced in this thesis, as well as other existing automated tools. The purpose of the tests was not to thoroughly compare the efficiency of the tools; instead, the different solutions were used to assess the security of the test system in order to show that different approaches can be valuable in different ways, and to estimate whether the non-aggressive approach could potentially constitute the basis for a complete automated tool to be used in production environments.

8.1 State of the application

As mentioned in chapter 7, the implementation focused on producing a proof of concept tool with sufficient functionality to perform a penetration test in a simulated testing environment. Priority was given to those steps that most reflect the non-aggressive approach investigated during this thesis project. At the time of testing, the tool included the following functionalities:

1. a GUI for the penetration tester to remotely connect to and control the testing engine, as well as to receive and display real-time data resulting from the execution of the tests;
2. the testing engine, configurable by means of XML configuration files;
3. a mysql database, possibly located on a different machine to which the testing engine can be configured to connect to;

4. creation and execution of multiple *tests*, i.e. jobs associated with specific organizations, networks, sub-networks, etc.;
5. possibility for the tester to configure the scope of the test in terms of IP addresses, as well as the steps to be executed during the test;
6. support for manual introduction of testing data by the penetration tester, possibly affecting all the items in the knowledge base, at any time before, during, or after the test;
7. ability to run the selected steps automatically by exploring the knowledge base and checking earlier execution history to avoid repetitions;
8. possibility to execute steps *individually*, and possibly alternate these executions with automated sessions;
9. functionality to generate a simple report based on the vulnerability checks performed during the test; and
10. permanent storage of data results, allowing the tester to pause their work and/or share the (interactive) results with colleagues.

Referring to table 6.1, that described the different steps of the penetration testing process, the following steps were implemented at the time of testing:

1. InitialInfo;
2. FindDCO;
3. FastScan;
4. WeakCredentialsAccounts;
5. WeakCredentialsDataBase;
6. RemoteCollect;
7. AccountDependencies; and
8. SharesAccessControl (partially implemented).

The state of the application presented in this section allows the tool to be used in a limited number of possible scenarios. However, the small number of functions which were implemented are sufficient to discover security vulnerabilities that are common in large corporations, as shown by the results of the penetration tests performed by Truesec. The set of possible scenarios can easily be broaden by extending the implemented tool (see chapter 7).

8.2 The testing environment

One of the initial goals of this thesis was to compare the automated tool introduced in this thesis with automated tools that follow a different (more

aggressive) approach. The implemented proof of concept provides a means to compare these two different approaches. However, part of the behaviour of the tool can only be evaluated by making assumptions about the behaviour of future extensions. Therefore, the two approaches are compared by examining the results of experimental tests, as well as theoretically analysing these two different procedures.

A virtual network environment was set up in order to evaluate the behaviour of the implemented tool. This virtual environment represents a relatively simplistic scenario (as described in the next section) of a small computer network. The implemented tool has been used to perform a (limited) penetration test against this virtual network using both an automated and a more manual approach, as will be discussed in section 8.2.2.

In order to compare the approach of the new tool with existing automated software, two other tools have been used to assess the security of this virtual environment: Metasploit Autopwn and Nessus vulnerability scanner. The two other tools presented as related work in section 2.4 (Core Security's Impact and Immunity's Canvas) were not available for testing due to their high price.

8.2.1 Configuration

The virtual network environment that was set up for testing consists of four hosts. The details of their configuration are presented in table 8.1.

Table 8.1: Configuration of the virtual machines used in the test.

IP	Hostname	Role	OS version	Services
192.168.78.10	DOMCOM	Domain Controller	Microsoft Windows Server 2008 R2 Standard Service Pack 1	DHCP DNS Kerberos RPC LDAP
192.168.78.1	SERVERONE	Web Server	Microsoft Windows Server 2008 R2 Standard Service Pack 1	HTTP RPC
192.168.78.130	WINDOWS7	Workstation	Microsoft Windows 7 Home	
192.168.78.3	BT	WorkStation	BackTrack 5 R3	SSH MYSQL

The DOMCOM machine is the DCO of the network. It also provides DNS and DHCP services for hosts connected to the network. A built-in account with administrative domain privileges was created when installing the DCO. The server SERVERONE hosts a web service on port 80. It is also configured to share a particular folder within the network. Two accounts with local administrative privileges were added to this machine. SERVERONE is part of the Windows Domain administered by DOMCOM. The WINDOWS7 machine is simply a workstation that is not part of the Windows Domain. Two local accounts are configured in this machine. The BackTrack machine hosts a mysql server and provides SSH access.

A few vulnerabilities were deliberately introduced in the virtual system in order to estimate the efficiency of the different tools:

1. weak credentials, in particular the mysql service on the BackTrack machine is accessible with default credentials [*mysql:mysql*] and SERVERONE has a local admin account with password *Password123*;
2. local account dependency, i.e. two different machines (WINDOWS7 and SERVERONE) are accessible with the same *local admin* account (both username and password);
3. insecure usage of high privileged account: the domain administrator has logged on to SERVERONE, which is not a DCO;
4. weak access control on shared resources; and
5. DNS clients on the three Windows machines vulnerable to a Remote Code Execution exploit (vulnerability reported by Microsoft in the bulletin at [29]).

8.2.2 Test Execution and Results

Three different kind of tests have been performed on the virtual network. The first test consisted in an exploitation attempt launched using Metasploit Autopwn. As explained in section 2.4.1, the process followed by Autopwn consists of scanning the target(s), detecting possible vulnerable services, and launching all matching exploits that can be found in Metasploit's exploit database. This attack was launched from the BackTrack machine and therefore targets the three remaining machines in the network. The sequence of commands executed

from the Metasploit console used to launch the attack were:

```
msf > db_connect
msf > db_nmap 192.168.78.10
msf > db_nmap 192.168.78.1
msf > db_nmap 192.168.78.130
msf > db_autopwn -p -t -e
```

The *db_connect* command connects the Metasploit framework with the database where the hosts' data will be stored. *db_nmap* performs a Nmap scan and stores the results in the database. Finally, *db_autopwn* triggers the remaining part of the process (exploit matching and execution). The version of the Metasploit framework used for this test was v4.7.0-dev [core:4.7 api:1.0]. The exploits contained in the database of this version of the framework were not able to successfully exploit any of the services in the three target machines, although a large number of matching exploits were found. The output of the Autopwn execution against the DCO are reported in appendix A. Similar results were produced for the other machines in the network.

The second approach to assessing the security of the network consisted in launching a Nessus vulnerability scan (see section 2.4.4). Nessus was configured to operate in an internal network. In addition to a SYN scan, the policy was modified to add a more intrusive TCP scan, as shown in figure 8.1.

Nessus was able to detect several vulnerabilities in the different hosts. Three of these vulnerabilities were signalled as *high severity* vulnerabilities. These critical issues correspond to the vulnerability MS11-030 (see Microsoft's bulletin at [29]) that was found in the three Windows machines, hence the three separate warnings. A medium severity vulnerability was found in both SERVERONE and WINDOWS7. This issue concerns the lack of message signing on the remote SMB servers, thus potentially allowing a man-in-the-middle attack against the SMB servers. A number or low severity vulnerabilities were found as well. The executive summary of Nessus' vulnerability scan is reported in appendix B.

Finally, the tool implemented in this thesis project was tested on the virtual network. For simplicity reasons, the testing engine, the GUI, and the database were all located on the same machine. One network interface controller (NIC) of this machine (manually specified in the configuration file of the engine) was connected to the virtual network and received an IP address as well as other network information from the DHCP service located at the DCO. We assumed that the DHCP server would give this machine an address on the network, this simplified the process of starting the penetration test by providing the tool with

Figure 8.1: Nessus scan policy adopted during the test.

basic network information. Future versions of the tool, however, should not rely on this assumption.

Since the tool is able to automatically detect the DCO(s) of the network it is attached to, only the IP addresses of the three other machines in the network had to be specified in the scope. Alternatively, the whole subnet could have been added to the scope, but this obviously would have required a much longer time to complete the scan phase. A screenshot of the configuration of the test is shown in figure 8.2. The steps that were not selected for execution were excluded as they were not fully implemented at the time of testing. The following list presents the

details of the execution of the test:

1. The step *InitialInfo* gathered network information: network name, local IP address, subnet mask, default gateway, DHCP servers, DNS servers, and domain name.
2. Part of this newly discovered information was used to find the domain controller (step *FindDCO*), which was detected and added to the knowledge base.
3. Step *FastScan* performed a Nmap scan on the four hosts (the three that were specified in the scope, plus the DCO that was just discovered).
4. The results of the scans were parsed and all the open ports and services were added to the respective hosts in the knowledge base, as shown in figure 8.3.
5. The mysql server running on the BackTrack machine was tested for weak credentials (step *WeakCredentialsDatabase*) and was accessible with default credentials [*mysql:mysql*].
6. Although the tool has discovered valid account credentials to access the mysql service on the Linux machine, no accounts were available for the other (Windows) hosts, thus the other steps could not be performed (a future improvement of the tool could include more steps that do not require valid credentials). At this point, the ability to manually insert data into the knowledge base became useful. Assuming that the tester knows the name of one of the employees working in the organization (for the purpose of this test, a fictitious account with username *John* was added to SERVERONE (without specifying any password)). To simplify the procedure, it was also assumed that valid credentials were available to access WINDOWS7, and the new account was added to the knowledge base, as shown in figure 8.4. This is a reasonable assumption considering real-world scenarios in which the owner of the system provides the penetration tester with a valid account in order to make the test less dangerous for the stability of the system.
7. The test was then re-run. Since the history is checked constantly, all operations that were already executed were skipped.
8. This time, the step *WeakCredentialsAccount* used the available username (*John*) to gain access to SERVERONE using the weak password *Password123*.
9. The step *RemoteCollect* was then able to use the two available accounts to remotely gather information from SERVERONE and WINDOWS7. This step stored in the knowledge base information about the accounts with a valid logon session that were found on the hosts (gsecdump execution, see section 2.3.6) as well as local resources shared within the network, and programs installed on the local system (an example is shown in figure 8.5).

10. The data collected from the remote hosts allowed the following step (*AccountDependencies*) to detect two issues related to account usage. The first was that John's account was found to be logged on *as a local admin* to both SERVERONE and WINDOWS7, while the second one concerned the *Domain Admin* account that was found to be logged on to SERVERONE, which is not a DCO. An example of vulnerabilities reported by the tool is illustrated in figure 8.6.

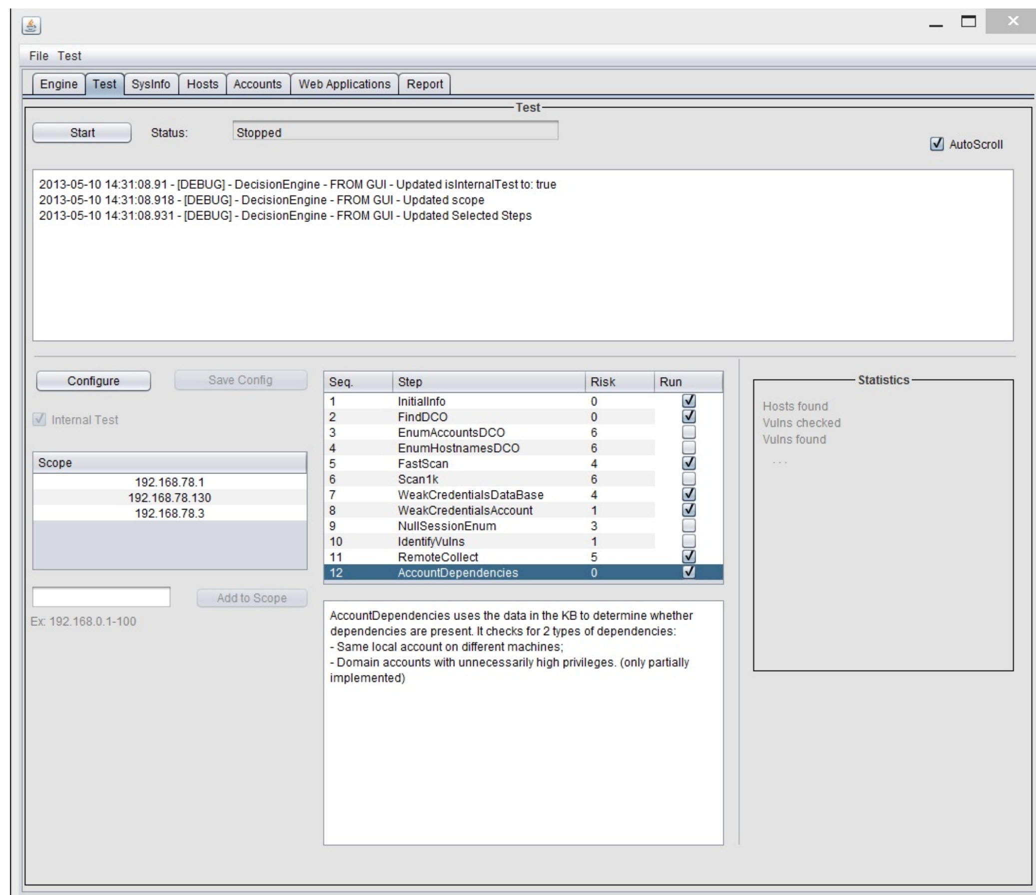


Figure 8.2: Configuration of the penetration testing tool.

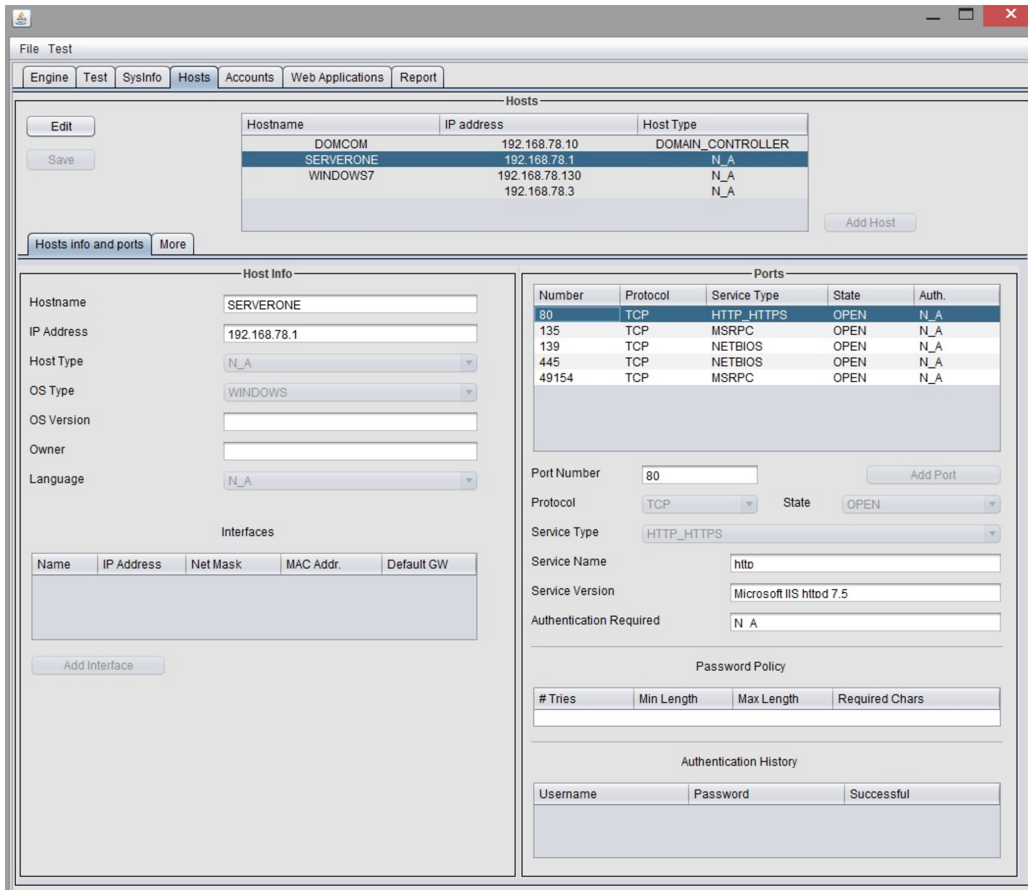


Figure 8.3: Hosts and services in the knowledge base.

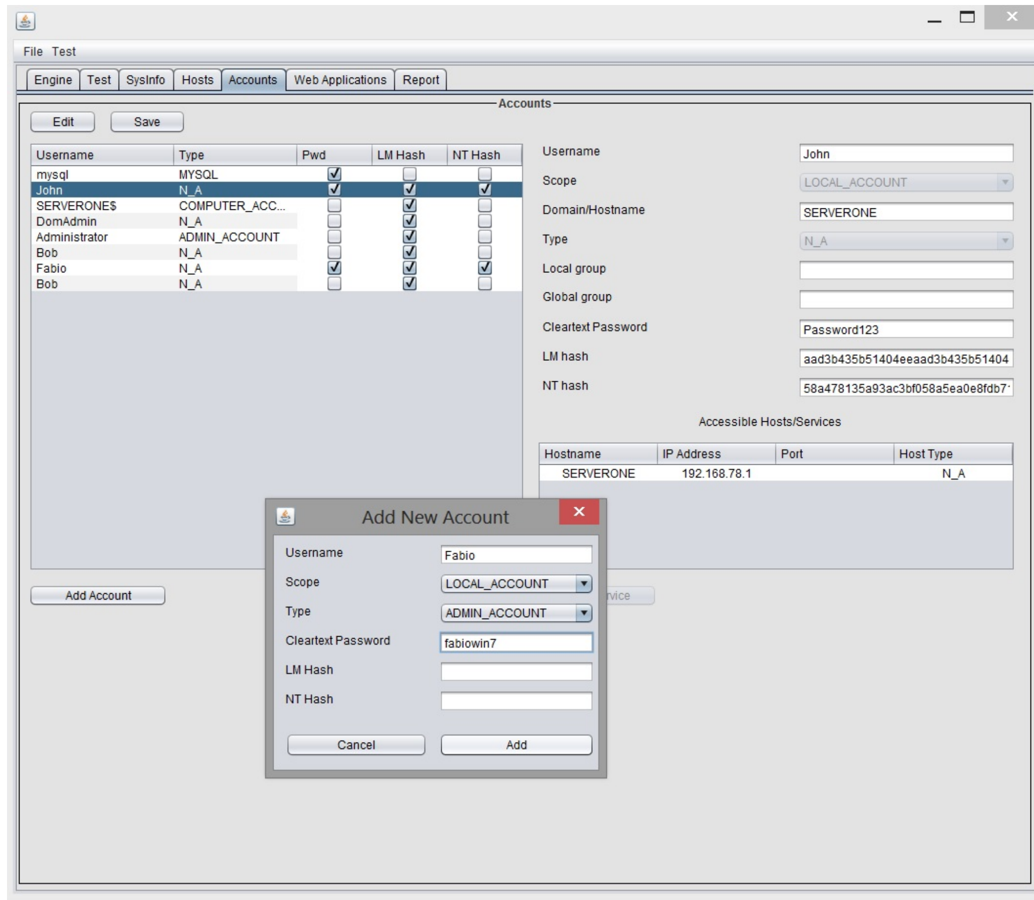


Figure 8.4: Manually adding an account to the knowledge base.

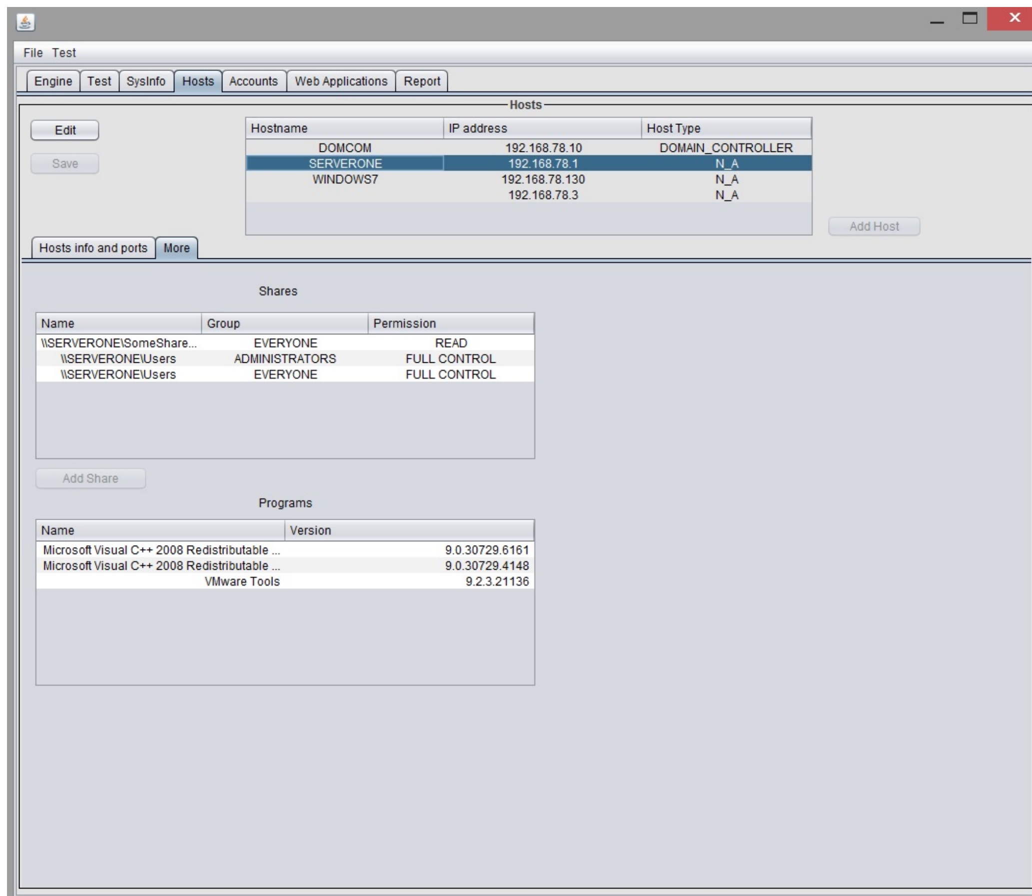


Figure 8.5: Network shares and programs collected from a remote machine.

Figure 8.6: Example of vulnerabilities reported by the tool.

8.3 Analysis of test results

The three tools used to test the system use very different approaches. As expected, diverse results were produced. These results can be summarized as follows:

- Autopwn was able to find several exploits matching the characteristics of the scanned services, but none of these exploits was executed successfully. This result is not very different from a simple scan result, since the only valuable outcome was the list of open ports and services on the different hosts.
- Nessus detected several vulnerabilities in the system and produced a thorough report of the vulnerable services encountered. However, issues such as default credentials and system dependencies are out of the scope of this tool.
- The new tool was able to collect several pieces of information about the system and detected a small number of vulnerabilities, such as weak credentials and account dependencies. Other data such as shared resources and programs installed on the hosts were presented to the penetration tester, who is responsible for spotting anomalies and misconfigurations. The tool (as currently implemented) did not include functions to automatically detect vulnerable services.

By analysing the results of the new tool, it was noted that the main feature that the tool lacks with respect to the other tools is the ability to determine whether a certain version of software is vulnerable to a known attack. Although the services and the programs installed on the hosts were detected, the tool lacks a vulnerability database that maps specific software versions to known security issues.

The same reasoning can be done for several other pieces of information that can be remotely collected from the hosts (once valid credentials are available), such as permissions on network shares that can lead to information exposure and/or weak access control on network resources. The detection of such vulnerabilities is simply a matter of extending the tool to check information that is already available in the knowledge base, and therefore would not require any further interaction with the system under test. Assuming that these type of extensions will be implemented in the future, the new tool can be considered able to report a sufficient number of vulnerabilities commonly encountered in large environments.

One important consideration is that the new tool required the user to manually add account information to the knowledge base, in order to allow the remote

harvesting of data. This approach is different from a fully black-box penetration test where the tester has no initial information about the system, and it is similar to an health check, as mentioned in chapter 7. This compromise allows the penetration tester to avoid using unsafe techniques, such as account brute-forcing or even testing for default credentials, and it is usually acceptable in production environments.

The main goal of this thesis was to determine whether the proposed approach to penetration testing can provide the same results as standard automated penetration testing procedures. Although two of the tools presented as related work in section 2.4 were not used during the test (Core Security's Impact and Immunity's Canvas), the tool developed following the suggested approach was able to collect *in a non-aggressive way* the information about services and other software necessary to detect the same vulnerabilities reported by other tools. The actual detection of these vulnerabilities is assumed to become part of the tool in a later version of the tool, *using extensions that do not require any further interaction with the system under test*.

Other goals that were specified at the beginning of this thesis project included the suitability of the new tool for usage in production environments, and the ability to manually influence the behaviour of the tool. These two goals can be considered to have been achieved. As shown during the test, the user of the tool is able to manually add data to the knowledge base, and this data allows the tester to skip a number of unsafe tests and therefore proceed with the assessment even in a production environment.

The other goal mentioned in chapter 1 stated that the tool should be able to provide the customer with an interactive report, and allow the customer to re-run specific parts of the testing process. Although not used during the test, this functionality is included in the current version of the tool. In particular, it is possible to (re-)execute individual steps from the list of available steps. However, as described in chapter 6, the execution of a step involves an exploration of the whole knowledge base, while the customer may be interested in performing a vulnerability check on a specific item (e.g. a specific host or service). The list of vulnerabilities checked by the tool (shown in figure 8.6) contains all the references to the items in the knowledge base. It is therefore straightforward to refine the execution to be limited to these desired items. Even though this functionality was not implemented in the version of the tool at the time of testing, future developers will find full support within the tool for the addition of this extension.

Chapter 9

Conclusions

This chapter presents the conclusions derived from the overall process followed during this thesis project. A number of suggestions for future work are also included in this chapter. The last section concerns aspects such as social impact, economics considerations, and ethic aspects related to this project.

9.1 Conclusion

The test of the implemented proof of concept showed that it is possible to execute a series on non-aggressive operations, and with the support of the penetration tester valuable information can be extracted from the system. This information can be used to detect weaknesses, anomalies, and misconfiguration of the system. Future extensions applied to the new tool can easily include these functionalities.

Therefore, we conclude that in the very limited and yet representative scenario that was simulated for use with our testing, the implemented application can be considered a more appropriate choice of penetration testing when the stability of the system under test is a major concern. However, support from the penetration tester would often be needed (in the form of manual insertion of data in the knowledge base). This is usually acceptable since the owner of the system under test often provides the penetration tester with (limited) access to the system, in order to limit the use of risky techniques.

If the only option is a thorough black-box testing, then some level of aggressiveness is needed to perform a complete penetration test (e.g. testing for default credentials, attempting limited brute forcing attacks, and/or exploiting a software bug). In situations where integrity and stability are of great importance,

it is preferable to provide the penetration tester with additional information (e.g. an account that can be used to access network resources), therefore performing a security assessment that is similar to a health check.

Another conclusion is that in scenarios with high availability requirements, i.e., those where the tester cannot afford to use risky procedures, automating the process of the penetration test is particularly challenging, because of the constant need to carefully limit the operations performed. Every new piece of information obtained must be analysed to determine whether the next step is appropriate. Therefore, we conclude that the preferred choice in such a situation would be a compromise between a fully automated solution and a manual test. The tool which has been implemented is an example of such a compromise, as it provides a framework that the penetration tester can use to select the operations to be executed and it allows the tester to refine the level of automation depending on the system to be tested.

9.2 Future work

This section presents a number of suggestions for possible improvements and extensions that could be introduced in the automated tool that was implemented during this thesis project. Some of these suggestions were already part of the initial idea of the complete tool, but were not implemented in the time available for this thesis project. Others are new concepts that were conceived during the development of the tool.

9.2.1 System State Change

In section 5.2.6 the issue of system state changes was pointed out. Although changes in the system are needed in order to verify that security problems have been solved, these changes may also cause inconsistencies and lead to incorrect behaviour of the tool. Effort should be directed to resolve this issue. In particular, the application should be able to detect changes that may affect its effectiveness (e.g. a re-assignment of a host's IP address). This could be achieved by periodically checking the system's state and keeping track of such changes, in order to keep the knowledge base up-to-date.

9.2.2 Additions

The current state of the application that has been developed (and described in section 8.1) only includes the functions necessary for a proof of concept. New steps, actions, and vulnerability checks should be added in order to cover additional aspects of penetration testing. For instance, some of the steps presented in table 6.1 are not fully implemented. Others (e.g. the steps concerning web applications) are not implemented at all.

Another addition could be the introduction of *agents*, i.e. software that can be installed on a remote machine and perform parts of the penetration test from the new location. Core Security's Impact, presented in section 2.4.2, utilizes the concept of agents to execute operations from different parts of the network under test. Agents allow the penetration tester to move from host to host in the network and acquire different perspectives on the system. This addition could include communication from the agents to the *parent* testing engine (the currently implemented tool), so that all the collected data could be stored and interpreted centrally.

9.2.3 Extensibility

Chapter 7 presented the list of changes that a developer must introduce in the tool to add a new functionality (step). These changes, although easy and relatively quick to undertake, require the developer to modify the code in multiple parts of the application. A future improvement could include the functionality to automatically perform all of the changes needed to add a specific step. This should be easy to implement because the code to be added always has the same structure, and only the implementation of the new step itself represents the real change. The new step could then be added as a new *module*, thus considerably simplifying the effort needed to add new steps to the penetration testing process.

9.2.4 Risk Definition

The risks that were assigned to steps, actions, and vulnerability checks were not defined in an methodical way. A consistent methodology should be introduced to assign risks to the different operations so that the tool could rely on these values to determine the overall risk of a certain procedure.

9.2.5 Efficiency

The behaviour of the tool is based on the logic defined in chapter 6, which was defined so that several components would be simple to maintain. The drawback of this approach is that the implementation is not optimized for efficient execution. In particular, the fact that the history must be checked for every action to be executed makes the application susceptible to scalability issues. For penetration tests performed on large systems and involving several operations, it is possible that the history checking could constitute a bottleneck. Future developers should take this into account and adjust the trade-off between code simplicity and efficient behaviour.

9.2.6 System Virtualization

This section suggests a different approach to penetration testing of systems in production environments, where stability and integrity are a major concern. Certain operations are considered to be potentially harmful to the correct functioning of the system under test. However, these operations might reveal important security vulnerabilities that are not always discovered by less aggressive techniques. These risky operations, however, could be executed on a *copy* of the system, thus facilitating testing without interacting with the original system itself. This requires an emulation of the original system in a virtual environment, since physically copying the system would not be feasible in most of the cases (see for example [30]).

The obvious advantage of this approach is that *any* technique, no matter how dangerous, can be used to test the system. However, there are a number of challenges to face. The amount of resources needed to emulate the environment could easily become too high, in terms of storage capacity and computational power. Moreover, reproducing every detail of the system requires a deep knowledge of every entity within the system. This knowledge may be difficult to acquire, and failing to reproduce the system *exactly*, makes the results of the test unreliable.

Despite these challenges, the virtualization approach seems to be an interesting topic to explore. Although very different in nature, this could be integrated with the tool developed during this thesis project. Only specific parts of the system could be virtualized, for example only those that are easier to emulate. The rest of the system could be tested using the methods currently included in the implemented tool.

9.3 Required reflections

The work carried out during this thesis project focused on automating the penetration testing procedures with particular attention on requirements such as stability and integrity of the system. From a more general perspective, this work aims at simplifying and increasing the accessibility of IT security software. By automating (complex) procedures, the target users of the software broaden considerably, and securing applications and services becomes easier. As a consequence, the average level of security in the IT world is potentially increased, as well as the trust that society gives to information systems.

From an economic perspective, the availability of tools to automate penetration tests in different scenarios makes it more affordable for small companies and institutions to assess the security of their systems. For instance, these tools could be pre-configured and used periodically to maintain an overview of the security of the system in order to see that it is up-to-date. The automation also gives an advantage to IT security companies, as it would require less resources to perform penetration tests.

As in most software used by security professionals, the tool developed during this thesis project may be used by malicious users, thus facilitating unauthorized attacks to information systems and networks. The Metasploit Framework is already very popular among non-experienced users, and has been the subject of debates concerning the legitimacy of making such a powerful tool readily available to any Internet user. An example of this discussion is Kurt Wismer's article: *Thoughts on Metasploit's impact* [28]. Tools such as Core Security's Impact and Immunitiy's Canvas have not been subjected to these type of critiques, mainly because their pricing has led to them not being readily available. The tool implemented as part of this thesis project is not (yet) powerful enough to raise such concerns. However, future extensions and improvements will cause the tool to become very effective, hence the issue of publicly releasing the software without any control is an important aspect to consider.

References

[1] Jason Andress and Ryan Linn. Coding for Penetration Testers: building better tools. Syngress, ISBN 978-1-59749-729-9, 2012.

[2] Massimiliano Montoro. Cain & Abel (version 4.9.43) (software). December 2011. Retrieved from http://www.oxid.it/cain.html

[3] JoMo-Kun, Fizzgig, pMonkey. Medusa (version 2.1.1) (software). May 2012. Retrieved from http://www.foofus.net/jmk/tools/medusa-2.1.1.tar.gz

[4] Johannes Gumbel. gsecdump (version 2.0b5) (software). 2010. Retrieved from http://www.truesec.se/sakerhet/verktyg/saakerhet/gsecdump_v2.0b5

[5] Johannes Gumbel. msvctl (version 0.3) (software). 2010. Retrieved from http://www.truesec.se/sakerhet/verktyg/saakerhet/msvctl_v0.3

[6] Bjorn Brolin. RunhAsh (version 1.0) (software). 2010. Retrieved from http://www.truesec.com/Tools/Tool/RunhAsh_v1.0_(x86)

[7] Rapid7, Inc. Metasploit Framework (version 4.5) (software). December 2012. Retrieved from http://www.metasploit.com/download/

[8] David Kennedy, Jim O'Gorman, Devon Kearns, and Mati Aharoni. Metasploit: The Penetration Tester's Guide. No Starch Press, ISBN 978-1-59327-288-3, 2011.

[9] Insecure.Com LLC. Nmap (version 6.25) (software). November 2012. Retrieved from http://nmap.org/download.html

[10] Gordon "Fyodor" Lyon. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Nmap Project, ISBN 978-0-9799587-1-7, 2009.

[11] The Wireshark Team. Wireshark (version 1.8.5) (software). January 2013. Retrieved from http://www.wireshark.org/download.html

[12] Ulf Lamping, Richard Sharpe, and Ed Warnicke. Wireshark User's Guide: for Wireshark 1.9, 2012 http://www.wireshark.org/docs/wsug_html_chunked

[13] PortSwigger Ltd. BurpSuite (version 1.5.04) (software). January 2013. Retrieved from http://www.portswigger.net/burp/download.html

[14] Yike Liu. WCDMA Test Automation Workflow Analysis and Implementation. Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, TRITA-ICT-EX-2009:6, April 2009 http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-91528

[15] Christopher Hadnagy. Social Engineering: The Art of Human Hacking. Wiley, ISBN-10 0470639539, December 2010.

[16] J. Postel. Transmission Control Protocol. *Internet Request for Comments*, RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[17] Abhinav Singh. Metasploit Penetration Testing Cookbook. Packt Publishing, ISBN 1849517428, 978-1849517423, 2012.

[18] TJ O'Connor. Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers. Syngress, ISBN 1597499579, 978-1597499576, 2012.

[19] Lee Allen. Advanced Penetration Testing for Highly-Secured Environments: The Ultimate Security Guide. Packt Publishing, ISBN 1849517746, 978-1849517744, 2012.

[20] Jeremy Faircloth. Penetration Tester's Open Source Toolkit, Third Edition. Syngress, ISBN 1597496278, 978-1597496278, 2012.

[21] Payment Card Industry Security Standards Council. PCI DSS Quick Reference Guide: Understanding the Payment Card Industry Data Security Standard version 2.0. October 2012 https://www.pcisecuritystandards.org/documents/PCI%20SSC%20Quick%20Reference%20Guide.pdf

[22] Dafydd Stuttard and Marcus Pinto. The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. Wiley Publishing, ISBN 978-0-470-17077-9, 2008.

[23] OWASP - The Open Web Application Security Project. Forced Browsing. Retrieved February 28, 2013 from https://www.owasp.org/index.php/Forced_browsing

[24] OWASP - The Open Web Application Security Project. Path Traversal. Retrieved February 28, 2013 from https://www.owasp.org/index.php/Path_Traversal

[25] Microsoft Corporation. Server Message Block (SMB) Protocol Versions 2 and 3. January 2013 http://msdn.microsoft.com/en-us/library/cc246482.aspx

[26] Ethan Cerami. Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL O'Reilly Media, ISBN 978-0-596-00224-4, 2002.

[27] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In Proceedings of the IEEE Symposium on Security and Privacy, 2012.

[28] Kurt Wismer. Thoughts on Metasploit's impact. November 3, 2011. Retrieved May 7, 2013 from http://anti-virus-rants.blogspot.se/2011/11/thoughts-on-metasploits-impact.html

[29] Microsoft. Microsoft Security Bulletin MS11-030 - Critical. April 12, 2011. Updated March 13, 2012. Retrieved May 9, 2013 from http://technet.microsoft.com/en-us/security/bulletin/ms11-030

[30] Subramanian, Lakshmi (KTH, School of Information and Communication Technology (ICT)). Security as a Service in Cloud for Smartphones. Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Trita-ICT-EX; 143, November 2011 http://kth.diva-portal.org/smash/record.jsf?pid=diva2:456509

Appendix A

Autopwn Results

[*] ===

[*] Matching Exploit Modules

[*] ===

[*] 192.168.78.10:135 exploit/windows/dcerpc/ms03_026_dcom (port match)

[*] 192.168.78.10:139 exploit/freebsd/samba/trans2open (port match)

[*] 192.168.78.10:139 exploit/linux/samba/chain_reply (port match)

[*] 192.168.78.10:139 exploit/linux/samba/lsa_transnames_heap (port match)

[*] 192.168.78.10:139 exploit/linux/samba/setinfopolicy_heap (port match)

[*] 192.168.78.10:139 exploit/linux/samba/trans2open (port match)

[*] 192.168.78.10:139 exploit/multi/ids/snort_dce_rpc (port match)

[*] 192.168.78.10:139 exploit/multi/samba/nttrans (port match)

[*] 192.168.78.10:139 exploit/multi/samba/usermap_script (port match)

[*] 192.168.78.10:139 exploit/netware/smb/lsass_cifs (port match)

[*] 192.168.78.10:139 exploit/osx/samba/lsa_transnames_heap (port match)

[*] 192.168.78.10:139 exploit/solaris/samba/trans2open (port match)

[*] 192.168.78.10:139 exploit/windows/brightstor/ca_arcserve_342 (port match)

[*] 192.168.78.10:139 exploit/windows/brightstor/etrust_itm_alert (port match)

[*] 192.168.78.10:139 exploit/windows/oracle/extjob (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms03_049_netapi (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms04_011_lsass (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms04_031_netdde (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms05_039_pnp (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms06_040_netapi (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms06_066_nwapi (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms06_066_nwwks (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms06_070_wkssvc (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms07_029_msdns_zonename (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms08_067_netapi (port match)

[*] 192.168.78.10:139 exploit/windows/smb/ms10_061_spoolss (port match)

[*] 192.168.78.10:139 exploit/windows/smb/netidentity_xtierrpcpipe (port match)

[*] 192.168.78.10:139 exploit/windows/smb/psexec (port match)

[*] 192.168.78.10:139 exploit/windows/smb/timbuktu_plughntcommand_bof (port match)

[*] 192.168.78.10:389 exploit/windows/ldap/imail_thc (port match)

[*] 192.168.78.10:389 exploit/windows/ldap/pgp_keyserver7 (port match)

[*] 192.168.78.10:445 exploit/freebsd/samba/trans2open (port match)

[*] 192.168.78.10:445 exploit/linux/samba/chain_reply (port match)

[*] 192.168.78.10:445 exploit/linux/samba/lsa_transnames_heap (port match)

[*] 192.168.78.10:445 exploit/linux/samba/setinfopolicy_heap (port match)

[*] 192.168.78.10:445 exploit/linux/samba/trans2open (port match)

[*] 192.168.78.10:445 exploit/multi/samba/nttrans (port match)

[*] 192.168.78.10:445 exploit/multi/samba/usermap_script (port match)

[*] 192.168.78.10:445 exploit/netware/smb/lsass_cifs (port match)

[*] 192.168.78.10:445 exploit/osx/samba/lsa_transnames_heap (port match)

[*] 192.168.78.10:445 exploit/solaris/samba/trans2open (port match)

[*] 192.168.78.10:445 exploit/windows/brightstor/ca_arcserve_342 (port match)

[*] 192.168.78.10:445 exploit/windows/brightstor/etrust_itm_alert (port match)

[*] 192.168.78.10:445 exploit/windows/oracle/extjob (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms03_049_netapi (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms04_011_lsass (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms04_031_netdde (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms05_039_pnp (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms06_040_netapi (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms06_066_nwapi (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms06_066_nwwks (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms06_070_wkssvc (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms07_029_msdns_zonename (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms08_067_netapi (port match)

[*] 192.168.78.10:445 exploit/windows/smb/ms10_061_spoolss (port match)

[*] 192.168.78.10:445 exploit/windows/smb/netidentity_xtierrpcpipe (port match)

[*] 192.168.78.10:445 exploit/windows/smb/psexec (port match)

[*] 192.168.78.10:445 exploit/windows/smb/timbuktu_plughntcommand_bof (port match)

[*] ==

[*]

[*]

[*] (1/58 [0 sessions]): Launching exploit/windows/dcerpc/ms03_026_dcom against 192.168.78.10:135...

[*] (2/58 [0 sessions]): Launching exploit/freebsd/samba/trans2open against 192.168.78.10:139...

[*] (3/58 [0 sessions]): Launching exploit/linux/samba/chain_reply against 192.168.78.10:139...

[*] (4/58 [0 sessions]): Launching exploit/linux/samba/lsa_transnames_heap against 192.168.78.10:139...

[*] (5/58 [0 sessions]): Launching exploit/linux/samba/setinfopolicy_heap against 192.168.78.10:139...

[*] (6/58 [0 sessions]): Launching exploit/linux/samba/trans2open against 192.168.78.10:139...

[*] (7/58 [0 sessions]): Launching exploit/multi/ids/snort_dce_rpc against 192.168.78.10:139...

[*] (8/58 [0 sessions]): Launching exploit/multi/samba/nttrans against 192.168.78.10:139...

[*] (9/58 [0 sessions]): Launching exploit/multi/samba/usermap_script against 192.168.78.10:139...

[*] (10/58 [0 sessions]): Launching exploit/netware/smb/lsass_cifs against 192.168.78.10:139...

[*] (11/58 [0 sessions]): Launching exploit/osx/samba/lsa_transnames_heap against 192.168.78.10:139...

[*] (12/58 [0 sessions]): Launching exploit/solaris/samba/trans2open against 192.168.78.10:139...

[*] (13/58 [0 sessions]): Launching exploit/windows/brightstor/ca_arcserve_342 against 192.168.78.10:139...

[*] (14/58 [0 sessions]): Launching exploit/windows/brightstor/etrust_itm_alert against 192.168.78.10:139...

[*] (15/58 [0 sessions]): Launching exploit/windows/oracle/extjob against 192.168.78.10:139...

[*] (16/58 [0 sessions]): Launching exploit/windows/smb/ms03_049_netapi against 192.168.78.10:139...

[*] (17/58 [0 sessions]): Launching exploit/windows/smb/ms04_011_lsass against 192.168.78.10:139...

[*] (18/58 [0 sessions]): Launching exploit/windows/smb/ms04_031_netdde against 192.168.78.10:139...

[*] (19/58 [0 sessions]): Launching exploit/windows/smb/ms05_039_pnp against 192.168.78.10:139...

[*] (20/58 [0 sessions]): Launching exploit/windows/smb/ms06_040_netapi against 192.168.78.10:139...

[*] (21/58 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwapi against 192.168.78.10:139...

[*] (22/58 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwwks against 192.168.78.10:139...

[*] (23/58 [0 sessions]): Launching exploit/windows/smb/ms06_070_wkssvc against 192.168.78.10:139...

[*] (24/58 [0 sessions]): Launching exploit/windows/smb/ms07_029_msdns_zonename against 192.168.78.10:139...

[*] (25/58 [0 sessions]): Launching exploit/windows/smb/ms08_067_netapi against 192.168.78.10:139...

[*] (26/58 [0 sessions]): Launching exploit/windows/smb/ms10_061_spoolss against 192.168.78.10:139...

[*] (27/58 [0 sessions]): Launching exploit/windows/smb/netidentity_xtierrpcpipe against 192.168.78.10:139...

[*] (28/58 [0 sessions]): Launching exploit/windows/smb/psexec against 192.168.78.10:139...

[*] (29/58 [0 sessions]): Launching exploit/windows/smb/timbuktu_plughntcommand_bof against 192.168.78.10:139...

[*] (30/58 [0 sessions]): Launching exploit/windows/ldap/imail_thc against 192.168.78.10:389...

[*] (31/58 [0 sessions]): Launching exploit/windows/ldap/pgp_keyserver7 against 192.168.78.10:389...

[*] (32/58 [0 sessions]): Launching exploit/freebsd/samba/trans2open against 192.168.78.10:445...

[*] (33/58 [0 sessions]): Launching exploit/linux/samba/chain_reply against 192.168.78.10:445...

[*] (34/58 [0 sessions]): Launching exploit/linux/samba/lsa_transnames_heap against 192.168.78.10:445...

[*] (35/58 [0 sessions]): Launching exploit/linux/samba/setinfopolicy_heap against 192.168.78.10:445...

[*] (36/58 [0 sessions]): Launching exploit/linux/samba/trans2open against 192.168.78.10:445...

[*] (37/58 [0 sessions]): Launching exploit/multi/samba/nttrans against 192.168.78.10:445...

[*] (38/58 [0 sessions]): Launching exploit/multi/samba/usermap_script against 192.168.78.10:445...

[*] (39/58 [0 sessions]): Launching exploit/netware/smb/lsass_cifs against 192.168.78.10:445...

[*] (40/58 [0 sessions]): Launching exploit/osx/samba/lsa_transnames_heap against 192.168.78.10:445...

[*] (41/58 [0 sessions]): Launching exploit/solaris/samba/trans2open against 192.168.78.10:445...

[*] (42/58 [0 sessions]): Launching exploit/windows/brightstor/ca_arcserve_342 against 192.168.78.10:445...

[*] (43/58 [0 sessions]): Launching exploit/windows/brightstor/etrust_itm_alert against 192.168.78.10:445...

[*] (44/58 [0 sessions]): Launching exploit/windows/oracle/extjob against 192.168.78.10:445...

[*] (45/58 [0 sessions]): Launching exploit/windows/smb/ms03_049_netapi against 192.168.78.10:445...

[*] (46/58 [0 sessions]): Launching exploit/windows/smb/ms04_011_lsass against 192.168.78.10:445...

[*] (47/58 [0 sessions]): Launching exploit/windows/smb/ms04_031_netdde against 192.168.78.10:445...

[*] (48/58 [0 sessions]): Launching exploit/windows/smb/ms05_039_pnp against 192.168.78.10:445...

[*] (49/58 [0 sessions]): Launching exploit/windows/smb/ms06_040_netapi against 192.168.78.10:445...

[*] (50/58 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwapi against 192.168.78.10:445...

[*] (51/58 [0 sessions]): Launching exploit/windows/smb/ms06_066_nwwks against 192.168.78.10:445...

[*] (52/58 [0 sessions]): Launching exploit/windows/smb/ms06_070_wkssvc against 192.168.78.10:445...

[*] (53/58 [0 sessions]): Launching exploit/windows/smb/ms07_029_msdns_zonename against 192.168.78.10:445...

[*] (54/58 [0 sessions]): Launching exploit/windows/smb/ms08_067_netapi against 192.168.78.10:445...

[*] (55/58 [0 sessions]): Launching exploit/windows/smb/ms10_061_spoolss against 192.168.78.10:445...

[*] (56/58 [0 sessions]): Launching exploit/windows/smb/netidentity_xtierrpcpipe against 192.168.78.10:445...

[*] (57/58 [0 sessions]): Launching exploit/windows/smb/psexec against 192.168.78.10:445...

[*] (58/58 [0 sessions]): Launching exploit/windows/smb/timbuktu_plughntcommand_bof against 192.168.78.10:445...

[*] (58/58 [0 sessions]): Waiting on 48 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 47 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 43 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 37 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 13 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 12 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 8 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 8 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 8 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 8 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 7 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 6 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 5 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 5 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 4 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 2 launched modules to finish execution...

[*] (58/58 [0 sessions]): Waiting on 0 launched modules to finish execution...

[*] The autopwn command has completed with 0 sessions

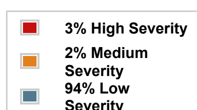
Appendix B

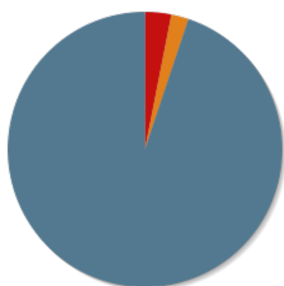
Nessus Executive Summary

Executive Summary:

>PRINT

TOP 10 HOSTS with ISSUES





Host	Status
192.168.78.1	High Severity problem(s) found
192.168.78.130	High Severity problem(s) found
192.168.78.10	High Severity problem(s) found

PLUGIN IDS	ISSUES
10736	26
11011	6
26917	3
24786	3
19506	3
25220	3
53513	3
20094	3
54615	3
10150	3
10394	3
10287	3
35716	3
11936	3
45590	3
10785	3
53514	3
57608	2
20870	2
22964	2
11002	2
25701	2
24260	1
46180	1
10884	1
10107	1
10663	1
43111	1
10397	1
11422	1
10114	1
43829	1

PLUGIN IDS	SEVERITY	# OF ISSUES	SYNOPSIS
53514	High	3	**MS11-030: Vulnerability in DNS Resolution Could Allow Remote Code Execution (2509553) (remote check)** Arbitrary code can be executed on the remote host through the installed Windows DNS client.
57608	Medium	2	**SMB Signing Disabled** Signing is disabled on the remote SMB server.
10736	Low	26	**DCE Services Enumeration** A DCE/RPC service is running on the remote host.
11011	Low	6	**Microsoft Windows SMB Service Detection** A file / print sharing service is listening on the remote host.
26917	Low	3	**Microsoft Windows SMB Registry : Nessus Cannot Access the Windows Registry** Nessus is not able to access the remote Windows Registry.
24786	Low	3	**Nessus Windows Scan Not Performed with Admin Privileges** The Nessus scan of this host may be incomplete due to insufficient privileges provided.
19506	Low	3	**Nessus Scan Information** Information about the Nessus scan.
25220	Low	3	**TCP/IP Timestamps Supported** The remote service implements TCP timestamps.
53513	Low	3	**Link-Local Multicast Name Resolution (LLMNR) Detection** The remote device supports LLMNR.
20094	Low	3	**VMware Virtual Machine Detection** The remote host seems to be a VMware virtual machine.
54615	Low	3	**Device Type** It is possible to guess the remote device type.
10150	Low	3	**Windows NetBIOS / SMB Remote Host Information Disclosure** It is possible to obtain the network name of the remote host.
10394	Low	3	**Microsoft Windows SMB Log In Possible** It is possible to log into the remote host.
10287	Low	3	**Traceroute Information** It was possible to obtain traceroute information.
35716	Low	3	**Ethernet Card Manufacturer Detection** The manufacturer can be deduced from the Ethernet OUI.
11936	Low	3	**OS Identification** It is possible to guess the remote operating system.
45590	Low	3	**Common Platform Enumeration (CPE)** It is possible to enumerate CPE names that matched on the remote system.
10785	Low	3	**Microsoft Windows SMB NativeLanManager Remote System Information Disclosure** It is possible to obtain information about the remote operating system.
20870	Low	2	**LDAP Server Detection** There is an LDAP server active on the remote host.
22964	Low	2	**Service Detection** The remote service could be identified.
11002	Low	2	**DNS Server Detection** A DNS server is listening on the remote host.

PLUGIN IDS	SEVERITY	# OF ISSUES	SYNOPSIS
25701	Low	2	**LDAP Crafted Search Request Server Information Disclosure** It is possible to discover information about the remote LDAP server.
24260	Low	1	**HyperText Transfer Protocol (HTTP) Information** Some information about the remote HTTP configuration can be extracted.
46180	Low	1	**Additional DNS Hostnames** Potential virtual hosts have been detected.
10884	Low	1	**Network Time Protocol (NTP) Server Detection** An NTP server is listening on the remote host.
10107	Low	1	**HTTP Server Type and Version** A web server is running on the remote host.
10663	Low	1	**DHCP Server Detection** The remote DHCP server may expose information about the associated network.
43111	Low	1	**HTTP Methods Allowed (per directory)** This plugin determines which HTTP methods are allowed on various CGI directories.
10397	Low	1	**Microsoft Windows SMB LanMan Pipe Server Listing Disclosure** It is possible to obtain network information.
11422	Low	1	**Web Server Unconfigured - Default Install Page Present** The remote web server is not configured or is not properly configured.
10114	Low	1	**ICMP Timestamp Request Remote Date Disclosure** It is possible to determine the exact time set on the remote host.
43829	Low	1	**Kerberos Information Disclosure** The remote Kerberos server is leaking information.

192.168.78.10

Scan Time

Start time:	Thu May 9 20:07:55 2013
End time:	Thu May 9 20:16:18 2013

Number of vulnerabilities

High	1
Medium	0
Low	37

Remote Host Information

Operating System:	Microsoft Windows Server 2008 R2 Standard Service Pack 1
NetBIOS name:	DOMCOM
IP address:	192.168.78.10
MAC address:	00:0c:29:79:e2:39

^BACK

192.168.78.130

Scan Time

Start time:	Thu May 9 20:07:55 2013
End time:	Thu May 9 20:10:46 2013

Number of vulnerabilities

High	1
Medium	1
Low	26

Remote Host Information

Operating System:	Microsoft Windows 7 Home
NetBIOS name:	WINDOWS7
IP address:	192.168.78.130

MAC address:	00:0c:29:d3:0f:45

^BACK

192.168.78.1

Scan Time

Start time:	Thu May 9 20:07:55 2013
End time:	Thu May 9 20:13:39 2013

Number of vulnerabilities

High	1
Medium	1
Low	29

Remote Host Information

Operating System:	Microsoft Windows Server 2008 R2 Enterprise Service Pack 1
NetBIOS name:	SERVERONE
IP address:	192.168.78.1
MAC address:	00:0c:29:3f:16:ed

^BACK