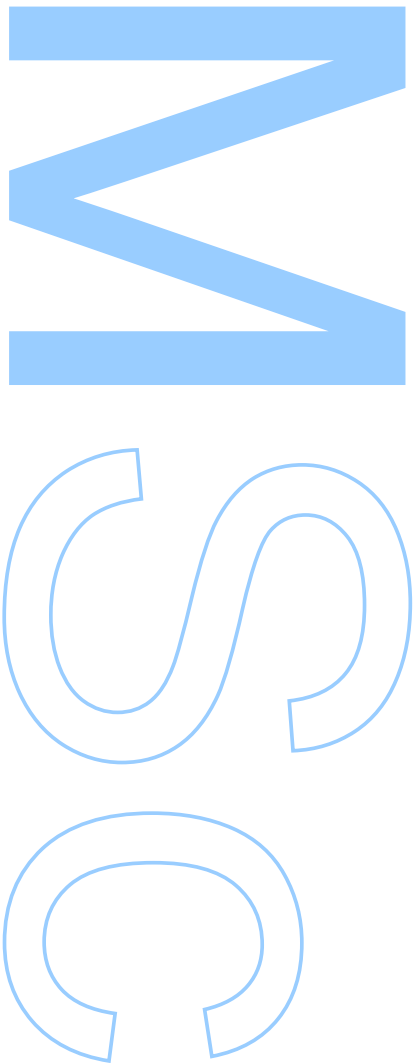
OrchRecon - A Distributed System for Reconnaissance and Vulnerability Scanning

Vítor Manuel Guedes de Oliveira Pinho
Mestrado em Segurança Informática
Departamento de Ciências dos Computadores
2020

Orientador
Rolando da Silva Martins
Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

Coorientador
André Martins Carrilho Costa Baptista
Professor Assistente Convidado
Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

UNIVERSIDADE DO PORTO

MASTERS THESIS

OrchRecon - A Distributed System for Reconnaissance and Vulnerability Scanning

Author:
Vítor PINHO

Supervisor:
Rolando MARTINS

Co-supervisor:
André BAPTISTA

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Computer Security

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências dos Computadores

January 8, 2021

Acknowledgements

I have a deep gratitude to my thesis supervisors, Professors Rolando Martins and André Baptista, for their incentive, help and contributions, without which this work would have been even more difficult. I am deeply grateful for welcoming my ideas and for providing all the guidance they did. Their knowledge and experience have encouraged me to accomplish this goal.

I wish to thank to my master's degree teachers as they provided some of the foundations for this endeavour. I could not forget to mention Professor Luís Antunes as the main responsible for putting this challenge among my plans.

I am also grateful to my colleagues for their support, help and for sharing their knowledge in the most varied topics. Among them, I will never forget the invaluable help that André Cirne gave me, far beyond what I could ask or expect. It was a precious help, indeed.

To my family, without their understanding and support this achievement would have been much more difficult. I am really grateful for all the encouragement that they continuously have given me.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências dos Computadores

MSc. Computer Security

OrchRecon - A Distributed System for Reconnaissance and Vulnerability Scanning

by Vítor PINHO

Nowadays there are a myriad of available tools to perform a penetration testing or a vulnerability assessment. While some of them focus on doing a single task, others combine the former in order to achieve better results. This approach, however more time-consuming, provides a wider coverage of the attack surface and a better understanding of a target's public exposure.

The trend to combine tools has one of its origins in the standards and regulations that focus on cyber security. As such, tool orchestration is a common practice and is experiencing an increasing attention either from the open source community, as well as from security companies that provide vulnerability assessment services. These tools aim, essentially, to uncover a target's assets dimension and public exposure and, after, check for their possible vulnerabilities.

Although the number of tools is considerable, there seems to be a tendency to stop maintaining the open source ones in the course of time. Additionally, each researcher has his preferences and tends to work with tools that match his kind of approach, leading to various possible combinations. Furthermore, most frameworks that combine multiple single tools are closed solutions in nature, which results in an inherent lack of flexibility.

OrchRecon proposes a solution that gives to the user an higher flexibility to use the tools of his choice, from a single bash script to those running inside a container, communicating through a distributed system conceived to take advantage of current cloud computation solutions. With this work, we designed and implemented a message-oriented middleware to support a tools framework for reconnaissance and vulnerability assessment.

With penetration testing and vulnerability assessment in mind, OrchRecon enhances the performance of the chosen tools, paralleling their execution and distributing the workload among the available resources, providing an optimised orchestration that, ultimately, leads to time savings by security researchers.

The preliminary results obtained in the performance tests reinforce these assumptions, even in scenarios of intensive use. Considering the Modules alone, we verified increases in performance from 60% to almost 300% by adjusting the parallelism level and when evaluating more intensive scenarios of simultaneous pipelines being performed, we measured increases from 60% on average to more than 100% when distributing the workload to two or three instances, respectively.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciências dos Computadores

Mestrado em Segurança Informática

OrchRecon - Um sistema distribuído para reconhecimento e identificação de vulnerabilidades

por Vítor PINHO

Actualmente, existe uma miríade de ferramentas disponíveis para executar um *Penetration Testing* ou a identificação de vulnerabilidades. Enquanto algumas se focam apenas numa tarefa, outras combinam as primeiras de forma a obter melhores resultados. Esta abordagem, embora mais demorada, possibilita uma cobertura mais ampla da superfície de ataque e uma melhor compreensão da exposição pública de um alvo.

A tendência para combinar ferramentas tem uma das suas origens nos *standards* e regulamentações focadas na cibersegurança. Por isso, a orquestração de ferramentas é uma prática comum e está a ser alvo de uma atenção crescente não só pela comunidade *open source*, mas também por empresas de segurança que providenciam serviços de análise de vulnerabilidades. Essas ferramentas têm essencialmente como função revelar a dimensão dos activos e a exposição pública de um alvo e, posteriormente, verificar as suas vulnerabilidades.

Embora o número de ferramentas seja considerável, parece existir uma tendência para que as ferramentas *open source* deixem de ser mantidas com o passar do tempo. Acresce ainda que cada investigador tem as suas preferências e tende a trabalhar com aquelas que se adequam à sua forma de abordagem, possibilitando várias combinações possíveis. Além disso, e considerando os *frameworks* que combinam várias ferramentas, as soluções fechadas são mais prevalentes, as quais podem sofrer de falta de flexibilidade.

A OrchRecon propõem uma solução que faculta ao utilizador uma elevada flexibilidade para usar as ferramentas da sua preferência, desde um simples *script* na *bash* até outras ferramentas executadas dentro de *containers*, comunicando através de um sitema

distribuído concebido para aproveitar as vantagens de algumas soluções de computação na nuvem. Com este trabalho, prototipámos um *message-oriented middleware* para possibilitar a execução de um *framework* de ferramentas para levar a cabo o reconhecimento e a identificação de vulnerabilidades.

Com o *penetration testing* e a identificação de vulnerabilidades em mente, a OrchRecon aumenta a performance das ferramentas escolhidas, paralelizando a sua execução e distribuindo as tarefas pelos recursos disponíveis, oferecendo um uso optimizado destas e um acréscimo na poupança de tempo dessas actividades.

Os resultados preliminares obtidos nos testes de desempenho reforçam essas hipóteses, mesmo em cenários de utilização intensiva. Considerando os Módulos isoladamente, verificámos aumentos de desempenho de 60% até cerca de 300% ajustando o nível de paralelismo e quando avaliámos cenários mais intensivos, no caso da execução de vários *pipelines* em simultâneo, pudemos aferir incrementos de 60% em média até mais de 100% ao distribuir o volume de execução por duas e três instâncias, respectivamente.

Contents

Acknowledgements iii

Abstract v

Resumo vii

Contents ix

List of Figures xi

List of Tables xiii

Listings xv

1 Introduction 1

2 State of the Art 3
2.1 Vulnerability assessment . . . 3
2.1.1 Legislation and Standards . . . 4
2.1.2 Security Controls Assessment . . . 7
2.2 Penetration Testing . . . 8
2.2.1 Pre-engagement Activities . . . 10
2.2.2 Discovery and Analysis Activities . . . 11
2.2.3 Attack Activities . . . 21
2.2.4 Reporting . . . 23
2.3 Web Vulnerabilities . . . 25
2.3.1 Injection . . . 25
2.3.2 Broken Authentication . . . 26
2.3.3 Sensitive Data Exposure . . . 27
2.3.4 XML External Entities (XXE) . . . 28
2.3.5 Broken Access Control . . . 30
2.3.6 Security Misconfiguration . . . 31
2.3.7 Cross-Site Scripting (XSS) . . . 32
2.3.8 Insecure Deserialization . . . 33
2.3.9 Using Components with Known Vulnerabilities . . . 34
2.3.10 Insufficient Logging and Monitoring . . . 35

3 Supporting Background Work **37**
3.1 Containerization . . . 37
3.2 Distributed systems . . . 38
3.2.1 Design issues . . . 39
3.2.2 Types of distributed systems . . . 41
3.2.3 Communication in distributed systems . . . 42
3.3 Reconnaissance automation . . . 45

4 OrchRecon **49**
4.1 Architecture overview . . . 50
4.1.1 Master . . . 52
4.1.2 Broker . . . 52
4.1.3 Pipeline Managers . . . 53
4.1.4 Database and Storage . . . 54
4.1.5 Module . . . 54
4.1.6 Scalability . . . 55
4.2 Implementation . . . 56
4.2.1 Setup . . . 56
4.2.2 Master . . . 57
4.2.3 Broker . . . 59
4.2.4 Pipeline Managers . . . 61
4.2.5 Database and Storage . . . 64
4.2.6 Module . . . 66
4.2.7 Distributed system . . . 69

5 Evaluation **73**
5.1 Modules Performance . . . 73
5.2 Pipeline Performance . . . 77

6 Conclusion **83**
6.1 Future Work . . . 84

Bibliography **85**

List of Figures

4.1 OrchRecon Diagram 51
4.2 Broker Diagram 52
4.3 Class Diagram 56
4.4 Dealer-Router pattern 70

5.1 httprobe - 1 vCPU 74
5.2 httprobe - 2 vCPU 74
5.3 httprobe - 4 vCPU 75
5.4 httprobe - Performance metrics 75
5.5 nuclei - 1 vCPU 76
5.6 nuclei - 2 vCPU 76
5.7 nuclei - 4 vCPU 76
5.8 Nuclei - Performance metrics 77
5.9 Pipeline - 1 vCPU 78
5.10 Pipeline - 2 vCPU 78
5.11 Pipeline - 4 vCPU 78
5.12 Pipeline Performance - 1 vCPU 80
5.13 Pipeline Performance - 2 vCPU 80
5.14 Pipeline Performance - 4 vCPU 80

List of Tables

2.1 Protection Mechanisms 16

3.1 Transparency Types 40
3.2 Reconnaissance Automation Tools 48

5.1 httprobe - Test results 75
5.2 Nuclei (Technologies) - Test results 77
5.3 Pipeline Performance - Test results 79
5.4 Pipeline - Test results 81

Listings

2.1 Check cookie . . . 34
4.1 Module configuration example . . . 55
4.2 Master class . . . 57
4.3 Heartbeat function . . . 58
4.4 Master check liveness . . . 58
4.5 Socket interfaces . . . 59
4.6 ConnectedElement class . . . 60
4.7 check liveness method . . . 60
4.8 Pipeline Manager start . . . 61
4.9 Manager class . . . 62
4.10 Modules validation . . . 62
4.11 Breadth First algorithm . . . 63
4.12 Database class . . . 64
4.13 Target class . . . 64
4.14 Save Target Function . . . 65
4.15 Folder creation . . . 65
4.16 Bucket file upload . . . 66
4.17 Command building . . . 66
4.18 Module properties . . . 67
4.19 Parallel execution . . . 68
4.20 Request-Reply approach . . . 69
4.21 Message polling . . . 70
4.22 Message transformation . . . 71
4.23 Queue implementation . . . 72

Acronyms

CEN European Committee for Standardization. 6

CENELEC European Committee for Electrotechnical Standardization. 6

CIS Center for Internet Security. 6, 7

CSC Critical Security Controls. 6, 7

CSIRT Computer Security Incident Response Team. 5

CSRF Cross-site Request Forgery. 27, 32, 33

CVE Common Vulnerabilities and Exposures. 11, 24, 73

CVSS Common Vulnerability Scoring System. 11, 24

CWE Common Weakness Enumeration. 11, 24

ENISA European Union Agency for Cybersecurity. 3, 5, 6

ETSI European Telecommunications Standards Institute. 5, 6, 8

EU European Union. 4–6

ICT Information and Communication Technologies. 4–6, 8, 9, 19

IDL Interface Definition Language. 43

ISMS Information Security Management System. 7

MOM Message-Oriented Middleware. 42, 44

NIST National Institute of Standards and Technology. 7–9

OS Operating System. 37–39, 41

OSINT Open Source Intelligence. 12, 13, 20, 45, 47, 49, 50

OWASP Open Web Application Security Project. 3, 8–10, 15, 19, 22, 25

PT Penetration Testing. 3, 8–12, 15, 17–21, 23–26, 28, 34, 36, 37, 49, 50

PTES Penetration Testing Execution Standard. 11, 16

RPC Remote Procedure Calls. 42–44

SDLC System Development Life Cycle. 8

Chapter 1

Introduction

OrchRecon is a reconnaissance and vulnerability assessment framework built on top of distributed infrastructure. Written in Python, it presents a message-oriented middleware that allows to integrate some chosen tools in a pipeline of tasks in order to do reconnaissance about the selected targets. Once the reconnaissance phase can be quite time consuming, there are several approaches to automate a series of tools that can produce results that will, after, feed the subsequent ones, reducing the human intervention as much as possible.

Starting from an initial idea from André Baptista and Miguel Regala to a tool that automates reconnaissance in a flexible way, we addressed this from the premise that a scalable distributed infrastructure allows for an increase in the global performance of the process of reconnaissance. Furthermore, current cloud computing solutions provide easy accessible resources that can shift these time- and resources-consuming tasks to external instances, providing a wide range of performance related options.

Driven by past experience, we assume that there are a set of well reputed tools to perform specific tasks that can be interlinked, as far as they can pass data to each other. Additionally, the choice of those tools should be kept flexible, as the criteria for it depends of an individual judgement or even of any circumstantial option.

We began this work by focusing on the vulnerability assessment activities from an organizational point of view. Departing from regulations, once there is an ongoing effort from national and international authorities to address cyber related problems, we understood the need for a consistent and structured approach. As such, some regulations, standards and frameworks provide some of the foundations to, consistently, act for a more secure environment.

We also reviewed the structure of penetration testings like their various phases and how they shall be conducted. As such, we considered the pre-engagement, discovery and analysis, attack and reporting activities. We deepened the discovery and analysis activities by considering the OWASP top ten web vulnerabilities and how they can be exploited.

In the supporting and background work, we wanted to contextualise some basics of distributed systems and containerization solutions. We focused in the communication between processes, namely through remote procedure calls and through message-oriented middleware. Furthermore, we analysed some similar applications from the perspective of automation and tool orchestration in order to observe how these features were implemented.

Concerning our solution, after providing an architecture overview, we present an insight into its main components. We resorted to some specific libraries to implement the messaging patterns and the database communication and used the Google's cloud computing resources to test it and to provide storage. Although its primary objective is to run in a distributed form, OrchRecon can also run locally, although not presenting the same performance increase.

At last we evaluated this prototype from a performance point of view. Although the results are preliminary, they corroborate the idea that OrchRecon can bring some efficiency to a time-consuming activity.

Chapter 2

State of the Art

Penetration Testing (PT) is an activity that occupies a prominent place in the context of cybersecurity. Nevertheless, it should be contextualised within a broader concern of risk management practices and controls and its role as a security control, by itself, should be emphasised and approached as a structured and comprehensive activity.

In this chapter we depart from a legislative point of view and verify how standards help to structure a response in accordance to it. Additionally, we detail how a PT may be structured, observing the perspectives of some standards, and provide some focus on OWASP Top Ten vulnerabilities.

2.1 Vulnerability assessment

Considering that a significant part of nowadays human interactions rely on technological means, and that it materialises in several contexts such as social, business, industrial, academic, among others, it seems obvious that there should be an important focus in determining that the supporting infrastructures work as intended regarding privacy, confidentiality, integrity or availability. Due to growing complexity, derived from the increased interconnection of systems, the variety of different technologies that must be integrated and also due to human errors, sometimes the outcomes are not as expected or anticipated. These unexpected outcomes usually are the visible part of systems flaws.

As defined by ENISA, "*a vulnerability is a weakness an adversary could take advantage of to compromise the confidentiality, availability, or integrity of a resource*", considering that, in this context, "*a weakness refers to implementation flaws or security implications due to design choices*" [1]. Thus, the need for vulnerability assessments seems unquestionable, as it

provides a process to verify the assumptions made on systems behaviour, identifying and quantifying the eventual vulnerabilities they might have [2].

To attest a growing focus on those concerns, we are confronted with cyber attacks news, in an almost daily basis, ranging from systems unavailability to data breaches, business disruption or ransomware events. As such, either due to legal or regulatory obligations or due to reputational aspects, institutions have the need to assess risk exposure as far as their ICT are concerned. Organizations shall, thus, implement a wide set of actions to conform with cybersecurity best practices.

To guide them trough this process, they can find several international standards that point to crucial actions that should be taken, such as ETSI TR 103 305: "CYBER; Critical Security Controls for Effective Cyber Defence" [3], which mimics a previous version of CIS Controls [4], ISO/IEC 27002:2013 "Information technology, Security techniques — Code of practice for information security controls" [5] or ISO/IEC 27007:2020 "Information security, cybersecurity and privacy protection — Guidelines for information security management systems auditing".

There is a general consensus that security assessments shall have some periodicity in order to detect, in a timely way, changes from the security point of view, demanding some pro-activity from organizations. These changes may appear for a variety of reasons, like updates in software, patch releases to detected vulnerabilities in software, network configuration changes or end of employee collaboration, among many others, although there should be considered, also, items such as physical facilities or human resources awareness.

Within the scope of this work, we will focus our attention in a fraction of this ecosystem, services exposed to the web.

2.1.1 Legislation and Standards

Especially since 2016, we find a growing concern from European authorities regarding cybersecurity. It has two main vectors: one, concerning personal data privacy, addressed by GDPR [6]; a second one, regarding the economic effects of defective Information and Communication Technologies (ICT) systems, as stated in EU NIS Directive [7]. Generically, it is recognised that networks, information services and systems have a vital role in a contemporary society and, as such, they need special attention to be protected from adversarial activities. These activities have no geographic boundaries, reason why European

authorities consider that, to tackle them, a common approach grounded on a common level of technical and technological preparation is needed.

GDPR aims to ensure the protection of EU citizens personal data, respecting their fundamental rights and freedoms in what data processing is concerned. For its part, the NIS Directive assumes the role of an European-wide cybersecurity legislation starting point, providing the legal foundations for a high common level of network and information security across all its countries. It establishes the requirements for each member state concerning cybersecurity, like the existence of CSIRTs, and creates mechanisms for a joint strategic and operational cooperation, imposing also security measures and security notifications for essential services and for digital services providers. Although its focus lays on these two groups, it sets the ground for good practices that shall be envisioned for any organization. Concerning its scope, the NIS Directive points to auditing and testing as means to certify that the organizations involved comply with the implementation of adequate measures and policies to address the inherent risk of their activities (Articles 15 and 16).

This Directive attributed a central role to the European Union Agency for Cybersecurity (ENISA) in supporting its implementation. With that goal in mind, it was published, more recently, a new version of EU Cybersecurity Act [8] that reformulates ENISA attributions, goals and organization, establishing also a framework for ICT cybersecurity certification. Its base attribution is the promotion of a coherent application of EU regulations in the cybersecurity field, grounded on a very high level of specialised knowledge and on the quality of the information it provides to the various stakeholders. In order to accomplish its role in the certification domain, ENISA is responsible to prepare candidate certification schemes on the request of the European Commission or the European Cybersecurity Coordination Group, ensuring that they are "*non-discriminatory and based on European or international standards, unless those standards are ineffective or inappropriate to fulfil the Union's legitimate objectives in that regard*". According to Cybersecurity Act, European cybersecurity certification schemes shall be designed to achieve a set of security objectives, including those that address vulnerability detection, to provide for the possibility of different assurance levels (articles 51 and 52) and shall include references to international, European or national standards applied in the evaluation phase (article 54).

There are three European Standardization Organizations responsible for the recognised European Standards: the European Telecommunications Standards Institute (ETSI),

the European Committee for Standardization (CEN) and the European Committee for Electrotechnical Standardization (CENELEC). ETSI deals with those related to Communications, Internet, Cloud Computing, Artificial Intelligence and Internet of Things technologies and has a formal collaboration with ENISA through a memorandum of understanding. It is responsible for a broad collection of standards concerning ICT and, as an European Standardization Organization, is also responsible for providing technical standards supporting EU directives and regulations. One of the standards, ETSI TR 103 456, provides guidance to conform with the legal measures and the implicit technical requirements imposed by the NIS Directive. Amid the mentioned technical requirements is a set of "*outcomes-focused cybersecurity risk management practices and controls to identify and protect assets, detect anomalous analyses and potential incidents, and respond to and recover from incidents that may impact network and information systems*" [9].

This points directly to the aforementioned ETSI TR 103 305: "CYBER; Critical Security Controls for Effective Cyber Defence", that establishes an implementable controls-based approach. It maps the CIS Controls from the Center for Internet Security (CIS) whose recognized industry best practices for securing ICT systems and data make it a reference, specifically to governmental bodies. CIS Controls map a wide set of international cybersecurity industry frameworks and can be used as a stand-alone tool or together with other frameworks [10, 11].

ETSI TR 103 305 [3] lists the top twenty Critical Security Controls (CSC) and includes information like:

- the importance of each one and how its absence can be exploited by an adversary,
- actions that organizations shall take to implement, automate, and measure effectiveness of each control,
- set of procedures and tools that enable implementation and automation,
- metrics and tests to assess implementation status and effectiveness.

These controls aim not only to avoid initial systems compromise or to prevent attacker's actions, but also to detect already-compromised systems and oppose to malicious running actions. As a consequence of the CSC implementation, it is expected that an effective cyber defence system is addressed observing five critical tenets: Offense informs defence, Prioritization, Metrics, Continuous diagnostics and mitigation and Automation.

This technical report is complemented with several parts that are reviewed more frequently. Part 1 [12] describes a set of technical measures to implement ETSI TR 103 305 and already refers to CSC version 7.0 released by CIS, which implies a minor change in the order of the controls established in the original document. Here, the CSC are characterized into "Basic" (controls 1 - 6), "Foundational" (controls 7 - 16) and "Organizational" (controls 17 - 20). The Basic controls are considered essential to a successful approach of this implementation and should be addressed at the very beginning of the process. Among them, is CSC 3: Continuous Vulnerability Management that envisions a set of actions that "*continuously acquire, assess, and take action on new information in order to identify vulnerabilities, remediate, and minimize the window of opportunity for attackers*". Within the Organizational controls we find CSC 20: Penetration Tests and Red Team Exercises that relates to CSC 3, as it produces information upon some detected vulnerabilities that should be addressed in the scope of the latter control. Its main purpose is to "*test the overall strength of an organization's defences (the technology, the processes, and the people) by simulating the objectives and actions of an attacker*". [12]

There are other standards pointing to similar security controls, like ISO/IEC 27002, Payment Card Industry Data Security Standard (PCI-DSS) or NIST SP 800-53, whose posture about due diligence concerning a compliant implementation of security controls stands out: "*Compliance is not about adhering to static checklists or generating unnecessary FISMA reporting paperwork. Rather, compliance necessitates organizations executing due diligence with regard to information security and risk management. Information security due diligence includes using all appropriate information as part of an organization-wide risk management program to effectively use the tailoring guidance and inherent flexibility in NIST publications so that the selected security controls documented in organizational security plans meet the mission and business requirements of organizations*" [13].

2.1.2 Security Controls Assessment

The security and privacy controls implemented within the scope of an Information Security Management System (ISMS) are the safeguards or countermeasures understood as the most appropriate to protect the confidentiality, integrity, and availability of an organization's information system [14], chosen through the implementation of its risk management process. There seems to exist a broad consensus that the only way to verify the appropriate implementation of security controls is through auditing and testing. Despite

the legally imposed security requirements and the compliance with standards, required by industry, countries, or internationally, regarding information security or privacy to some organizations, others also adhere to these practices as a mean to promote some distinctive feature on their services/products regarded as an added value when compared to their competitors.

Standards like ISO 27002 or those from ETSI and NIST advocate the implementation of procedures to ensure that adequate security controls are in place, validating their presence through a set of other security controls focused in vulnerability management processes. Penetration tests and red team exercises are included in that set and typify a specialised assessment conducted against systems in order to identify vulnerabilities that may be exploited by an adversary. Their purpose is to mimic adversarial and hostile actions to get an in-depth evaluation of a target's weaknesses or deficiencies. According to the cited standards, these tests should occur with some periodicity and should be performed by external entities, as well as by internal security teams (Red Teams) [3, 5, 14].

Additionally, the security controls assessment should also be understood as extended to systems development. With regard to that, OWASP proposes the definition of testing objectives through all the System Development Life Cycle (SDLC). Therefore, a set of actions is proposed for each phase of the SDLC in order to integrate security tests in the development workflows [15].

2.2 Penetration Testing

As already seen, Penetration Testing (PT) is a Security Control present among all the most relevant standards in use, nowadays, concerning information security. Historically, it has been used to verify the correct operation of the established defences, to validate the chosen security controls adequacy and to produce evidence about existing vulnerabilities through a set of actions that try to reproduce real-world attacks [12]. It can also be useful for determining how resilient are system towards real attack patterns and how sophisticated an attacker needs to be to successfully compromise a system or, from the defender's perspective, what should be the countermeasures to mitigate threats against a system and how efficient to detect attacks and respond appropriately they can be [16].

Penetration Testing is a proactive and authorised process to find security flaws in ICT systems, concerning applications, networks, configurations and human actions, that, alone or combined, allow the exploitation of vulnerabilities which may compromise the

whole system or parts of it. Typically, this implies the identification of methods for circumventing implemented security mechanisms and is, essentially, an interactive manual task, yet supported by a set of tools.

Performing a PT may be done with different approaching techniques, known as *white box* or *black box*. The first implies that the agent performing the test has a wide knowledge of the target, namely with access to the the source code, network topology or other information that may give him a previous insight. In a black box approach, sometimes the only information that is given is a target's name, leaving to the researcher all the work of gathering related information. This type of PT is considered to be more similar to a real world scenario, although less efficient and cost-effective than white box type. Sometimes, an intermediate approach is used, combining some elements from the two referred techniques, known as *gray box*. Additionally, NIST SP 800-115 proposes also that tests may be performed from different viewpoints. Therefore, PT should be performed either from an internal and an external perspective, considering the attacker within or outside the target's perimeter, as well as a scheduled event or as a covert process [16].

As such, the implementation of a PT poses, by itself, some risks like turning systems inoperable or with a degraded performance, disclosure of personal or sensitive information, among others. However, it is considered a valuable approach that provides insight about an organization's security posture, considering it is conducted after basic security mechanisms are in place and in the context of a comprehensive information security management program [12]. It is, also, a mean to enhance an organization's understanding of its ICT system and allows an estimation of the level of effort required to adversaries in order to breach the system safeguards [14]. Although there are several methodologies for the execution of a PT, some of them stand out and are frequently pointed as industry references: NIST Special Publication 800-115, PCI DSS Penetration Testing Guidance v. 1.1, OWASP Web Security Testing Guide, Open Source Security Testing Methodology Manual ("OSSTMM"), Penetration Testing Execution Standard and FedRAMP Penetration Testing Guidance v. 2.0.

NIST Special Publication 800-115, from the National Institute of Standards and Technology, provides practical guidelines for designing, implementing and maintaining technical information about security testing and, although approaching it as an overview of the key elements, it provides some emphasis on specific techniques [16]. PCI DSS Penetration Testing Guidance, from the Payment Card Industry Security Standards Council,

presents general guidance and guidelines for PT, focusing on its principal components, on the qualifications of a penetration tester, on its methodologies and on reporting guidelines [17]. OWASP Web Security Testing Guide is an initiative of the security community to design a complete testing framework. It focus on web applications security assessments throughout all phases of software development life cycle, as a strategic approach to enhance cyber security, and addresses PT as an element of a balanced approach on this subject. Its methodology results from industry experts consensus and details every aspect that needs attention in a web application testing activity [15]. The OSSTMM, from the Institute for Security and Open Methodologies, is developed in an open community effort, and subject to peer and cross-disciplinary review. Its purpose is to provide a scientific methodology for operational security tests over all channels (Human, Physical, Wireless, Telecommunications and Data Networks), that may be adaptable to almost any audit type, like PT, security assessments or vulnerability assessments among others, analysing and measuring how well security works. Its security testing methodology is grounded on the security of operations verification [18]. Penetration Testing Execution Standard is pointed by OWASP as one of the PT methodologies. Developed by Iftach Ian Amit, it proposes a structured approach in seven phases and is accompanied by technical guidelines, providing a rationale for testing activities and recommending appropriate tools [19]. FedRAMP Penetration Testing Guidance, from the Federal Risk and Authorization Management Program, provides guidance to PT with a standardized approach to security assessment, authorization and continuous monitoring for cloud products and services. This federal organization is responsible for authorizing cloud service providers to sell their products to governmental agencies in USA [20].

Although using different grouping options or terminology, in general, these references may be summarized in four major phases: (i) Pre-engagement Activities; (ii) Discovery and Analysis Activities; (iii) Attack Activities; (vi) and Reporting.

2.2.1 Pre-engagement Activities

Pre-engagement activities encompass a set of steps leading to the preparation of a successful PT. During this phase, the parts involved agree on the type of test to be performed, how it will be done and what it will target. According to NIST SP 800-53A [14], the PT scope should be defined in a clear way, considering topics like the environment subject to test (e.g., facilities, users, organizational groups), the attack surface (e.g., servers, desktop

systems, wireless networks, web applications), the threat sources to simulate (e.g. internal attacker, casual attacker, single or group of external targeted attackers, criminal organization) or specific objectives for the simulated attacker, among others. Assets depending on third parties should be clearly identified and a testing permission should be granted from them. Furthermore, the organization being tested should provide appropriate documentation about the assets defined in the scope. Depending on the type of test to be performed (white-box, black-box, gray-box), previous PT reports, software documentation or network diagrams may be provided.

Another crucial topic of this phase is the agreement on the *Rules of Engagement* without which any test should not start. These rules provide detailed guidelines regarding the execution of a PT, considering topics like times of day for testing, duration of tests, degree of exploitation, evidence handling, communication channels during ongoing activities, third-party or cloud environments permissions to test, identification of potential risks, testing locations and reporting requirements. Furthermore, a success criteria should be established for each environment subject to test, in order to prevent the possibility of testing boundaries being exceeded [14, 17].

An important topic that must be considered in the Rules of Engagement is how the vulnerabilities should be categorised and ranked. A common approach across industry is to resort to the Common Vulnerability Scoring System (CVSS) framework and to listings like Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) [21–23].

Finally, a permission to test document acknowledging the awareness of the test, the risk of system instability or inoperationality should be signed by the management, addressing the legal implications of the activities to perform.

2.2.2 Discovery and Analysis Activities

Intelligence Gathering is considered to be at the base of a successful PT and consists in acquiring as much information as possible on a target, in order to map its cyber presence and to set future steps for the subsequent phases. Taking into account certain real-world constraints such as time, effort and access to information, among others, Penetration Testing Execution Standard (PTES) defines three levels for intelligence gathering as it allows to clarify the expected results according to the chosen approaches [24]:

- Level 1 - a compliance driven approach, obtained almost entirely with automated tools;
- Level 2 - use of automated tools from Level 1 complemented with some manual analysis in order to get a good understanding of the target, including information such as physical location, business relationships, etc;
- Level 3 - in the context of a full-scope PT, comprising information obtained in Levels 1 and 2 and considering also relationships on Social Networking platforms, heavy analysis and deep understanding of target's cyber presence.

According to Faircloth [25], within intelligence gathering we may consider Reconnaissance and Scanning and Enumeration activities. While the first relies essentially in non-intrusive methods, Scanning and Enumeration implies interaction with the systems being tested. However, and assuming the referred activities together in this process, we will consider the following phases concerning discovery and analysis activities: (i) Open Source Intelligence (OSINT) gathering, (ii) Network Footprinting, (iii) Application Identification, (iv) Protection Mechanisms Detection, (v) Human reconnaissance, (vi) Vulnerability Analysis and (vii) Threat Modeling.

OSINT gathering

With reconnaissance methods, a "real-world" target (a company, corporation, government, or other organization) is mapped into a cyberworld one, defined as a set of relevant DNS names [25]. Publicly available resources are the main information source for it and, sometimes, new assets, out of the established scope and rules of engagement, are revealed, posing the need of clarification with the costumer before engaging into scanning and enumeration activities.OSINT gathering aims to collect relevant information about a target such as its organizational structure, its cyber presence or how it relates with other organizations from publicly available resources. However, it may be outdated, incomplete or even manipulated, for what it should be confirmed and validated using different sources.

Beyond organization level intelligence, related individuals information is also analysed, such as e-mail addresses or nicknames in Social Networks. On the organizational level, the following topics should be investigated: physical locations, information related with organization activity (partners, clients, competitors), organizational charts,

electronic documents, infrastructure assets (owned network blocks, e-mail addresses, external infrastructure profile, used technologies, remote access, applications usage, defence technologies, code in public repositories) and financial information. In parallel, employees information like their social networks profiles, personal history, internet presence, physical location, contacts, mobile footprint and background checks shall also be gathered. Essentially, the collected information with an adequate analysis can provide a set of possible entry-points into the target.

To accomplish that, search engines are some of the most relevant tools and, when used with search operators[1], they can output much more granular information. Data mining tools, like Maltego [26] or The Harvester [27] are also becoming more relevant and turn data from sources like financial databases, business reports and web archives into relevant information. Other common tools include Netcraft[28] and command-line tools like WHOIS.

Network footprinting

Network footprinting derives IP/host name combinations from the DNS domains outputted from OSINT gathering. It already implies some interaction with the target and consists also in probing for services or devices in order to get, at the end, a prioritised set of possible entry-points. Among others, the main activities are Port Scanning, Fingerprinting, Banner Grabbing, SNMP Enumeration, Zone Transfers, SMTP Mail Bounce, DNS Discovery, Reverse DNS, DNS Bruteforce, Web Application Discovery and Virtual Host Detection & Enumeration [24].

Port scanning probes an host for active services and open ports, allowing an attacker to get a set of reachable resources. The most used tool to perform it is nmap [29] which has several options that range from stealthy tests to highly ”noisy” ones.

After identifying active services, it is important to determine which operating systems (fingerprinting) and applications version (banner grabbing) the attacker is facing. It consists in sending packets to the target and analyse its responses, matching them with a set of known services responses, with tools like Telnet, nmap or netcat.

Simple Network Management Protocol (SNMP) is a UDP based protocol for monitoring and managing a variety of systems, including network devices and servers. It is widely deployed, controls some of the most important devices or systems on a network

[1] A search operator is a special keyword that extends the capabilities of regular search queries, and can help obtain more specific results. They generally take the form of operator:query. Some commonly supported search operators are: site, inurl, intitle, intext, inbody, filetype [15]

and, with the appropriate queries, returns considerable information that allows enumeration of hosts and its services [30], making SNMP enumeration an important step to perform.

Although currently DNS zone transfer has a limited use, it should integrate a comprehensive information gathering process, as it may return a list of DNS entries for the target domain. Also known as AXFR, it is a type of DNS transaction that may operate in two modes, full (AXFR) and incremental (IXFR), and is usually done with tools like host, dig or nmap.

Some additional DNS discovery should be tried concerning variations of the main domain. Described as *Domain name expansion* by Faircloth [25], it builds on two assumptions to perform such step: (i) if a target has a certain domain name is expected it also owns similar-sounding ones; (ii) if a target domain name exists in a certain top-level domain (TLD) it may exist in a different TLD.

Another important step is to probe the already obtained IP for additional DNS, performing a Reverse DNS query, in order to find valid server names belonging to the target. This step is usually extended to probe entire netblocks and may lead to some conclusions, considering the outputs: (i) a range is likely to belong to a target if there is a relative density of hosts with similar DNS names in it; (ii) if other DNS names appear in a range known to belong to the target, those domains may also be relevant targets and related to the original [25].

Considering that hosts name in an organization usually conform with some kind of convention, it turns DNS Bruteforce as a natural step during reconnaissance. It is performed by querying DNS of a list of potential host names and observe if they are resolved. This allows to discover host names that are not publicly known.

Another approach with limited success rate is SMTP mail bounce. An assumed non-existent e-mail address in the target domain is used to send a normal e-mail message with the expectation that it is rejected by the server. When it happens, a normal behaviour is an automated message being sent back to the sender with some basic problem description. However, sometimes this message retrieves also information about the SMTP server like software and version or even the host names and IP addresses of the servers that handled it, providing valuable information about target's infrastructure, its architecture and how critical services are hosted. Flemish ethical hacker, Inti De Ceukelaire, expanded this topic to abuse some e-mail based services like support inboxes, billing systems, printing

services or ticket trackers in order to map e-mail aliases and retrieve sensitive information [24, 25, 31, 32].

An important step in reconnaissance is to identify all the web applications exposed to an external actor. A considerable number of applications have documented vulnerabilities and attack strategies while others, intended for an internal use, end up being publicly accessible and may present misconfigurations that could be exploited later. The existence of vulnerable plugins shall also be investigated as they, often, contain more vulnerabilities than the main application [15, 24].

One of the ways web hosting providers have for making web sites available is through Virtual Hosting, which consists in using web servers to host more than one domain name on the same machine. It can be done either in a name-based or in an IP-based virtual hosting way, resulting in having several host names on the same IP address or having separate IP addresses for each host, respectively. As such, Virtual Host Detection provides insight about target's hosting type at the same as it allows to check if all detected hosts effectively belong to the target's scope.

Application Identification

Sometimes, the scope of a PT is a web application by itself. The rationale underlying its testing is the same with the appropriate adaptations. OWASP Testing Guide [15] provides a very detailed framework concerning web applications security testing. As such, it proposes also an information gathering phase intended to provide an understanding of the application's logic. During this step, beyond (i) identifying application entry points and (ii) mapping execution paths, (iii) Search Engine Discovery Reconnaissance for Information Leakage, (iv) Web Server Fingerprinting, (v) Web Server Metafiles review for Information Leakage, (vi) Web Server applications enumeration, (vii) Webpage Comments and Metadata review for Information Leakage, (viii) Web Application Framework fingerprinting, (ix) Web Application fingerprinting and (x) application architecture mapping are the other activities proposed.

Additionally, active testing is also prescribed, comprising a set of tests distributed by 10 sub-categories which address 91 controls:

- Configuration and Deployment Management Testing
- Identity Management Testing
- Authentication Testing

- Authorization Testing
- Session Management Testing
- Input Validation Testing
- Error Handling
- Cryptography
- Business Logic Testing
- Client Side Testing

The guide provides a structure for each test, including a summary, test objectives, how to test and remediation sections. It also details, when applicable, different approaches concerning the test type: white-box, gray-box or black-box.

Protection Mechanisms Detection

In order to maximize the effectiveness and efficiency of the exploitation phase, it is important to get a deep understanding of the implemented protection mechanisms. While there are some indications to disable some of such protections during vulnerability assessments, like in PCI DSS [17] concerning intrusion protection systems (IPS), intrusion detection systems (IDS) and web application firewalls (WAF), deep knowledge of their presence would allow to test target's systems bypassing some of them, thus minimizing the detection ratio.

Table 2.1 illustrates the mechanisms that PTES recommends to identify and map, according to their context within the established scope [24]:

Network based	Host Based	Application Level	Storage
Packet filters	Stack / Heap Protections	Application Protections	Host Bus Adapter
Traffic shaping devices	Application Whitelisting	Encoding Options	LUN Masking
Data Loss Prevention (DLP) Systems	AV / Filtering / Behavioral Analysis	Potential Bypass Avenues	Storage Controller
Encryption / Tunneling	DLP Systems	Whitelisted Pages	iSCSI CHAP Secret

TABLE 2.1: Protection Mechanisms

Human reconnaissance

Human reconnaissance is focused on obtaining extra information from the target organization through their employees rather than a social engineering approach, which will be addressed later. One of the approaches is to identify places where people from the target

organization post information about themselves or where information related to them is available. Relationships, contacts information, blog posts, social networks or any relevant document are topics of interest. Beyond search engines or tools like Maltego, this step usually needs direct interaction, either physical or verbal, in order to get relevant information and, in parallel, establish an empathetic vector that may be more cooperative in what privileged information is concerned.

Another approach, mainly on more sensitive targets, could involve direct observation or electronic surveillance as means to establish behavioural patterns such as working routines, dress code, among others.

This step may confirm intelligence gathered previously and help to rank it according to its relevance in topics like key employees, partners, suppliers, procedures, access paths, among others.

Vulnerability Analysis

After the previous steps are accomplished, there is a need to turn the gathered information into security intelligence in order to make informed decisions. OSSTMM presents the concept of actionable intelligence, the final part of the analysis process, as information extracted from facts that can be used to make decisions, which can influence risk analysis, threat modeling or attack paths. To acquire such level of information, the analysis builds on the information gathered, applying the concept of critical security thinking, referred as the practice of resorting to logic and facts to form an idea about security [18]. OSSTMM poses a great emphasis on the quality of the analysis process, applying a scientific method approach, to avoid biased testing, and alerting for analysis errors derived from incomplete and inconsistent testing.

During this step, an analyst will evaluate the services, applications, and operating systems of scanned hosts and will confirm which results, alone or combined, may be considered a vulnerability, comparing them with vulnerability databases or even with the tester's own knowledge or personal database. The analysis will categorize vulnerabilities, determine their causes and identify false positives. This categorization, as proposed in NIST SP 800-115, may be done according to the implemented security controls which may facilitate vulnerability analysis, remediation and documentation. Another goal is to immediately warn the organization about any critical vulnerabilities that shall be urgently addressed, independently of when it was detected during the assessment [16]. Inclusive, and depending on the Rules of Engagement, suspend the PT could be a measure to take.

Scanning and Enumeration activities rely, till some extent, on automated tools. Some of these assessment tools, while scanning networks or web applications, compare the obtained responses with known signatures of vulnerabilities, identifying their presence with some degree of probability. However, because their high accuracy is not guaranteed all the time, their results need to be cross-checked with the results of other tools and with manual testing to ensure a proper validation and to isolate false positives. Although more time-consuming, manual examination tends to provide more accurate results than comparing results from multiple tools and, simultaneously, allows unknown or less researched vulnerabilities to be detected [16, 33].

Furthermore, after validating a vulnerability belonging to the scope of the PT, its potential exploitability shall be investigated, in order to address it in the Attack phase. One of the ways to analyse a potential vulnerability is to replicate the same environment in a testing lab. Additionally, related documentation, manuals, vendor-issued information, security research and known exploits as well as default configurations or passwords, hardening settings and common misconfiguration errors shall be taken into account. This kind of approach has the potential to find more reliable exploits, decreasing inaccurate results and the risk of disrupting target infrastructure [33]. Regardless the goal of a PT being the identification of vulnerabilities, the identification of their root causes should also be addressed, in order to enhance the organization's overall security posture, as it may expose the lack of some security requirements or security controls that should have been putted in place [16]. Some common root causes, as highlighted in NIST SP 800-115, include:

- Insufficient patch management, such as failing to apply patches in a timely fashion or failing to apply patches to all vulnerable systems
- Insufficient threat management, including outdated antivirus signatures, ineffective spam filtering, and firewall rulesets that do not enforce the organization's security policy
- Lack of security baselines, such as inconsistent security configuration settings on similar systems
- Poor integration of security into the system development life cycle, such as missing or unsatisfied security requirements and vulnerabilities in organization-developed application code

- Security architecture weaknesses, such as security technologies not being properly integrated into the infrastructure (e.g., poor placement, insufficient coverage, or outdated technologies), or poor placement of systems that increases their risk of compromise
- Inadequate incident response procedures, such as delayed responses to penetration testing activities
- Inadequate training, both for end users (e.g., failure to recognise social engineering and phishing attacks, deployment of rogue wireless access points) and for network and system administrators (e.g., deployment of weakly secured systems, poor security maintenance)
- Lack of security policies or policy enforcement (e.g., open ports, active services, unsecured protocols, rogue hosts, weak passwords

Besides comparing applications responses with know signatures, vulnerability analysis shall test for all known kinds of potential threats, related to the context of the target. OWASP releases, periodically, a list of the most prevalent threats in web applications, whose latest version, OWASP Top Ten 2017 [34], will be analysed later. Common approaches include fuzzing techniques, trying for unexpected inputs, discover injection points of unsanitized inputs, testing for security misconfigurations or trying for faulty access controls, among others.

Threat Modeling

With the information previously gathered, the following steps need to be prepared with a cost/benefit relation in mind, prioritising the potential threats according to its inherent risk. Threat modeling provides the necessary approach as it implies a process to construct threat scenarios to be later explored in the exploitation phase. Although there are similarities with a Threat Modeling process implemented in a typical information security risk management program, on this context the focus is, essentially, on creating an adversary profile, identifying plausible threat events and developing threat scenarios. This threat-centric model is one of the purposes a cyber threat model can serve, namely for PT [35].

There are a significant number of threat modeling frameworks with different emphases or specific to certain ICT domains. From the analysis of Bodeau et al. [35], CBEST approach is depicted as one of the most useful for the context of a PT. This framework

was developed by the Bank of England in 2016 and is focused on the identification of specific threat actors and their common attack patterns using the available gathered information. Each threat actor is identified and characterized according to motivation, capabilities, skill level and sophistication, persistence and risk sensitivity, among others and its *modus operandi* is identified, when possible. CBEST approach allows the conception of an assertive threat actors model in what predicting likely threat events is concerned. However, despite the chosen framework, threat modeling should be consistent on the representation of threats, their capabilities and their qualifications, aligned with the organization point of view, and must guarantee testing reproducibility in a consistent way [36].

Regardless of the type of PT being performed, the threat model should always be based on an attacker's perspective, considering OSINT gathered [36] and other relevant information like previous PT reports, regular vulnerability assessment reports or previous security breaches, when appropriate. Thus, risk estimation shall evince the likelihood of threat events or scenarios to materialise, concerning the assumptions made over the most relevant vulnerabilities and their impact on target's assets, as well as the costs it may bring to the organization. To build such scenarios, attack trees or attack graphs are widely used techniques, not only for cyber systems, but also for cyber-physical and physical systems, providing a structuring framework [35]. They provide an easily understandable testing path assuming that there is a deep knowledge of the system being tested and the related threats

A well established threat model will help in prioritising tasks, enhance a strategic approach and will be meaningful for stakeholders as it gives support to resource allocation, projects the attack surface reduction and addresses the value of the already implemented security controls.

Limitations

Although a PT is, desirably, a structured and matured process, it still has some limitations. The first one is related with the circumstance that a PT represents the reality of a target at a certain moment in time. Furthermore, that representation is built by an individual, or a group of individuals, that may overvalue some approaches at the expenses of others and that rely on tools and methods, themselves with their own limitations. Additionally, individuals and tools usually search for known vulnerabilities, by which unknown threats may remain undetected.

Understanding these limitations should reinforce the idea that vulnerability management shall be a permanent activity in an iterative improvement process. It shall incorporate a wide range of approaches, so the weaknesses of some of the approaches may be mitigated by the others.

2.2.3 Attack Activities

Exploitation

Discovery and analysis activities should produce a set of vulnerabilities that must be tested during this phase. This is a very dynamic process whose primary objective is to circumvent the existing security controls and get access to the target system and its presumed secured assets. Since it can take several paths, a previous threat analysis must exist, in order to prioritise found vulnerabilities and choose the appropriate approaches as far as their exploitation is concerned [37]. Typical attack vectors include web applications, infrastructure vulnerable assets like misconfigured servers or vulnerable exposed services, wireless access points or mobile devices like smartphones. Although they are very diversified, the end goal of the exploitation phase is to get an improper access to target's resources included in the rules of engagement and try to maintain that access in the future, which will be addressed in the post exploitation step.

In a wide scope PT, social engineering attacks shall be included in the rules of engagement, as it is believed that the human factor is the weakest link in the security context [38]. Its goal is to test the human factor, concerning users security awareness related to the applicable security controls [16], and can be performed through several means, either digital or not. In its simplest form, social engineering attacks try to mislead users into revealing sensitive information or into installing malicious software on the target's infrastructure, enabling the attacker to circumvent existing security controls that could block his intentions. Phishing is one of the most prevalent social engineering attacks, where the attacker attempts to deceive a user through an authentic-looking e-mail, requesting information such as credit card numbers, Social Security numbers, user ID or passwords, or trying to direct them to a bogus web site in order to collect some kind of information or to make him install some kind of malware.

Other attacks need interaction with the target on a technological level, either against the infrastructure or against some application. Based on the information previously gathered, it is possible to find already known exploits that may be tried, although they could need some costumization to successfully materialise the attack. Centering our analysis on web applications, there is a baseline for vulnerabilities assessment established in the OWASP, widely adopted by the industry, named *Top Ten*, which is detailed later (see section 2.3). Despite a more recent version being in preparation, the current reference was published in 2017 and highlights the most sensible vectors in a web application that need to be assessed [34].

In this phase, an attacker validates his findings from the discovery and analysis activities, exploiting the verified vulnerabilities, which demands a clear understanding of the context to attack. The manual testing component is higher, mostly with web applications, and must consider the several components involved, how their vulnerabilities can be chained in order to increase the exploitation degree or even existing exploits that may be tailored according to the intended purposes [37]. As such, when a vulnerability is found, the next step is try to leverage it to other attacks or to find extra information about the target, reconciling these findings with the previously gathered information. In this case, further analysis and testing is required. One of the main objectives for an attacker is related to privilege escalation, as it will facilitate system takeover. In this context, the extent of the system's compromise must be determined, in order to address the post-exploitation step [16, 38].

Post Exploitation

After a successful exploit, it is important to determine the relevance of the compromised machine or application, identifying sensitive data, analysing infrastructure, assessing exfiltration viability, trying for persistence and for further penetration into the infrastructure. However, depending on the rules of engagement, not all actions may be allowed, reason why the tester shall gird up to mutually agreed actions to avoid unnecessary risk for the client's systems and for himself, from a legal point of view [39].

From the new perspective of a compromised resource, additional information can be gathered and analysed to determine the full extent of compromise. Information related to the network configuration of a compromised machine may reveal additional subnetworks, servers or name servers, identifying new targets for further testing. Concerning network services, it may be possible to detect previously unidentified services due to the

presence of filtering systems, identify VPN connections that interact with unknown systems as well as enumerate user accounts, services or hosts managed by directory services. As much as possible, information regarding installed software or services like IDS, IPS, database servers, certificate authority services, source code management servers, messaging services, monitoring or backup systems, among others, shall be collected and analysed concerning the new paths of attack they may reveal. Depending on the depth of the PT as well as on the rules of engagement, additional sensitive data may be collected with key-logging and screen capture software [39, 40].

At this step, the tester shall also try to leverage the possibilities uncovered with the new information. Therefore, data exfiltration possibilities shall be tested, as well as arbitrary code execution or even backdoor and rootkit installation as forms of acquiring persistence for future compromise. Additionally, evasion techniques that cover malicious actions like audit log manipulation shall be employed, as they will provide a perspective on the maturity of the implemented security controls [38, 39].

When the PT is considered finished, the analyst shall clean up all the systems with which he interacted, including user accounts created for testing, binaries, scripts, files and folders. Furthermore, all the changed configurations shall be restored to original values and any backdoor or rootkit must be removed. The collected information must be treated in the terms specified in the rules of engagement considering, as a base rule, that its storage shall be always encrypted [39].

2.2.4 Reporting

A PT becomes completed with a report delivery, as should be stated in the rules of engagement. Although the structure of the report may be adjusted, considering its purpose, it must describe the identified vulnerabilities, present a risk rating of each one and give guidance about mitigation of the discovered weaknesses, although PCI DSS, FedRamp and OSSTMM don't consider it mandatory. The results of a PT may have several goals and can be used as a reference for corrective actions, to access the implementations of security requirements, to meet compliance requirements, to measure an organization's progress towards meeting security requirements or to define mitigation activities based on a cost/benefit analysis, among others [16].

Usually, these reports begin with an executive summary stating the major findings and presenting an high level view of the PT evincing, also, that the vulnerabilities and

their severity should be taken as an input to the organization's risk management process. The executive summary presents an explanation of the overall purpose of the test, stating the targets considered in scope, the type of test performed and the testing goals. Additionally, it unveils the overall risk score, according to the scoring mechanism established at the pre-engagement phase, justifying the rationale followed to reach it. It may conclude with a recommendations summary that points to the steps needed to mitigate the identified risks and their respective priority with the associated level of effort. The intended audience of this section is the organization's management, therefore requiring a very concise summary of the test's findings that allows an informed decision about the steps to take [15, 41].

The main part of the report addresses the PT from a technical point of view, presents the testing narrative, detailing its scope, the selected targets, the methodologies used to perform the test, the testing limitations, the tools used, the timeline of the test and the vulnerabilities found. The extent of the organization's information gathered should be presented, specially those considered sensitive and publicly available. This information may reveal the maturity of the overall security posture, namely the human resources. For each finding a risk score has to be proposed, preferably according to Common Vulnerability Scoring System (CVSS) framework [21], a relationship with known vulnerabilities shall be established, based on CVE and CWE lists [22, 23], and a detailed description on how to reproduce it must be presented. Additionally, all the identified attack paths shall be reported, notably those that chain together some of the identified vulnerabilities, pointing out the level of access acquired, the compromised assets, the required level of skill to perform it and the required level of access to deploy the attack. Generically, the information to provide has to clear out the steps needed to mitigate the vulnerabilities found [15, 18, 20, 41].

There are some advantages to resort to standardised frameworks or lists like CVSS, CVE or CWE, once they provide a clear reference for communicating the characteristics and severity of software vulnerability. The use of these references, that shall be stated in the rules of engagement, make possible to have a common reference that is meaningful and easily understandable across industry to support a threat based analysis.

2.3 Web Vulnerabilities

Web applications may have a wide variety of vulnerabilities that can arise from factors like insecure coding practices or from the use of vulnerable third-party libraries, among others. To minimise the associated risks, OWASP has been developing the "Top Ten" project which, periodically, reviews the consensus based most critical web vulnerabilities in order to provide effective first steps towards secure coding practices. Although the 2020 version is being prepared, the actual one reports to 2017 and lists the following Top Ten application security risks: Injection, Broken Authentication, Sensitive data exposure, XML external entities (XXE), Broken access control, Security misconfiguration, Cross-site scripting (XXS), Insecure deserialization, Using components with known vulnerabilities and Insufficient logging and monitoring.

Although the context of this project is within a broader effort of enhancing the security of software, namely establishing a continuous application security testing throughout the entire software development life cycle, the referred vulnerabilities shall be tested during a PT.

2.3.1 Injection

Injection flaws occur when data provided by an user, or that can be tampered with, is not properly validated, filtered or sanitized by an application at server side, therefore allowing to relay malicious code through it to be executed at the backend. Although client side validation is a current practice, either for functional or security reasons, it can be easily bypassed, thus requiring another validation step at server side. This is a very prevalent vulnerability that can affect applications that use interpreters or that implement their functionalities resorting to operating system features or other external programs combined with user-supplied data, particularly in legacy code [34, 42].

In order to detect such vulnerabilities, all the user-supplied data should be tested to check if it is being validated, filtered or sanitized, used in non-parameterized queries, used in Object Relational Mapping (ORM) search parameters or directly used or concatenated in any type of commands. They are frequently found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Additionally, vectors associated to code injection, template injection or buffer overflows shall also be verified. Automated testing of all parameters, headers,

URL, cookies, JSON, SOAP, and XML data input points, associated with fuzzing techniques, are one of the ways to detect vulnerable injection points and, in the case of a white-box PT, source code analysis is an essential method to accomplish it [34].

To exploit these kind of vulnerabilities, the context where they occur must be analysed, in order to produce a manipulated input that, when interpreted at server side, a different action different from those intended may be executed. These kind of vulnerabilities may have as a consequence data disclosure, corruption or loss as well as remote code execution which, in certain cases, may enable a complete host takeover.

2.3.2 Broken Authentication

Authentication flaws emerge when protected content becomes accessible by other users than the intended ones. Since the use of the pair username and password is the most common authentication method and that, currently, there are a considerable amount of leaked credentials from past data breaches, this data can be used in the context of some attack methods to compromise some users access to an application. As such, there is an ongoing effort to implement multi-factor authentication and to adopt more secure practices regarding the use of this method of authentication. Another concern is related with session management implementation, namely in what is concerned with session or authentication tokens life cycle (generation, storage, validity) [34].

There are some indicators that could point to the presence of such vulnerabilities. Therefore, an analyst should verify issues like: transmission of credentials over an encrypted channel; use of default credentials; predictable default password generation; account lockout mechanisms; authentication schema bypass; vulnerable credential recovering; browser cache weaknesses; password policies; password change or reset functionalities; authentication over an alternative channel. Because some applications operate over different channels, sometimes vulnerabilities that are not present in one of them may show up in another, like in a mobile app or desktop application. Thus, even if those channels are out of the scope considered in the rules of engagement, they should be mentioned, because their presence broadens the attack surface, impacting the degree of assurance of the assessment [15].

To avoid continuous authentication, session management mechanisms provide the needed functionality to maintain an user's state towards the application. As such, it

should be tested considering: cookies that implement session management; cookies attributes; session fixation; exposed session variables; Cross-site Request Forgery (CSRF); session termination/logout; session timeout; session puzzling. Session tokens should be tested concerning their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage as well as their tamper resistance, structure and character set [15].

Resorting to breached data, an attacker has access to a considerable amount of username and password combinations that may be tried using a credential stuffing attack. Additionally, brute force or other automated attacks may also be tried, as well as default or weak credentials testing, particularly if lock out mechanisms are not implemented. In order to bypass authentication schemes, parameter modification, session token prediction or even SQL injection may be tried. Another testing vector is the usual credential recovery functionality or the "forgot password" process that, alone or combined with other kind of exploitation, may grant access to an user's account.

Session management mechanisms should also be tested, namely for predictable session or authentication tokens generation, tokens exposure in URL or lack of tokens invalidation after logout, idle or absolute session timeout [34]. Understanding cookies creation and management must be the first step to a successful attack. Thus, tampering with session tokens may provide a way to bypass authentication mechanism, allowing for a range of effects like impersonation or privilege escalation. CSRF may provide an indirect way of performing some malicious activities resorting to an authenticated user that, in an unnoticed way, will materialise the attacker's intents. In the case of successful attacks like cross-site scripting, there is the possibility to exfiltrate a session or authentication token from an user and, therefore, impersonate him towards the application, especially if session logout or timeout functionalities are not properly implemented.

2.3.3 Sensitive Data Exposure

Some targets deal with sensitive data like personal health records, credit card numbers or business information, among others, therefore being a potential target to attacks. These kind of data should be protected either when at rest or in transit with the use of cryptography, as established in legislation like GDPR or other standards like PCI-DSS. However, there are still applications that don't implement such security controls or that do it in an improper way, allowing attackers to access sensitive data.

Therefore, verifying the traffic established with the application is important to reveal if the data is being transmitted in clear text or not. When the protocol in use is HTTPS, a downgrade to HTTP should be attempted, since some applications don't enforce the use of TLS. If cryptography is employed, the algorithms in use should be verified as well as the protocols and the key length. Additionally, the existence of accessible cryptographic keys should be verified, as it could be the only way to circumvent this security control. Another important detail is to verify if the certificates are being validated [34]. When performing a white-box PT, code and configurations review is the fastest way to identify these vulnerabilities.

When data transit or storage is not encrypted, accessing it is trivial. In the case where cryptography is employed, some approaches may be tried, like using stolen keys, attack known vulnerabilities of some algorithms or protocols, test for the use of weak or reused keys and even force the use of weaker ciphers or, in a worst case scenario, force the use of no encryption. When there are other available services in separate tcp port, their eventual vulnerabilities may provide an entry point to the target or, some times, could provide access to private keys, allowing future decryption of transmitted data [15]. Possible attacks against TLS include Heartbleed, BEAST, CRIME, TIME, BREACH, STARTTLS or Padding Oracle attacks, among others, as well as those that explore faulty or insecure implementations [15, 43].

2.3.4 XML External Entities (XXE)

Some applications use XML format to exchange data between the browser and the server, allowing the abuse of some of its features if the XML parser is weakly configured. This format is very flexible and its structure, type of data values and other items may be defined through the XML document type definition (DTD). The DTD contains or points to markup declarations like element type, attribute-list, entity or notation declarations and is declared within the optional *DOCTYPE* element at the start of a XML document. XML entities are one of the ways to represent data inside a XML document which, due to the flexible nature of XML, may be customised. External entities are one of such custom entities and are declared with the *SYSTEM* keyword, specifying an URL from which the entity's value is retrieved from. When the the URL is accessed by the parser to process the entity, it replaces its occurrences with the contents of the resolved URL, allowing for the manipulation of the behaviour of the application. Additionally, the *file://* URL schema,

and other protocols such as *gopher://* may also be used instead of *http://*, providing several attack paths ranging from accessing to private network resources, as the processing occurs at server side, to injecting payloads through attacker controlled resources, once the URL may point to an attacker controlled resource [44–46].

Applications accepting XML data through requests or file uploads may be vulnerable to this kind of attack. Defining an external entity pointing to a system file may reveal its contents in the server response. However, when the results aren't returned with the response, XXE may be verified using out-of-band techniques or through error messages that leak sensitive data.

Therefore, XXE may be exploited to retrieve file content when XML data is modified in order to define an external entity pointing to a file from the server's file system that is, then, returned in the application response. As an example, supposing that an application checks for the existence of a product submitting a similar XML payload to the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<productCheck><productId>1337</productId></productCheck>
```

If the application has no protections in place, an attacker could manipulate the request submitting a modified version in order to retrieve the */etc/passwd* file, sending the following XML payload:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<productCheck><productId>&xxe;</productId></productCheck>
```

The defined external entity &xxe; is resolved by the XML parser, including its content inside the *productID* tag, and returning the file content in the application's response.

Other attacks are also possible, like server-side request forgery, when the external entity is defined with an URL pointing to a back-end resource, as well as data exfiltration through an out-of-band channel, sending information from the application back-end to an attacker controlled server, or data retrieving through error messages or even a denial-of-service attack like *Billion laughs attack*[1] [34, 45].

[1] https://en.wikipedia.org/wiki/Billion_laughs_attack

2.3.5 Broken Access Control

Access control mechanisms try to enforce content and functionality access based on the authorization that an user may have to perform it and, in the context of a web application, it is dependent on authentication and session management mechanisms. When not properly in place, there is the opportunity for an attacker to have an improper access to data or functionality, giving place to perform privileged actions or read reserved information. Authorization may be implemented following one of two main approaches: Role Base Access Control (RBAC) and Access Control Lists (ACL). Furthermore, some web applications build their access control mechanisms upon the concept of Model-View-Controller (MVC) which allows for some flexibility, concerning the placement of security controls. However, sometimes those controls are placed at the "view" layer, behaving like filters to the content that must be rendered. Thus, the authorization check may happen after some previous actions have already taken effect, like changing database contents, although not visible to the user [47–49].

Access control flaws are, mostly, detected using manual means as they manifest themselves at a functional level, depending on the application business logic. The assessment for these vulnerabilities shall consider access control bypass via modifying requests, namely detecting insecure direct object references (IDOR), path traversal possibilities, insecure access control tokens or cookies and, additionally, testing for the authorization schema bypass with forced browsing or for privilege escalation possibilities. This kind of testing requires a variety of user profiles and testing accounts and should be quite extensive in order to test all the application functionalities and its robustness concerning proper authorization [34].

To exploit this kind of vulnerabilities, the main vector is request manipulation, either on the URL or on its data section or headers, including also cookies that may be stored on the browser. In general, if the user's access rights or role information is placed at location controllable by him, an attacker may tamper with that data. Detecting tamperable parameters, thus, could be quite effective in enabling an attacker to access restricted data or functionalities. To illustrate a parameter-based access control method bypass, let's consider the following URL to access an application:

```
https://nosecurity.com/login/home.jsp?admin=false
```

If there is no other mechanism to enforce authorization besides the *admin* parameter, an attacker could grant access to administrator functionalities if the requests are tampered with the following version:

```
https://nosecurity.com/login/home.jsp?admin=true
```

Other vectors of attack to consider are the identification of unprotected functionalities, IDOR vulnerabilities or faulty multi-step authorization processes that fail to implement access controls in every step, allowing to skip the protected ones and accepting request submissions with the required parameters at the unprotected steps [34, 47].

2.3.6 Security Misconfiguration

When deploying web applications in a production environment, there should be an awareness that what is really being exposed is an array of interconnected elements like hardware, applications, third party libraries and their related security issues. As such, all these elements need to be mapped and reviewed in order to get some degree of assurance about their security. Security misconfigurations may occur at any of those elements, including the network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage services. Additionally, the administrative tools used to manage all the referred elements should also be considered as an attack vector [15, 34].

During the discovery and analysis activities, an appropriate mapping of the technologies and elements composing the web application should have been accomplished which, resorting to vulnerabilities databases information, shall help to outline an initial testing approach for each component of the application stack. The use of automated tools can be helpful in detecting misconfigurations like the use of default accounts, credentials or configurations, the presence of unnecessary services or the use of insecure options. Therefore, an understanding of how different elements of the application interact is quite determinant to a comprehensive assessment. Furthermore, some details may denote the presence of this type of vulnerabilities, like error messages disclosing excessive information or absence of some security headers[34].

Exploiting such vulnerabilities may have several approaches, including taking advantage of default configurations and credentials usage or abusing discrepancies in how elements in the same data flow parse HTTP requests, among others. An example of the last possibility is an HTTP Request Smuggling attack which exploits the discrepancies in

parsing HTTP requests when one or more HTTP "*devices/entities (e.g. cache server, proxy server, web application firewall, etc.) are in the data flow between the user and the web server*" [50]. This attack relies on sending specially-crafted HTTP requests that are parsed differently by the involved devices, allowing for several kinds of additional attacks like web cache poisoning, session hijacking, cross-site scripting and most importantly, the ability to bypass web application firewall protection.

2.3.7 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is a vulnerability usually found in web applications where a malicious agent successfully injects code, generally a browser script, that is executed when an unsuspecting user accesses some web resource. XSS allows an attacker to perform actions reserved to the legitimate user, since the malicious script is executed in the context of the latter's session. It has three types, Stored XSS (persistent), Reflected XSS (non-persistent) and DOM Based XSS, related to the origin of the malicious scripts: storage system, http request and client-side code, respectively. Generally, XSS may allow an attacker to impersonate the victim user, perform actions that the user is able to do, read any data accessible to the user, capture user's credentials, perform virtual defacement of the web site or inject trojan functionalities into the web site. When present, XSS makes CSRF protections useless, unfolding worrisome paths of attack [34, 51].

XSS is a type of injection vulnerability, thus occurring when user supplied data, not appropriately sanitised or escaped, succeeds in executing JavaScript and HTML in the victim's browser. This vulnerability is detected in a similar way to other injection vulnerabilities, probing every data entry points, and can resort to automated tools to accomplish it. The application responses are, therefore, analysed, searching for the test input in the response and verifying if they have been, or not, sanitised, encoded or replaced. When inputs are sanitised, bypassing XSS filters should be tried, once these mitigations could be improperly implemented [15, 34].

Depending on the type of XSS present, the attacks are deployed in different ways. Considering an application with a blog service where users post messages they could be displayed to other users in the following way:

```
<p>Hi! This is a secure message.</p>
```

However, an attacker may submit the same message accompanied with a script that executes JavaScript code in an unsuspected way:

```
<p>Hi! This is a secure message.<script>document.location='https://
www.attackersite.com/?data='+document.cookie</script></p>
```

In this example, the user's session cookie is exfiltrated to the attacker's server and can be used to enter his running session at the application, bypassing the authentication step. This method could also be leveraged to perform other actions on behalf of the legitimate user, like changing his own data within the application context, bypassing eventual CSRF protections or even accomplishing full account takeover [15, 34].

2.3.8 Insecure Deserialization

Data structures may be transformed in a byte stream representation that retains its properties, attributes and assigned values, providing a simpler way to store it in an inter-process memory, a file or a database, as well as exchange it over a network between different elements of an application. This transformation process is known as serialization and may be reverted, in order to restore the byte stream to a replica of the original data structure, maintaining the aforementioned characteristics. Deserialization may be insecure when it is operated over user-controllable data, since an attacker can manipulate or replace a serialized object, independently of the process to accomplish it, that may force the application to execute unintended code at deserialization time, allowing for remote code execution in the worst case scenario. As such, deserialization of user-controlled data should be avoided, even if additional verification is done, given that anticipating all verification needs does not seem feasible in order to account for every tampering possibility. One of the main problems lies in the fact that some of the attacks are, actually, executed before the deserialization process is finished, making verification useless, once it happens after the end of this process [34, 52, 53].

Insecure deserialization may be present in languages like C, C++, Java, Python, PHP, Ruby and, probably, others. Therefore, knowing how objects are serialized in each language is determinant to detect it. Some use string formats, with several degrees of human readability, while others employ binary formats. The latter, may give a false sense of security, however, despite they could require more effort, the main problem persists. As an example, PHP resorts to a string format where letters represent the data type and numbers indicate each entry length. Considering a *Client* object with the following attributes:

```
$client->name = "Bob";
$client->isAdmin = false;
```

When serialized, it will be represented:

```
O:6:"Client":2:{s:4:"name":s:3:"Bob"; s:7:"isAdmin":b:0;}
```

As such, if some of the data being exchanged between the browser and the server presents a similar structure, there should exist serialized objects, with a high degree of probability. Additionally, when performing a white-box PT, some methods should be detected, like *serialize()* and *unserialize()* in the context of PHP language [52, 54].

Deserialization vulnerabilities may be exploited in various ways like modifying object attributes or data types, abusing applications functionalities, abusing constructor methods, injecting arbitrary objects or using gadget chains. As an example, if we consider the object above as an user's session data stored in a cookie, an attacker intending to escalate his privileges, could try to change the attribute *isAdmin* to 1 (true). Therefore, he would have to re-encode the object and overwrite the cookie with the new data:

```
O:6:"Client":2:{s:4:"name":s:3:"Bob"; s:7:"isAdmin":b:1;}
```

If the privileges are checked considering the cookie content, as in listing 2.1, the *Client* object is deserialized and instantiated with the *isAdmin* attribute set to true, granting access to the admin interface, if the authenticity of the serialized object is not verified. However, if such verification was implemented, another kind of exploit could be considered, like abusing constructor methods or injecting arbitrary objects [34, 54].

```
$client = unserialize($_COOKIE);
if ($client->isAdmin === true) {
// allow access to admin interface
}
```

LISTING 2.1: Check cookie

2.3.9 Using Components with Known Vulnerabilities

Software development, with a high degree of probability, resorts to code re-use, either open source, libraries or frameworks. Parallel to that, complex applications need to integrate diverse technologies like web servers, application servers, load balancers, database management systems, virtualization environments, other applications, APIs, with all of these elements running on top of operating systems. Once any of these components may

present their own vulnerabilities, its determinant for the global security of an application that each of the elements solely integrates it when is properly patched and has no conflicting configurations. Considering that, frequently, any of the components runs with elevated privileges, when one of them is exploitable, a full compromise of the system becomes easier [15, 34].

During the discovery and analysis activities, the application components have been identified thus allowing to outline an initial testing approach for each one. Important details to consider are software versions which permit the comparison with vulnerabilities databases information. Also, the presence of unsupported, not upgraded or unpatched software increases the probability of vulnerabilities existence. Incompatible elements also present risks, as the referred HTTP Request Smuggling attack illustrates (see section 2.3.6).

Therefore, exploiting an application with components with known vulnerabilities could mean to build an attack path that leverages each of those vulnerabilities in order to compromise a system. The approach needs to be as varied as the technologies employed are.

2.3.10 Insufficient Logging and Monitoring

From a security perspective, logging and monitoring applications operation are important controls operating on two levels: register relevant events for future analysis; checking if the actions triggered are within the expected application usage. Usually are implemented in a related way, where the monitoring tools act based on the logs collected. As such, monitoring allows protection mechanisms to be triggered when, for example, an attacker is trying to brute force a login form, while logging allows for a posterior analysis to determine the attack provenance and the actions tried to perform it. Logging should register security relevant events, including successful and failed authentication events, access control failures, deserialization failures and input validation failures, providing relevant information for a future forensic analysis, including the reconstruction of an attack timeline. Thus, logs should be clear and easily monitored and analysed, and should follow storage and protection good practices [15, 34].

While establishing if logging and monitoring controls are implemented could be difficult, there are some indicators that may evince that they may not be properly in place, at least. One of such contexts is when rate limiting is not present, allowing for brute force attacks. Assuming that most successful attacks begin with reconnaissance and vulnerability testing, and that the latter implies interaction with the application, allowing such

testing without any kind of limitation will increase the likelihood of a successful exploitation. In a white-box PT, checking for the existence and correct implementation of such controls is simpler. However, later analysis of a Penetration Testing (PT) should include attacks reconstitution based on logs information, to determine if logging mechanisms are properly set [15, 34].

Chapter 3

Supporting Background Work

In this chapter we will approach central elements that give support to the implementation of OrchRecon. Distributed systems pose a set of challenges, namely those regarding communication between several components, as well as others related to its design as far as transparency, scalability or heterogeneity are concerned. As such, some of the related core concepts were reviewed in order to support our architecture choices.

Additionally, we reviewed some of the open source tools to perform reconnaissance in the context of a Penetration Testing that implement some level of automation, with a special focus on those that better relate with our solution.

Once containerization was the chosen approach to our module implementation, we also present a high level analysis of this technology, as it provides fully configured environments to each tool without interfering with others' dependencies.

3.1 Containerization

Containerization is an Operating System (OS) virtualization technique that allows for process isolation and is particularly adequate for application management. It builds from the concepts of *namespaces* and *cgroups*, kernel mechanisms in Linux distributions used to isolate processes on a shared OS, supported by the Linux kernel containment features, commonly known as LXC. *Namespaces* grant process isolation, preventing groups of processes from seeing other groups resources like network interfaces, inter-process communication and mount points. Additionally, *cgroups* enable limitation and management of process groups, considering hardware resources like memory or CPU utilisation, allowing containers to share them [55].

Once containers share the same OS, they provide a smaller and lightweight form of virtualization when compared with others like hypervisors, becoming a popular solution for delivering self-contained applications including, when necessary, middleware and business logic implementations. Furthermore, containerization encapsulates the environment where applications are developed, bundling their code with other configuration files, libraries and dependencies it may require, allowing for faster deployments with fewer errors or bugs, at the same time as it provides consistent environments, isolation and portability [55, 56].

Docker, the most popular container solution, extends LXC with a kernel- and application-level API and provides a runtime engine which manages OS resources for containers. A Docker container is an isolated, lightweight, executable package of software that includes all the resources required for an application execution: code, runtime, libraries and settings. They are built with layers of individual images on top of a base image, containing all the required resources, ranging from OS fundamentals to a complex pre-built application stack. This building process is implemented with a script, a Dockerfile, consisting of various instructions that, incrementally, modify the base image [57, 58].

3.2 Distributed systems

Distributed systems give support to an increasing myriad of daily activities like web search, e-mail, financial trading systems, mobile communications, multiplayer online gaming or GPS location as well as corporate or industrial networks, just to name a few examples. One of the prime reasons to build a distributed system is resource sharing, here understood in a very broad sense, ranging from hardware components to software-defined entities like databases, files or other kinds of data that are network connected [59]. Over the years, several definitions have been established to characterise it of which that of Steen et al. [60] presents it in a very broad perspective: *"A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system"*. This perspective is compatible with others like that from Coulouris et al. [59] and identifies the same kind of challenges that should be considered when developing such systems.

Considering computing elements as hardware devices or software processes, their independent operation is, however, tied to a common objective from the system's perspective. Therefore, some questions arise concerning their synchronisation and coordination,

how they communicate, how their belonging to the system is managed, namely concerning membership, registration and egress, and how these states are communicated to other elements [60].

An apparently single coherent system, according to the author, is one that behaves according to users expectations and hides the collection of computing elements that constitute it. These characteristics present some other challenges concerning systems design, such as distribution transparency, heterogeneity or failure recovery, as they are composed of multiple networked elements, sometimes with different OS, and its eventual failure in time that cannot be avoided.

To deal with heterogeneity due to OS, hardware or programming languages diversity, distributed systems are, usually, developed on top of middleware. It presents a uniform computational model for developers and is implemented on the application layer, providing a programming abstraction to manage the underlying heterogeneity in a networked environment. Middleware manages resources across the network and provides services like inter-application communication, security, accounting or masking of and recovery from failures [59, 60].

3.2.1 Design issues

Developing a distributed system should be weighed in order to check if it is a justified option. Steen et al. consider that such a system (i) should assure that resources are easily accessible, (ii) should hide the fact that resources are distributed across a network, (iii) should be open and (iv) should be scalable. Beyond these goals, Coulouris et al. refer some additional challenges that should also be considered: heterogeneity, security, failure handling, concurrency and quality of service.

As already referred, resource sharing is one of the main purposes for building a distributed system, whether they are facilities, storage, peripherals or data, files, applications, networks, among others. Economic reasons are some of the driving forces to such an option which, also, facilitates collaborative work, teleconferencing or information exchange. Peer-to-peer networks are an example of a distributed system built for resource sharing, making file sharing a very easy task [60].

Distributed systems feature of hiding resource distribution over a network is referred as *Transparency* by the literature, so that a system looks to its users and applications as

a single coherent system rather than a set of independent elements, sometimes physically separated across multiple machines over large distances. According to the Reference Model of Open Distributed Processing (RM-ODP), there are eight types of distribution transparency, as depicted in Table 3.1 [61].

Transparency Type	Description
Access Transparency	Masks differences in data representation and invocation mechanisms to enable interworking between objects
Failure transparency	Masks from an object the failure and possible recovery of other objects (or itself) to enable fault tolerance
Location transparency	Masks the use of information about location in space when identifying and binding to interfaces
Migration transparency	Masks from an object the ability of a system to change the location of that object
Relocation transparency	Masks relocation of an interface from other interfaces bound to it
Replication transparency	Masks the use of a group of mutually behaviourally compatible objects to support an interface
Persistence transparency	Masks from an object the deactivation and reactivation of other objects (or itself)
Transaction transparency	Masks coordination of activities amongst a configuration of objects to achieve consistency

TABLE 3.1: Transparency Types

Scaling a distributed system is a central concern for developers and has become one of the most important design goals. Steen et al. refer three dimensions to measure scalability:

- Size scalability
- Geographical scalability
- Administrative scalability

From the size perspective, a system can be scalable if there isn't a noticeable performance degradation when more users or other resources are added to it. Concerning geography, a system is scalable if the distance between its components or users, with the associated communication delays, is hardly observable. Analysing from an administrative point of view, a scalable system is one that keeps an easy management despite it depends from several independent administrative organizations. In this regard, one of the major problems arises when there are conflicting policies concerning resource utilisation, management and security [60].

Besides the referred design issues, Coulouris et al. pointed some additional challenges concerning the implementation of distributed systems. *Heterogeneity*, here understood as

the variety and difference applied to networks, hardware, OS, progrmaming languages or implementations by different developers, should be masked in a distributed system. As such, there is the need to resort to a programming abstraction provided by a middleware. Middleware provides an uniform computational model, operating at the software layer, that deals with existing differences in hardware, OS or networks. These models may include remote object invocation, remote SQL or distributed transaction processing, among others. Another solution to handle heterogeneity is through virtualization, either in the form of process virtual machines, like Java virtual machine, or through the migration of a collection of processes, including the underlying OS [60].

3.2.2 Types of distributed systems

Distributed systems can be quite different from each other, according to the services they provide. From the distinction proposed by Steen et al. between distributed computing systems, distributed information systems and pervasive systems, and considering several types within each group, we will refer to the first two.

Distributed computing systems

Concerning distributed computing systems, the authors subdivide this group in two types, according to the heterogeneity of the underlying infrastructure: cluster computing and grid computing. Cluster computing is an approach in which a single compute-intensive program runs in parallel on a set of interconnected machines that cooperate to deliver a single high-performance computing capability. Its infrastructure is based on a set of similar machines running the same OS.

On the other hand, when the system is constructed as a federation of computer systems, with different hardware, software, networks, programming languages and administrative domains, we are towards grid computing type. Grid computing enables the sharing of such resources in order to address intensive computational tasks, resorting to a middleware that provides resources from different administrative domains to users and applications related to specific virtual organizations.

The evolution of the referred distributed computing system types led to a utility based approach known as cloud computing. It is characterized by a dynamic and scalable utilisation of a pool of accessible virtualized resources, often paid on a per-utilisation basis,

that enable outsourcing the entire infrastructure for compute-intensive services in a dynamic way [59, 60, 62].

Distributed information systems

Networked applications typify another class of distributed systems of which a server side application, including a database, providing resources to client applications is an example. This architecture presents challenges like how to implement communication between application components and how to assure that the application stays in a consistent state.

Typically, clients send requests in order to perform an operation, or a group of operations, at server side, sometimes involving different servers. As an example, operations on a database materialise through transactions which require a set of primitives supplied either by the underlying distributed system or by the language runtime system. The available primitives depend on the kind of objects involved in the transactions of which READ and WRITE are frequent examples. Furthermore, beyond the use of primitives, transactions may also include Remote Procedure Calls (RPC). Transactions have a characteristic property, either all of the involved operations are executed successfully or none is executed, which together with three others constitute the *ACID* properties:

- Atomicity: a transaction either entirely succeeds or all the involved operations fail;
- Consistency: a transaction changes a system from a consistent state to other consistent state;
- Isolation: concurrent transactions do not interfere with each other;
- Durability: a committed transaction performs a persistent change.

3.2.3 Communication in distributed systems

Communication between processes is a core subject for distributed systems, for which some models where developed beyond the usual request-reply protocol. Two of the most widely-used models for providing such resources are Remote Procedure Calls (RPC) and Message-Oriented Middleware (MOM). While the first model is adequate for client-server architectures, the latter is most suitable for applications that do not follow that kind of interaction [60].

Remote Procedure Calls

The concept of RPC, attributed to Birrel and Nelson [63], proposes a mechanism to execute procedures in remote machines as if they were available locally, offering access and location transparency while hiding details like encoding and decoding of parameters and results, message passing, calling semantics or distribution. Generally, it is implemented over a request-reply protocol associated with an invocation semantics. As such, some design issues arise: programming with interfaces, calling semantics and transparency [59].

Interfaces establish explicitly how interactions between different components of an application can exchange data, namely specifying the procedures and variables that can be remotely accessed, the requested parameters and the output results. Since there is a clear separation between an interface and its implementation, they present some benefits: programmers only have to deal with the offered abstraction, they do not need to know the language or the platform in which it is implemented and there is room for software evolution as far as any change adheres to the interface specification or, in the case of a change in the interface, there is a backward compatibility. Interfaces are often specified through an Interface Definition Language (IDL) which allow procedures implemented in different languages to interact with each other. An IDL provides the means for defining an interface considering the possible operations, their parameters and outputs and the respective data types. Subsequently, an interface thus defined is compiled into a client stub and server stub, building the appropriate run-time interface [59, 60]. An example of an IDL use is Google Protocol Buffers, which provides a language-neutral mechanism for serializing structured data and a compiler to generate source code that is invoked to deal with the respective data structures [64].

Concerning the reliability of the remote interactions from the perspective of the client, RPC provide a set of invocation semantics intended to deal with possible failures, namely server crashes. Therefore, some types of semantics are considered: *maybe*, *at-least-once* and *at-most-once* semantics. The first type arises when there are no fault-tolerance mechanisms implemented and failures like lost of requests or replies or even remote server crashes are not addressed. *At-least-once* semantics is an approach that guarantees that a RPC was performed at least one time, but eventually more, returning to the client either a result or an exception. In the last approach, *at-most-once* semantics, the client receives either a reply or an exception, but has the guarantee that the RPC was performed not more than one time [59].

Since one of the premises of RPC is to look like a local procedure call, its implementation shall hide a set of actions like data marshalling and unmarshalling, message re-transmission or remote process location, providing a transparent approach from the perspective of a programmer. However, some issues may have to be addressed, like network latency or differences in parameter passing, for which some authors argue that the transparency level should not completely hide such details, so they can be handled appropriately [59].

Message-Oriented Middleware

There are circumstances where the use of RPC is not the most appropriate approach for communication within a distributed system, namely when both connecting parties are not guaranteed to be executing at the time of requests or replies. In these contexts, a message-oriented communication is more adequate, as it allows for an asynchronous communication approach [60].

The Message-Oriented Middleware (MOM) programming model is based on the concept of *queues*, through which processes can send or receive messages from other processes, providing support for persistent asynchronous communication, once it offers an intermediate-term storage capacity for messages. Assuming that each application has its own associated queue, the communication is performed by inserting messages in the appropriate queue which will then be forwarded to the destination, usually implementing a *first-in-first-out* (FIFO) policy. The MOM guarantees that the message will be placed in the recipient's queue but does not guarantees either when or if it will be read by the recipient [59, 60].

A message consists of a *destination*, *metadata* and the *body* of which the latter remains unchanged by the queuing mechanism and is, normally, serialized. The queue systems provide message persistence, storing messages indefinitely until they are read at the destination, guaranteeing that they are delivered once, only. The message receiving process may happen in three different ways: i) it blocks until an awaited message is received; ii) it performs a poll operation, checking the queue status and returning a message when available, in a non-blocking operation; iii) it issues a notification when a message is in the respective queue [59].

Message queuing systems may have different kinds of implementation, either centralised approaches or distributed ones. A centralised approach brings simplicity to the architecture of the messaging system but at the cost of being a single point of failure and,

in the case of heavy workloads, can become a bottleneck. As such, more distributed approaches have been developed, allowing for different topologies that can match requirements such as scalability and performance.

Many messaging applications may be organised according to a few communication patterns like *request-reply*, *pipeline* or *publish-subscribe* that provide the resources to enable various styles of communication like *one-to-many*, *many-to-one* or *one-to-one*. On a programming level, several approaches have been made in order to provide such communication abstractions while dealing with connections management. One of those efforts is provided by ZeroMQ, a high-performance asynchronous messaging library that supports the common messaging patterns over a variety of transports, like TCP, inter-process, multicast, WebSocket among aothers. It provides a higher level of abstraction in socket-based communication by pairing sending and receiving sockets according to their specific type [60, 65].

3.3 Reconnaissance automation

Reconnaissance automation is a permanent quest for security analysis. It ranges from single bash scripts to more sophisticated tools that try to automate repetitive tasks, bringing the benefits of speed, accuracy and wider coverage. The concept of automation, as previously referred, is present in regulations and standards as a base approach to vulnerability discovery and, although it does not provide error proof results, its benefits supersede some of the limitations. In the context of OSINT gathering, automation is helpful as it can produce a wide amount of data that, subsequently, must be narrowed to conform with the existing rules of engagement. This breadth approach is common in many tools for what there is also the need for an human factor to bring an appropriate insight.

There is an ongoing yearly installment about tools and techniques for bug hunters and red teamers concerning, namely, OSINT gathering. This year, at DEFCON 28, Jason Haddix presented an update to his *The Bug Hunter's Methodology* where he makes an in depth analysis of the reconnaissance methodology of a target and approaches several kinds of tools [66, 67].

He proposes a segmentation of the tools in four tiers:

- C-Tier - automation built with scripting up other tools in bash or python, applying a short number of techniques and without a proper workflow and extensibility.

- B-Tier - automation built with writing some of their own modules, applying a medium number of techniques in the context of a workflow
- A-Tier - automation built with writing almost all of their own modules, with an iterative workflow and data management through a database
- S-Tier - automation built with writing all their own modules, with an fast iterative workflow and data management through database, scaling across multiple instances and using novel techniques and features.

Despite of the distinctions made, sometimes we find tools with characteristics of another tier. However, it provides an updated set of automation approaches that are actively maintained. As such, we analysed some of the A and S-Tier tools in order to understand their architecture, resorting to public repositories, when available.

Findomain (A-Tier)

This tool, written in Rust, is aimed at monitoring target domains to issue alerts when new subdomains are found. It resorts to tools like Nmap, Amass, Sublist3r, Assetfinder and Subfinder to keep an updated list of new exposed resources, gathering data like new discovered subdomains, host IP, HTTP status, screenshots of the HTTP websites, open ports and subdomains CNAME, among others. However, only some of the payed version provides all the features, like continuous monitoring, screenshots or Nmap port checking as well as data storage, while the free version basicaly provides subdomain enumeration [68].

Rock-ON (A-Tier)

Rock-ON is a reconnaissance tool that automates a simple workflow in Ruby. Its features include subdomain scraping, ASN, IP, Ports and virtual host enumeration among others. To implement that, it uses tools like Sublist3r, Knock, Subfinder, Amass, AltDns, Nmap or Vhost amid others. Essentialy, it resorts to a set of scripts that, explicitly, run each of the embedded tools [69]. As such, it does not match the referred database criteria, common to both top tiers, although it has an iterative workflow concerning subdomain enumeration.

recon-pipeline (A-Tier)

This reconnaissance tool, implemented in Python, approaches the concept of pipeline

in order to scan a target. As the other examples, this one also resorts to Amass, Gobuster, Massscan or Waybackurls, among others. It provides the possibility to run a single tool but also presents wrappers around multiple commands to perform complete scans through its own shell. The gathered data is stored in a SQLite database and can be queried in various forms. It also offers the option of pipe each output to other commands, extending its use capabilities. Besides the command line interface it also provides a graphical dashboard to visualise the scan results. Contrary to the previous applications, this one has the possibility to add more tools, called scanners, as well as create more wrappers in order to adapt the application to the user intents [70].

Intrigue Core (S-Tier)

As advertised by the owner company, Intrigue Core is an open framework for discovering and enumerating the attack surface of organizations. It has an open source version, written in Ruby and JavaScript, that implements an engine to perform about 150 tests that enable the creation of a graph representation of the attack surface. It detects exposed database and TCP/UDP services, as well as vulnerable application stacks or obsolete libraries. It can run individual tasks or in a fully automated mode to routinely collect information about the organizations' attack surface. It emulates the actions of an intelligent actor to interactively map internet-facing systems, exposed services, and applications [71, 72].

SpiderFoot (S-Tier)

SpiderFoot is an automation tool focused in OSINT gathering. Written in Python, it integrates with public data sources to extract information like IP addresses, domains, subdomains, hostnames, subnets, ASN, e-mail addresses, phone numbers, person's names and usernames as well as data leaks. With the collected data, it performs some analysis, establishing correlations among the retrieved information. The user can select which modules to run either through the GUI or through the command line interface. Additionally, it gives the user the possibility to write his own modules and integrate them in the application [73, 74].

Osmedeus (S-Tier)

Osmedeus is an automated framework for reconnaissance and vulnerability scanning written in Python. As other frameworks, it resorts to tools like Subfinder, Httprobe,

Gospider, Gowitness, Aquatone among many others to perform subdomain enumeration, take screenshots, perform port scanning or check for vulnerabilities as well as other tasks. The user may run specific modules and also choose how fast the scans should be performed. It may be operated either through a GUI or through the command line interface and produces reports with the collected findings [75].

The table 3.2 summarises the characteristics of the analysed tools. As a conclusion, we note that there is an emphasis in resorting to already known tools and in their combination, in order to widen the findings coverage. It is a common approach to have a pre-set workflow and, in most cases, the tool selection is fixed, limiting the user's choice to the existent native modules.

Name	Module's origin	Running Mode	GUI	Database	Tier
Findomain	Third-party	Pre-set Workflow	Yes	Yes (paid version)	A
Rock-ON	Third-party	Pre-set Workflow	No	No	A
recon-pipeline	Third-party	Individual tool / Pre-set Workflow	No	Yes	A
Intrigue Core	Third-party / Proprietary	Individual tool / Pre-set Workflow	Yes	Yes	S
SpiderFoot	Proprietary	Tool selection	Yes	Yes	S
Osmedeus	Third-party	Tool selection	Yes	Yes	S

TABLE 3.2: Reconnaissance Automation Tools

Chapter 4

OrchRecon

As previously referred in 2.2.2, the main purpose of the reconnaissance process is to map a target's presence in the cyberworld as comprehensively as possible. Additionally, OSINT gathering uncovers other relevant information that can enhance the effectiveness of the attack phase. Therefore, some scenarios may illustrate a few of its benefits in the context of a PT.

Assuming that social engineering attacks are within the scope of a given PT, knowing details about a company employees, such as their hobbies, social network publications, blog postings, relationships, among others, may favour a more successful approach. As such, generic phishing attacks may evolve into spear phishing ones, which tend to be more difficult to detect and, potentially, more effective [76].

OSINT about a specific target can prove to be quite fruitful. Querying a search engine resorting to search operators may uncover files with usernames, passwords or other sensitive information that may become useful later. Therefore, during the attack phase, the researcher will have valuable information in scenarios where he has to bypass authentication procedures.

Regarding the exposed services that a target has, some of them may use unpatched versions of a software with known vulnerabilities. Consequently, searching that kind of information may save time and turn the attack phase more effective, considering that some of the existing faults may already have an associated exploit.

In the context of application development, investigating a company's code repository or its employees comments on some public forums, may give some insight about software architecture or related details. This kind of research may prove highly valuable as

it uncovers things like hardcoded credentials, private api keys or certificate private keys, among other kinds of information, that may have been inadvertently left in the code.

From the examples above, reconnaissance worths a special attention in the context of a PT as it may empower the researcher with intelligence that can be leveraged in the attack phase. Therefore, and taking into account the benefits of orchestration [77], we approach this phase of a PT with the help of OrchRecon.

OrchRecon elaborates from the concept of reconnaissance automation based on a customised workflow. As other frameworks, it resorts to existing tools to perform specific tasks, but giving the user the flexibility to choose those it considers more suitable and building upon a distributed system approach in order to increase the overall performance when compared with a typical workflow. OrchRecon is intended to be used in the context of a Penetration Testing (PT) or a *Bug Bounty* program for Open Source Intelligence (OSINT) gathering. However, depending on the chosen tools, it may be leveraged to integrate an attack path.

Considering the literature that refers to specific tools to be used during a PT, it is noticeable that some of them are not maintained nowadays. Additionally, although some tools are intended to perform some specific task, there is some user discretion that relates to its effectiveness, usability or performance, for example. Furthermore, it is usual to combine the output of more than one tool, in order to increase the information gathering coverage. As Kritikos et al. pointed out, tools should be configured according to the context and, if orchestrated with others, there is an increase of the overall coverage level [77]. As such, OrchRecon presents a message-oriented middleware for a distributed system and provides an open and customisable framework that allows for some degree of paralleling, depending on the available computing resources. It enables a human-guided PT with a high degree of automation amid the chosen tools.

This tool may be used either locally or in a cloud environment which, in the context of this project, should be considered as Google Cloud Computing Services resources [78].

4.1 Architecture overview

OrchRecon is built as a distributed system that allows to chain a set of tools to perform a desired set of tasks. It provides a message-oriented middleware to orchestrate the use of the chosen tools and, to accomplish that, the user must guarantee that the output from

some tool can be used as input to another, establishing a pipeline of tasks. It is composed by six main components: Master, Broker, Pipeline Manager, Daemon, Database and Modules, as illustrated in figure 4.1.

The Master component is responsible to receive tasks constituted by the target domain to be tested and the pipeline of tools to chain. Afterwards, each task is dispatched to the Broker component which is responsible for the communication between the Master and the Pipeline Managers, queuing up the tasks if none of the managers is available. Therefore, an available Pipeline Manager will receive a task and will manage the pipeline execution, launching each Module. It also keeps the Database updated about the execution of the pipeline and manages file storage related to the tools output. In each running instance there is a Daemon that connects to the Broker and sends information about resources usage. It starts up new Pipeline Managers upon request from the Broker, according to the metrics sent. A Module is an abstraction for a specific tool. The user just has to ensure that it is configured to receive the desired input and that it produces an adequate output to the tool with which it is linked.

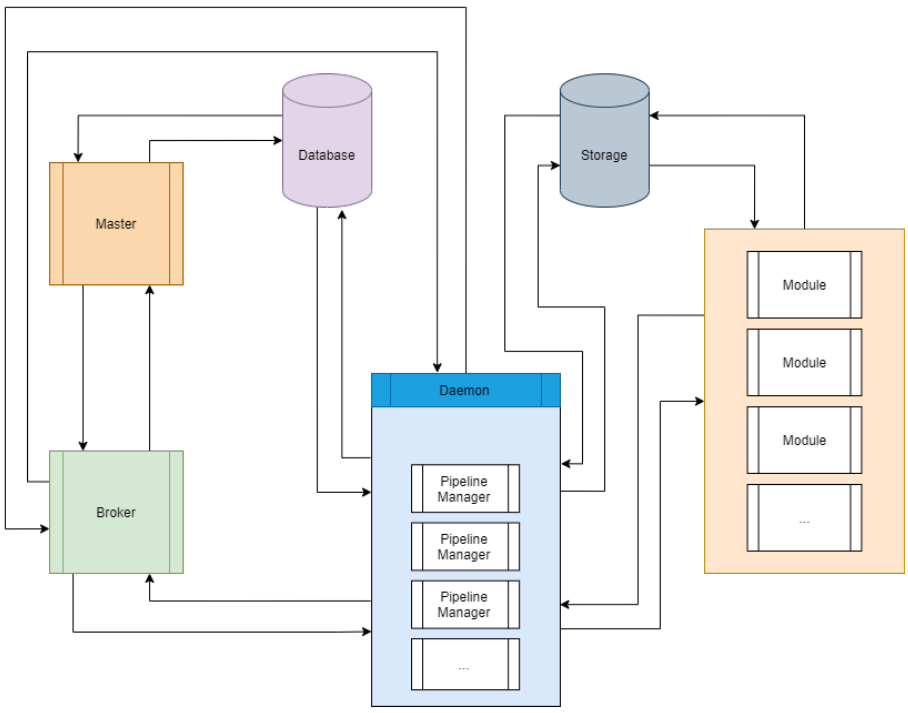


FIGURE 4.1: OrchRecon Diagram

4.1.1 Master

The Master component is the application entry point. It establishes a TCP connection with the Broker and accepts tasks with the following structure: <domain > <module > [<modules >]. It creates a record for the task in the Database, later associated to a *Target* object, and returns to a waiting state. It also receives notifications concerning the tasks state.

4.1.2 Broker

The communication between the Master and the Pipeline Managers is established through the Broker component, as illustrated in figure 4.2. It works like a proxy, routing traffic between the Master and the managers, and queues tasks if there is no available Pipeline Managers. It has a permanent connection with a Daemon present at every running instance and receives CPU usage metrics to decide in which of them a new Pipeline Manager should be started to accept an incoming task. This new Pipeline Manager will run in the instance with the lowest CPU usage in a certain time window, reason why the Broker acts also like a load balancer.

The communication between the components is done through a network connection over which an asynchronous messaging scheme is implemented.

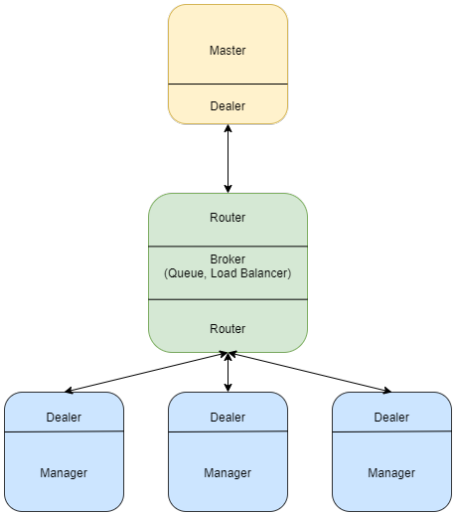


FIGURE 4.2: Broker Diagram

4.1.3 Pipeline Managers

The Pipeline Manager implements the core functionalities of the application. It is started by the Daemon and, as soon as it receives a task, it validates its structure. If the validation is successful, it creates a *Target* and a *Pipeline* objects.

A Target is an object mapped in the Database and has some properties like the submitted domain for the task at hand. The Pipeline is the set of tasks abstraction and, among its properties, it includes the set of tools (Modules) to run. As such, there is a validation of the indicated Modules availability and if it is possible to establish a directed graph between them, considering that each connected edge is established by similar output and input data types, respectively. For such, it resorts to a Breadth First search algorithm that establishes the connected Modules and those that can not be reached, as illustrated in Algorithm 1.

Algorithm 1: Breadth First Search Algorithm

```
Result: Directed Graph
queue = modules_to_run[0];
while queue do
    vertex = queue.pop() ;
    foreach module in modules_to_build_pipeline do
        if (module not in modules_to_run) and (vertex.out_type = module.in_type or
         module.in_type = "target") then
            queue.append(module);
            modules_to_run.append(module);
        end
    end
end
```

In the case of a successful validation, the Pipeline executes each Module, according to its configurations. When the option for paralleling is enabled within a certain Module, the Pipeline Manager splits the input data for the number of the established concurrent processes and assembles the respective outputs to a single final results file. Additionally, the Pipeline Manager updates the Target information in the database, concerning the Modules execution in order to enable its consultation during the process.

4.1.4 Database and Storage

As referred, each task is submitted to the Database by the Master component, although in a non validated state. Afterwards, and upon validation from the Pipeline Manager, it is used to keep the state of each Target, mapping some of its properties. In the case of an invalid task, the record is deleted.

Concerning Storage resources, the user may choose to use the instance hard-drive or a *Bucket* in the cloud. The first possibility is adequate for the local running mode or for the situations where the used instances persist after exiting the application. The second approach should be used in the cases where we resort to a Managed Instance Group, once the associated instances are eliminated at the end, being also an option to the other use cases.

4.1.5 Module

The Module component is responsible to provide the user the flexibility it may need for each tool. However, this implies a previous knowledge about their features. As previously referred, a Module is an abstraction for a specific tool that possesses a set of properties, as exemplified in listing 4.1, to control the latter's execution, demanding a coherent taxonomy in order to accomplish that. This coherence operates on two levels: (i) at the type of data level and (ii) at the command level.

In the first case, the user may use a chosen taxonomy that allows establishing a relation between the input and the output data from different Modules. As an example, if *Module ABC* produces an output suitable for the use of *Module XYZ*, the output data type of the first must adopt the exact reference of the second's input data type. As is often the case, a particular tool may have more than one functionality, either accepting different input data types and/or producing different output data types. As such, the user may configure several Modules based on the same particular tool, differentiating its input and/or output data types, as needed.

At the command level, the taxonomy is strict, once it reflects code details. The user needs to acquire some knowledge about it in order to properly configure the command property of each Module. In general, those references are related with read and write actions concerning tool's configuration or input and output files.

Another feature is related with Module's parallel execution. When the parallelism level is set to a value greater than one, the application will run the according number of

Module instances, splitting the input data among the running Module instances and, at the end, reassembling the output data.

```
{
  "name" : "subfinder",
  "in_type" : "target",
  "out_type" : "url",
  "command" : "docker run --rm -v _CONFIG_DIR_:/root/.config/subfinder -it ice3man
  /subfinder -d _TARGET_ -nW -silent > _PATH_/_OUTPUT_FILE_",
  "module_dir" : "/modules/subfinder",
  "concurrency_level" : 0,
  "parallelism_level" : 0
},
```

LISTING 4.1: Module configuration example

4.1.6 Scalability

The option for a distributed system architecture is linked to the foreseen need to scale up computing resources once some reconnaissance and vulnerability tools can be quite demanding or time consuming, allowing to scan several targets in parallel. It implements an asynchronous messaging approach and a simple load balancing function. The Broker has three interfaces, one connecting to the Master, which receives the tasks to perform, a second one to the Daemons for receiving information about the running instances and their CPU usage metrics, and the third to the Pipeline Manager(s) for routing the tasks sent by the Master and for receiving status information about the execution.

The Broker keeps a list of available Pipeline Managers in a pool and dispatches the incoming tasks to the first available one. In the case of unavailable managers, the tasks are queued and will be forwarded as soon as any of them finishes its workload and returns to the pool. It keeps also a list with the connected Daemons and the respective CPU usage metrics in order to request the deployment of a Pipeline Manager to the lowest resource usage instance.

Additionally, there is a heartbeat feature that allows for mutual connection acknowledgement between the Broker and the Master, the Pipeline Managers and the Daemons. The communication is made through TCP sockets and resorts to ZeroMQ, which provides a high-performance asynchronous messaging library [65]. ZeroMQ is at the core of the application communication, for what it will be detailed later.

4.2 Implementation

OrchRecon is implemented in Python in order to easily prototype the system, as it allows for an object-oriented approach and has a wide collection of third-party libraries to assist in the development of such a project. Among those third-party libraries some stand out because of the core functionalities they assure: ZeroMQ and SQLAlchemy, responsible for implementing the messaging layer and the database interaction, respectively. There are four main components, Master, Broker, Pipeline Manager and Daemon which are implemented with the support of a set of classes, as illustrated in figure 4.3, where the most relevant are referenced.

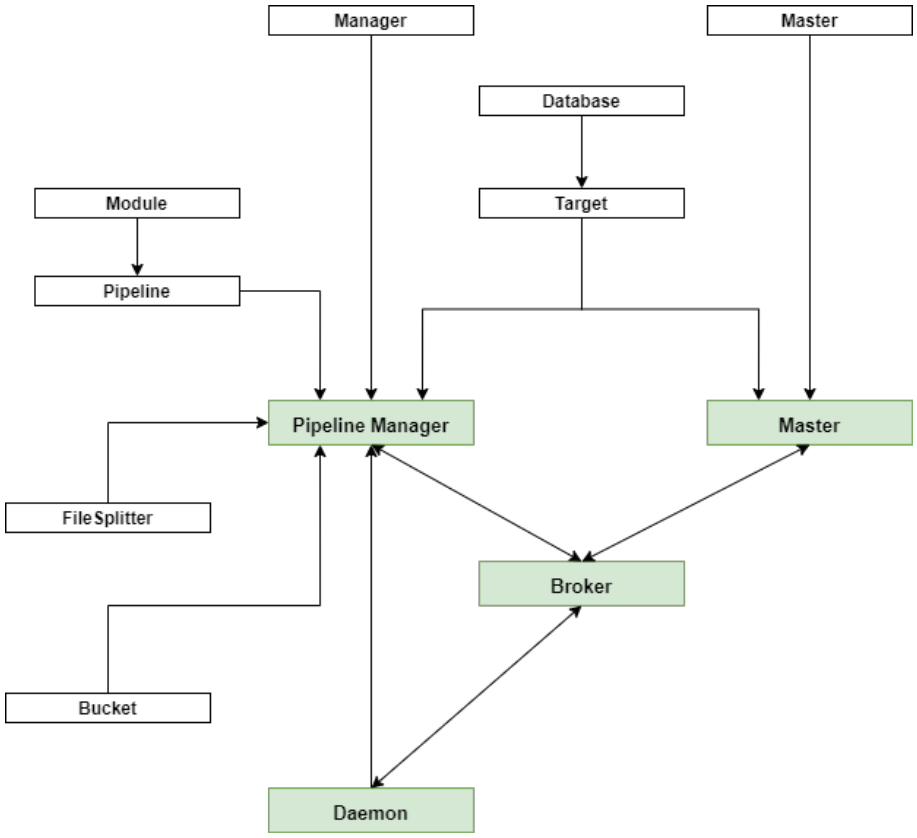


FIGURE 4.3: Class Diagram

4.2.1 Setup

To run OrchRecon in Google Cloud environment, there are some preliminary steps. First of all the user should create service account credentials, store the generated file in *creds* folder and update *bucket.py* with the appropriate file path and bucket name. Then, 2

instances shall be created: the first to run the Database and the Broker; the second to create a base image for the *Managed Instance Group* (MIG). Both instances shall be deployed with Ubuntu 18.04 LTS in a 20 GB disk and allow for HTTP and HTTPS connections. Taking advantage of the internal network capabilities, namely DNS service, the Broker instance should be named *broker* in the network interface properties.

After copying the application files to each of the instances, the user should run *install_requirements.sh* script in each one to install all the necessary libraries and build the necessary docker containers. It also creates a service in each of the instances to start automatically the respective component: Broker and Daemon. Afterwards, the second instance shall be used to create an image that will be the base of the MIG, which automatically deploys or stops instances according to computation needs.

After finishing the installation, the user may connect to the application from his local terminal running the script *cloud.init* which will start the Broker and will deploy a MIG from the created base image. As they run the respective services on startup, OrchRecon is ready to receive a pipeline to execute.

4.2.2 Master

At the present, the Master merely sends to the Broker new tasks to perform and listens to incoming informative messages. It is built on top of the *Master* class that allows to abstract all the communication details: connection to the Broker, message sending and receiving, and heartbeating related parameters. This class implements also a thread no keep the heartbeating messages flow.

On startup, an object *Master* is created, receiving as parameters an identifier and the endpoint of the Broker interface (see Listing 4.2).

```
1   class Master:
2
3       def __init__(self, tag, broker):
4           self.name = tag  # String: Broker ID
5           self.broker = broker  # String: tcp endpoint
6           self.context = zmq.Context()  # zmq Context
7           self.socket = None  # zmq Socket
8           self.poller = zmq.Poller()
9           self.broker_status = False  # Broker connection status
10          self.broker_connection()
11          self.next_heartbeat = None
12          self.queue_liveness = None
```

```
13
14              _thread.start_new_thread(heartbeat, (self,))
15
16          def update_heartbeat_params(self):
17              self.next_heartbeat = time.time() + configs.HEARTBEAT_INTERVAL
18              self.queue_liveness = configs.HEARTBEAT_LIVENESS
```

LISTING 4.2: Master class

Then, a parallel thread is also started to keep the heartbeating message sending to the Broker, as illustrated in listing 4.3, moment from which the Master is ready to receive a task input or an incoming message from the Broker.

```
1   # Function to run in autonomous thread to keep
2   # Heartbeat
3   def heartbeat(master):
4       while True:
5           rep = [b'', configs.HEARTBEAT]
6           master.send(rep)
7           time.sleep(configs.HEARTBEAT_INTERVAL)
```

LISTING 4.3: Heartbeat function

The Master accepts messages of the *Task* and *Heartbeat* types. When a new task message arrives, it is displayed to the user.

Additionally, the Master checks the Broker's heartbeat messages to validate its liveness, or to determine the connection is lost, after a waiting period of ten seconds (see listing 4.4).

```
1   """
2   Check Broker liveness
3   """
4   if time.time() > master.next_heartbeat:
5       master.queue_liveness -= 1
6
7   if master.queue_liveness == 0:
8       print(f'[MASTER {ident}] {configs.B_NOT_CONNECTED} | Waiting 10 seconds for
    reconnection')
9       time.sleep(10)
10
11  if master.queue_liveness < 0:
12      exit()
```

LISTING 4.4: Master check liveness

4.2.3 Broker

The Broker has the core role concerning the system communication, resorting to ZeroMQ library to implement the socket interfaces with the Master, the Pipeline Managers and the Daemons (see Listing 4.5). The chosen communication pattern is ZeroMQ's *Dealer - Router* in order to allow for a complete asynchronous communication. Therefore, when a *Request - Reply* pattern is needed, it must be coded in the application logic.

```
1   # Prepare context and sockets
2   context = zmq.Context()
3   frontend = context.socket(zmq.ROUTER)
4   frontend.bind("tcp://*:5000")
5
6   backend = context.socket(zmq.ROUTER)
7   backend.bind("tcp://*:6000")
8
9   daemon = context.socket(zmq.ROUTER)
10  daemon.bind("tcp://*:9000")
11
12  poller = zmq.Poller()
13  poller.register(frontend, zmq.POLLIN)
14  poller.register(backend, zmq.POLLIN)
15  poller.register(daemon, zmq.POLLIN)
```

LISTING 4.5: Socket interfaces

The Broker polls out these three interfaces periodically, according to an established heartbeat interval, and processes messages according to its origin and type. At present there are three types of messages: Task, Info and Heartbeat. Whenever a message reaches the Broker, it is dispatched to the appropriate recipient, if applicable, after a transformation in order to keep its structure coherent through all the system. A message is a bytes list with the following structure: [<sender_address >, b", <service_type >, <service_info >, <payload >, ...].

Additionally, the respective dictionary that keeps track of each connected element is updated. This feature is implemented through a class that is instantiated every time a connection arrives the Broker(see Listing 4.6).

```
1    class ConnectedElement:
2        def __init__(self: object, address: str, status=None, load=None):
3            self.address = address
4            self.expiry = time.time() + configs.HEARTBEAT_INTERVAL * configs.
     HEARTBEAT_LIVENESS
5            self.status = status
6            self.liveness = configs.HEARTBEAT_LIVENESS
7            self.load = load  # Used with Daemons
```

LISTING 4.6: ConnectedElement class

Therefore, an updated record of every connected element is kept in order to check its liveness. In the Daemons case, the CPU usage is also kept to provide the ground to the instance's choice where a new Pipeline Manager shall be deployed. Whenever a connected element fails to communicate before the set expiry time, its liveness is decreased and, as soon as it reaches zero, the Broker assumes the connection is lost, withdrawing that element from the respective dictionary (see Listing 4.7).

```
1    def check_liveness(elements: ConnectedElement):
2        t = time.time()
3
4        for _, e in elements.items():
5            if t > e.expiry:
6                e.liveness -= 1
7
8        dead_elements = []
9        for adr, e in elements.items():
10           if e.liveness == 0:
11               dead_elements.append(adr)
12
13       for d in dead_elements:
14           elements.pop(d)
```

LISTING 4.7: check liveness method

This liveness check is performed periodically, aligned with the interfaces poll out, depending on the settings for the heartbeat interval and for the tolerance for non responsiveness.

The Broker has also a queuing function in the event of unavailable Pipeline Managers. Indeed, all the tasks go through the queue, in order to keep the messaging service simpler and to deal with Pipeline Manager's starting process. When a new task arrives to the Broker, it causes the start of a new Pipeline Manager and the queuing of the respective task (see listing 4.8).

```
1    # Process TASK messages
2    if service == configs.TASK:
3        info = request[3]
4
5        # Process NEW TASK messages
6        if info == configs.T_NEW:
7
8            # Ask Daemon for a new Pipeline Manager
9            # Find lowest CPU usage machine
10           low_cpu_deamon = find_lowest_load(daemons)
11
12           # Send START message
13           daemon_request = [low_cpu_deamon.address, b'', configs.TASK, configs.
     W_START]
14           daemon.send_multipart(daemon_request)
15
16           queue.append(request)
```

LISTING 4.8: Pipeline Manager start

4.2.4 Pipeline Managers

The Pipeline Manager component implements the core functionalities that materialise the actions needed to carry out a certain task entered by the user. It is built on top of the *Manager* class that allows to abstract all the communication details: connection to the Broker, message sending and receiving, and heartbeating related parameters. This class implements also a thread no keep the heartbeating messages flow. The application logic depends on two other classes: *Target* and *Pipeline*. The first implements a Target object that represents the domain being tested and is mapped in the Database. The latter implements all the actions related to the Modules associated to the task, like pipeline validation or their execution order.

On startup, an object *Manager* is created, receiving as parameters an identifier and the endpoint of the Broker interface (see Listing 4.9). As referred, a parallel thread is also

started to keep the heartbeating process and, then, the Pipeline Manager waits for an incoming message.

```
1   class Manager:
2
3       def __init__(self, tag, broker):
4           self.name = tag  # String: Pipeline Manager ID
5           self.broker = broker  # String: tcp endpoint
6           self.context = zmq.Context()  # zmq Context
7           self.socket = None  # zmq Socket
8           self.poller = zmq.Poller()
9           self.broker_status = False  # Broker connection status
10          self.broker_connection()
11          self.next_heartbeat = None
12          self.queue_liveness = None
13
14          _thread.start_new_thread(heartbeat, (self,))
```

LISTING 4.9: Manager class

The Pipeline Manager accepts messages of the *Task* and *Heartbeat* types. When a new task message arrives, the task structure (<domain > <module > [<modules >]) is validated in two steps: number of parameters and feasibility of the the chosen pipeline. A *Pipeline* object is created and the requested Modules are validated (see Listing 4.10) for which, on success, an execution path is determined through a breadth first algorithm (see listing 4.11).

```
1   def validate_input_modules(self):
2
3       with open("modules.json") as f:
4           configured_modules = json.load(f)
5
6       check = False
7
8       for in_mod in self.input_modules:
9           check = False
10
11          for conf_mod in configured_modules:
12              if conf_mod['name'] == in_mod:
13                  self.modules_to_build_pipeline.append(create_module_instance(
    conf_mod))
14
15                  check = True
```

```
16                      break
17
18              if not check:
19                  self.modules_not_available.append(in_mod)
20                  break
21
22          return check
```

LISTING 4.10: Modules validation

```
1     def bfs_pipeline_sequence(self):
2
3         queue = [self.modules_to_run[0]]
4
5         while queue:
6             vertex = queue.pop()
7
8             for node in self.modules_to_build_pipeline:
9                 if node not in self.modules_to_run and (vertex.out_type == node.
      in_type or node.in_type == 'target'):
10                    queue.append(node)
11                    self.modules_to_run.append(node)
```

LISTING 4.11: Breadth First algorithm

To accomplish that, the *Pipeline* object requires a *Module* object that maps the Module's configurations, in order to have the information about their input and output types. When an execution pipeline is determined, each Module it contains will be executed according to its parallelism level. If the user sets a Module to run in parallel, the original input file will be splitted according to that parallelism level and the execution will be done in asynchronous parallel sub-processes. Otherwise, the Module is executed in a single sub-process.

When an error is produced during a Module's execution, there is a simple error recovery mechanism that tries to repeat its execution a certain number of times, after which the application flow continues. After the pipeline is completed or in the cases where an error was produced by the validation steps the Pipeline Manager shuts down.

4.2.5 Database and Storage

Due to the application's dual mode of execution, locally or in a cloud environment, the approach to the Database and to the Storage resources followed a similar logic. Therefore, when the application is executed locally, the MySQL service must be running at the "localhost" and, when in a cloud environment, it should run in a defined instance, for which there is the need to resort to the DNS services of the cloud infrastructure.

As referred, the Target object maps the information about a task in the Database, like the domain under analysis, the task's status or the Modules' status, keeping it updated through a set of methods that resort to SQLAlchemy library [79]. This library provides an object-relational mapper (ORM) that allows to map classes to a database, providing an object-oriented approach while abstracting from some of the SQL issues.

The Database class, as illustrated in listing 4.12 creates an object to establish a connection with the MySQL server, which will be used by the Target class to implement data transactions with it.

```
1   class Database:
2       def __init__(self):
3
4           self.engine = sqlalchemy.create_engine('mysql://{}:{}@database/orchrecon
    '.format(username, password))
5           Session = sessionmaker(bind=self.engine)
6           self.session = Session()
```

LISTING 4.12: Database class

As illustrated in listing 4.13, this library allows to map an object into a record in the database, favouring code readability and maintaining, at the same time, the control over the implementation of the ORM.

```
1   from sqlalchemy.ext.declarative import declarative_base
2   from database import Database
3
4   [REDACTED]
5
6   # SQLAlchemy settings
7   Base = declarative_base()
8
9   Column = sqlalchemy.Column
10  Integer = sqlalchemy.Integer
```

```
11      String = sqlalchemy.String
12      JSON = sqlalchemy.JSON
13
14      class Target(Base):
15          # Target object maps "target" in DB
16          __tablename__ = 'targets'
17
18          id = Column(Integer, primary_key=True)
19          id_target = Column(String)
20          domain = Column(String)
21          url = Column(String)
22          target_status = Column(String)
23          finished_tasks = Column(JSON)
```

LISTING 4.13: Target class

At the present example, a Target object maps a record in the "targets" table of the database and any change in the first just needs to be commited to the latter (see listing 4.14).

```
1       def save_target(self):
2           try:
3               self.database_session.commit()
4               print("[TARGET] Target updated into DB!")
5
6           finally:
7               self.database_session.flush()
```

LISTING 4.14: Save Target Function

Concerning storage, all the produced data is stored locally in a folder named from the identifier attributed to the task (see Listing 4.15).

```
1       def create_folder(self):
2           if not os.path.isdir(self.id_target):
3               os.mkdir(self.id_target)
```

LISTING 4.15: Folder creation

However, when the application is executed with the Bucket option enabled, the data that must persist is stored in a cloud storage instance (Bucket) and the transient data

continues to be managed locally, following the same organization and naming principles (see Listing 4.16).

```
1 if configs.BUCKET:
2
3     source_file_name = f'{MAIN_PATH}/{target.id_target}/{target.domain}.{module.
      out_type}'
4     destination_blob_name = f'{target.id_target}/{target.domain}.{module.out_type}'
5     bucket.upload_blob(source_file_name, destination_blob_name)
```

LISTING 4.16: Bucket file upload

As such, there is a *Bucket* class to implement the needed methods for uploading and retrieving files, while managing the authentication towards the cloud storage service.

4.2.6 Module

As previously referred, a Module is an abstraction for a specific tool that possesses a set of properties, as exemplified in listing 4.1. It is implemented through a *Module* object that maps the mentioned properties and is referenced by the *Pipeline* and the *Target* objects.

The *command* property has a generalisation of the command to execute the respective Module, which will be processed by a method at the Pipeline Manager, in order to convert that generalisation into a concrete command. Therefore, a strict taxonomy is implemented with self-descriptive terms like *_OUTPUT_FILE_*, *_PATH_*, *_TARGET_* or *_INPUT_FILE_* that will be replaced by the aforementioned method, as illustrated in listing 4.17.

```
1     def build_command(target, module, input_file, output_file):
2         cmd = module.command.replace("_OUTPUT_FILE_", output_file)
3         cmd = cmd.replace("_PATH_", "'%s'" % MAIN_PATH + "/" + target.id_target)
4         cmd = cmd.replace("_CONFIG_DIR_", "'%s'" % MAIN_PATH + module.module_dir)
5         cmd = cmd.replace("_TARGET_", target.url)
6         cmd = cmd.replace("_INPUT_FILE_", input_file)
7
8         return cmd
```

LISTING 4.17: Command building

The *in_type* and the *out_type* properties are related with the data consumed and produced by each Module. This data is represented by files and the mentioned *out_type* property is appended to the respective files to be recognised by the Modules that accept that

kind of data through their *in_type* property. As depicted in listing 4.18, we may conclude that the *httprobe* Module consumes the data produced by the *subfinder* one.

```
[
  {
    "name" : "subfinder",
    "in_type" : "target",
    "out_type" : "url",
    "command" : "docker run --rm -v _CONFIG_DIR_:/root/.config/subfinder -it ice3man
    /subfinder -d _TARGET_ -nW -silent > _PATH_/_OUTPUT_FILE_",
    "module_dir" : "/modules/subfinder",
    "concurrency_level" : 0,
    "parallelism_level" : 0
  },
  {
    "name" : "httprobe",
    "in_type" : "url",
    "out_type" : "web.url",
    "command" : "cat _PATH_/_INPUT_FILE_ | docker run --rm -i httprobe --prefer-
    https > _PATH_/_OUTPUT_FILE_",
    "module_dir" : "/modules/httprobe",
    "concurrency_level" : 0,
    "parallelism_level" : 4
  }
]
```

LISTING 4.18: Module properties

As is often the case, a particular tool may have more than one functionality, either accepting different input data types and/or producing different output data types. Therefore, it is possible to configure several Modules based on the same tool just by adjusting its input and/or output data types as well as the command generalisation.

Managing data with this approach gives the flexibility to run tools in multiple forms, from command line scripts to container execution, provided that the data is consumed from and produced to files. At present, our main approach resorts to docker containers as it prevents conflicts between several tools' dependencies.

The *parallelism_level* property controls how the Module is executed. If the value is greater than one, the application checks the input length and divides the data, if justifiable, for a number of concurrent sub-processes not greater than the defined level (see listing 4.19).

```
if '_INPUT_FILE_' in module.command and module.parallelism_level > 1:

    if configs.BUCKET:
        source_blob_name = f'{target.id_target}/{target.domain}.{module.in_type}'
        destination_file_name = f'{MAIN_PATH}/{target.id_target}/{target.domain}.{
    module.in_type}'
        bucket.download_blob(source_blob_name, destination_file_name)

    input_file = f'{MAIN_PATH}/{target.id_target}/{target.domain}.{module.in_type}'

    temp_file_path = f'{MAIN_PATH}/{target.id_target}/'

    # Split input file into temporary files, according to a split factor
    splitter = FileSplitter(input_file, module.parallelism_level, temp_file_path)

    tasks = []
    temp_file_id = 1
    temp_files = []

    # Update module status
    target.set_finished_tasks(module.name, module.status)

    for temp_input_file in splitter.splitted_files:
        """
        Starts n parallel processes, according to a split factor
        """
        temp_output_file = f'{target.domain}.temp_{temp_file_id}'

        # Build command to run
        new_cmd = build_command(target, module, temp_input_file, temp_output_file)
        tasks.append((loop.create_task(run_cmd_2(new_cmd))))
        temp_files.append(temp_output_file)
        temp_file_id += 1

    await asyncio.wait(tasks, return_when=asyncio.FIRST_EXCEPTION)
```

LISTING 4.19: Parallel execution

After all the sub-processes are finished, the temporary files are reassembled in a single output file and deleted. These operations over the input and temporary files are implemented through the methods in the *FileSplitter* class.

The *parallelism_level* property should be set with some aspects in mind, namely the overhead that multiple sub-processes may cause, which could not improve the overall

performance at all. Some fine-tuning is needed considering also that some tools are multi-threaded and could be more efficient to adjust the number of threads on the tool's settings.

4.2.7 Distributed system

The application here presented implements a message-oriented middleware to enable easy scalability in order to enhance the chosen tools by the user. Although it works in a local mode of execution, the scalability it pretends to reach is better obtained with the use of cloud resources which, nowadays, are easily available and relatively affordable. Therefore, it shall be assumed that we will be referring to that mode, unless otherwise stated.

Infrastructure

Concerning the infrastructure, we resorted to Google Cloud Computing services [78] to run a Broker and a Database in the same instance and a pool up to three instances to run the Pipeline Managers. This pool of instances, designated a *Managed Instance Group* (MIG) [80], may automatically scale up and down according to a CPU usage threshold of 20%. All these instances are a clone of the same image that is already fully configured with the necessary Modules and related dependencies.

The referred instances run in the same internal network and resort to its DNS services to reach the Broker and Database instance. Additionally, the Broker exposes an external IP that is obtained through the initialisation function to provide to the Master, so it can be operated from an external machine.

Messaging service

As already referred, the messaging core functionality is assured by the Broker. It is implemented over a *Dealer - Router* asynchronous communication pattern provided by ZeroMQ [65]. Although this pattern is more demanding in terms of implementation, it provides more flexibility than a simple *Request - Reply* as it does not block the execution. As such, when this kind of communication is necessary, that must explicitly coded, as illustrated in listing 4.20.

```
1    reply = self.receive()
2
3    while not reply:
4        reply = self.receive()
```

LISTING 4.20: Request-Reply approach

Dealer - Router enables an easy scaling approach as it just needs one "fixed" point, the Router IP, allowing for an arbitrary number of Dealers to connect. That approach was followed to connect the Broker with the Master, the Daemons and the Pipeline Managers as illustrated in figure 4.4.

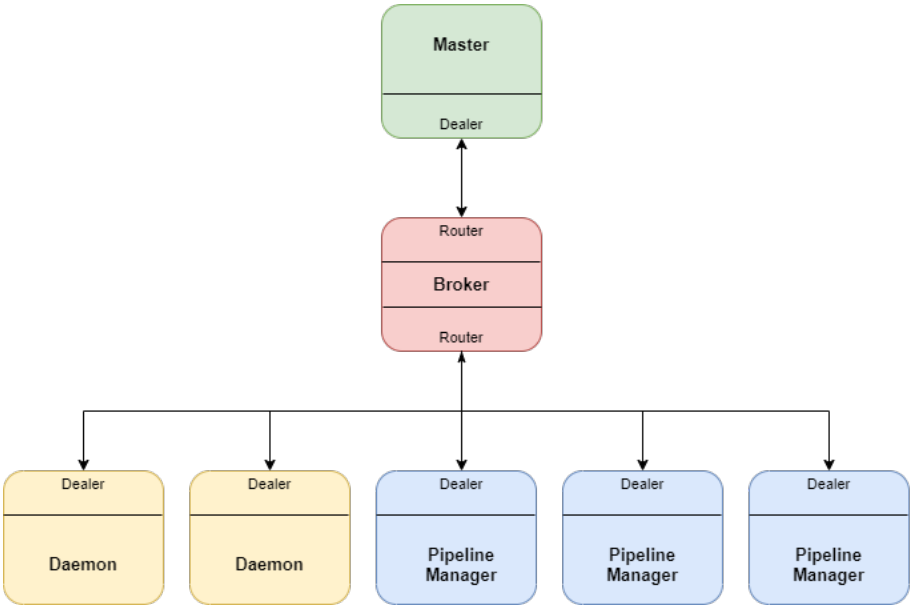


FIGURE 4.4: Dealer-Router pattern

As stems from the image, the Broker needs three interfaces to communicate with these three kinds of components, as already illustrated in listing 4.5. It polls incoming sockets at a defined rate and, when receives messages from any of the interfaces, routes them according to the service type, the service info and the payload, if present (see listing 4.21).

```
1     while True:
2
3         # Socket poller: Backend + Frontend + Daemons
4         socks = dict(poller.poll(configs.HEARTBEAT_INTERVAL * 1000))
5
6         """
7         Messages arrived from daemon
8         """
9         if socks.get(daemon) == zmq.POLLIN:
10
```

```
            request = daemon.recv_multipart()
            service = request[2]
            address = request[0]
            load = request[3]

[REDACTED]

        """
        Messages arrived from the backend (Pipeline Managers)
        """
        if socks.get(backend) == zmq.POLLIN:

            request = backend.recv_multipart()
            service = request[2]
            address = request[0]

[REDACTED]



        """
        Messages arrived from the frontend (Master - may exist more than one)
        """
        if socks.get(frontend) == zmq.POLLIN:

            request = frontend.recv_multipart()
            service = request[2]
            address = request[0]
```

LISTING 4.21: Message polling

Thus, the messaging structure must be explicitly implemented, as there is the need to keep record of the addresses where to reply. As already pointed, a message is a bytes list with the following chosen structure: [<sender_address >, b", <service_type >, <service_-info >, <payload >, ...]. Therefore, when a message must be forwarded by the Broker, it must be transformed, so the recipient receives the same structure. These structures are flexible, with the exception of the first two elements of the list: the sender's address and an empty byte. In order to keep compatibility with ZeroMQ message envelopes, the empty byte works as a delimiter for the sender's address in a coded message approach like the one followed. The listing 4.22 illustrates the transformation operated in a message received from a Pipeline Manager that is being forwarded to the Master.

```
    # Process TASK messages
```

```
2     if service == configs.TASK:
3         info = request[3]
4 
5         # Process FORMAT ERROR, FINISHED TASK and INVALID PIPELINE in TASKS service
6         if info == configs.P_TASK_FORMAT_ERROR or \
7            info == configs.T_FINISHED or \
8            info == configs.P_INVALID_PIPELINE:
9 
10 [REDACTED]
11 
12            task_sender = request[4]
13            task = request[5]
14            forward_reply = [task_sender, b'', service, info, task]
15            frontend.send_multipart(forward_reply)
```

LISTING 4.22: Message transformation

In order to structure the messaging service, and considering future improvements and enhancements, we conceived three types of services, Task, Info and Heartbeat, in which each service has associated information messages that relate to the service or sender status. In the present, only new Task messages are queued, as the others are, basically, status messages that do not occur when the components are disconnected. The new tasks queue is implemented in a *first-in-first-out* approach, as illustrated in listing 4.23.

```
1     # Process queued tasks
2     if queue:
3         popped_task = queue.pop(0)
4 
5         # Check available Pipeline Manager
6         available_worker = check_available_element(workers, configs.P_READY)
7 
8         if available_worker:
9             send_task_to_manager(available_worker, popped_task, backend, workers)
10        else:
11            # If there is none workers available, Task returns to queue
12            queue.append(popped_task)
```

LISTING 4.23: Queue implementation

Chapter 5

Evaluation

To evaluate this prototype we performed two kind of tests: (i) Module's performance related to the parallelism level and (ii) system's performance related to simultaneous pipelines execution. We resorted to Google Cloud services to run different kinds of instances and chose a limited number of Modules to run against one target, acronis.com, which has a public bug bounty program in HackerOne [81] with a wide scope concerning its subdomains.

The selected Modules for the pipeline were *Subfinder*, *httprobe* and *Nuclei*. Subfinder [82] gathers subdomains of a given target in a passive way, querying public sources. This initial list is then tested with httprobe [83] to extract working servers. At last, Nuclei [84] detects the technologies associated to the working servers. Although we used Nuclei only for detecting the exposed technologies, it has wider capabilities, like detecting misconfigurations and a wide range of Common Vulnerabilities and Exposures (CVE).

Subfinder Module was never paralleled during the testing phase as it takes only a single input, the target domain. However, as it resorts also to a word-list for testing, we should envision for future work to have a similar mechanism for those Modules that, beyond the input from another tool, also have secondary inputs like the alluded word-list.

5.1 Modules Performance

In this test we ran *httprobe* and *Nuclei* with the default configurations in three instances of 1, 2 and 4 virtual CPU and with 3.75, 7 and 15 GB of RAM memory, respectively, which refer to Google's machine type *n1-standard-1*, *n1-standard-2* and *n1-standard-4*. The CPU platform was *Intel Haswell* and the Operative System was Ubuntu 18.04 LTS. Furthermore,

each Module was tested with several parallelism levels (1, 2, 4, 8, 16 and 32) in sets of ten executions.

The graphics (5.1, 5.2, 5.3) illustrate the obtained results of *httprobe* per instance. On the left axis we have the mean execution time in seconds, on the right the standard deviation and on the bottom side the parallelism level.

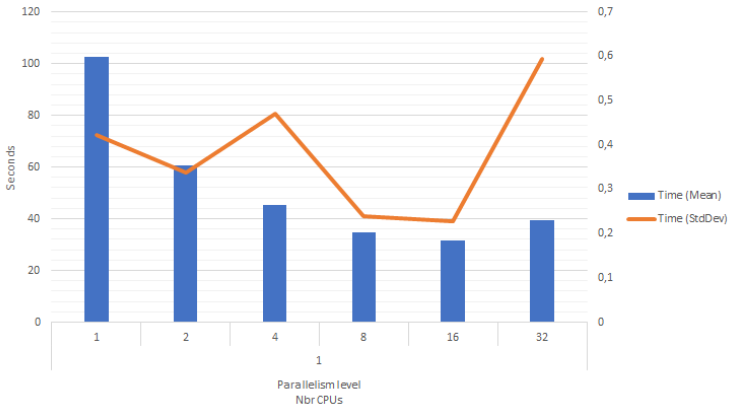


FIGURE 5.1: httprobe - 1 vCPU

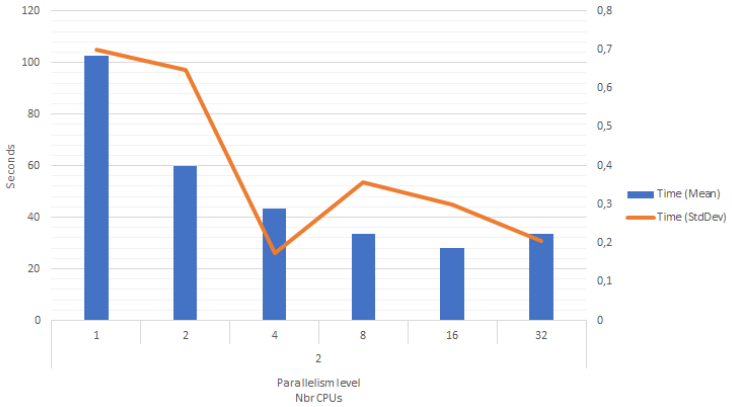


FIGURE 5.2: httprobe - 2 vCPU

The tool behaviour was stable, validating between 291 and 292 subdomains. As the parallelism level increases we observe a reduction in the execution time until the level of 16, although the performance gain is not linear. We also observe that the increase in computation resources has a growing impact in the total execution time as the parallelism level is increased as summarised in table 5.1.

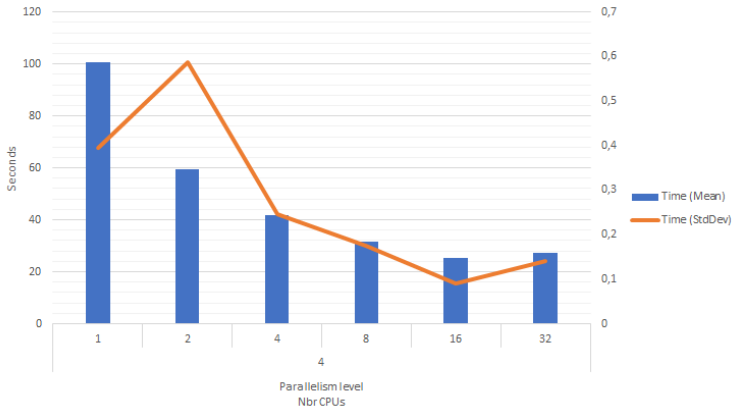


FIGURE 5.3: httprobe - 4 vCPU

Parallelism Level	1		2		4		8		16		32	
Nbr. vCPU	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**
1	102,68	0,42	60,74	0,34	45,27	0,47	34,55	0,24	31,60	0,23	39,36	0,59
2	102,64	0,70	59,98	0,65	43,41	0,17	33,52	0,36	28,17	0,30	33,40	0,21
4	100,47	0,39	59,45	0,59	41,83	0,25	31,52	0,17	25,17	0,09	27,25	0,14

TABLE 5.1: httprobe - Test results

Considering each testing instance in isolation, we observe performance gains for *httprobe* ranging from 69% to almost 300%, assuming the base value at parallelism level 1 in each instance, as illustrated in figure 5.4.

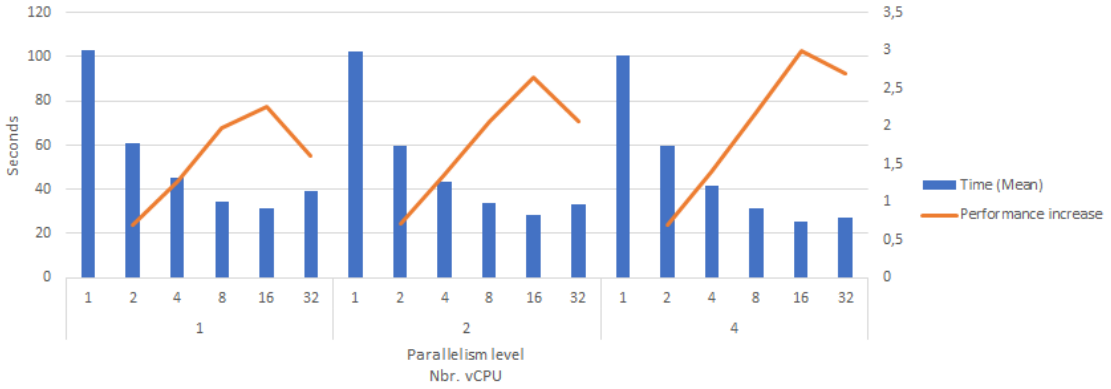


FIGURE 5.4: httprobe - Performance metrics

Concerning the tests with *nuclei*, the following graphics (5.5, 5.6, 5.7) illustrate the results. From the previously validated subdomains, *nuclei* received a list of 291 to test which technologies were present. Its behaviour was less stable, ranging from 513 to 580 detections, with 93.75% of the tests ranging between 540 and 580 detected technologies.

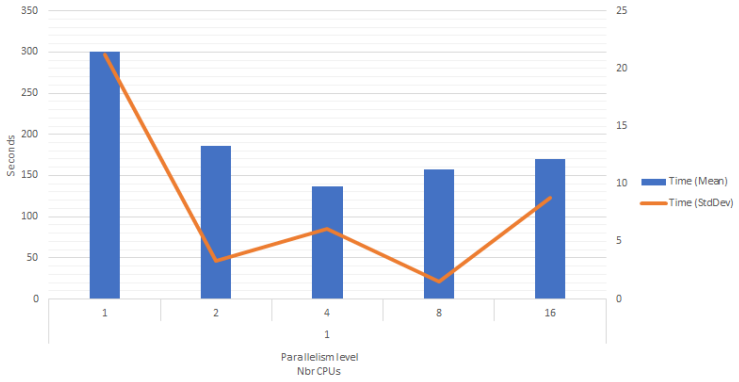


FIGURE 5.5: nuclei - 1 vCPU

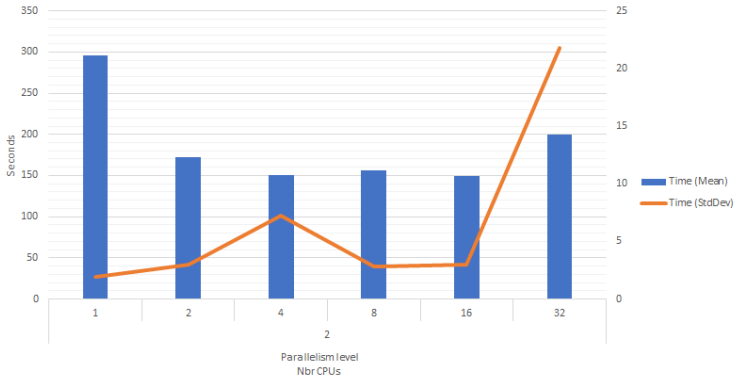


FIGURE 5.6: nuclei - 2 vCPU

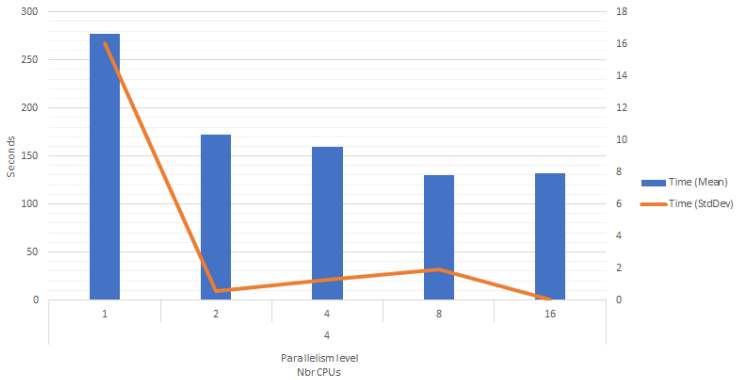


FIGURE 5.7: nuclei - 4 vCPU

As the parallelism increases, we observe a reduction in the execution till a certain level. As with the other Module, the performance gain is not linear and the increase of computation resources, although with an impact in the overall performance, has not the same stable results, as illustrated in table 5.2

As can be observed from the results, we only succeeded to obtain data for parallelism level of 32 with a 2 vCPU instance, fact for which we do not have an explanation, at the

moment. Furthermore, those results are considerably worst than the ones for parallelism level of 16. As we will see in the pipeline performance tests, a similar behaviour will be observed, leading instances to a blocking state.

Parallelism Level	1		2		4		8		16		32	
Nbr. vCPU	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**
1	300,13	21,19	185,95	3,28	136,87	6,07	157,85	1,55	170,38	8,80		
2	296,33	1,95	171,78	2,96	150,67	7,22	156,04	2,83	149,57	3,02	199,48	21,82
4	277,43	16,04	171,88	0,58	159,55	1,25	129,64	1,90	132,05	0,01		

TABLE 5.2: Nuclei (Technologies) - Test results

Considering each testing instance in isolation, we observe some performance gains for *nuclei*, although there is a shorter improvement margin when compared with *httprobe*. Nevertheless, we still were able to record values ranging 61% to 120%, assuming the base value at parallelism level 1 in each instance, as illustrated in figure 5.8.

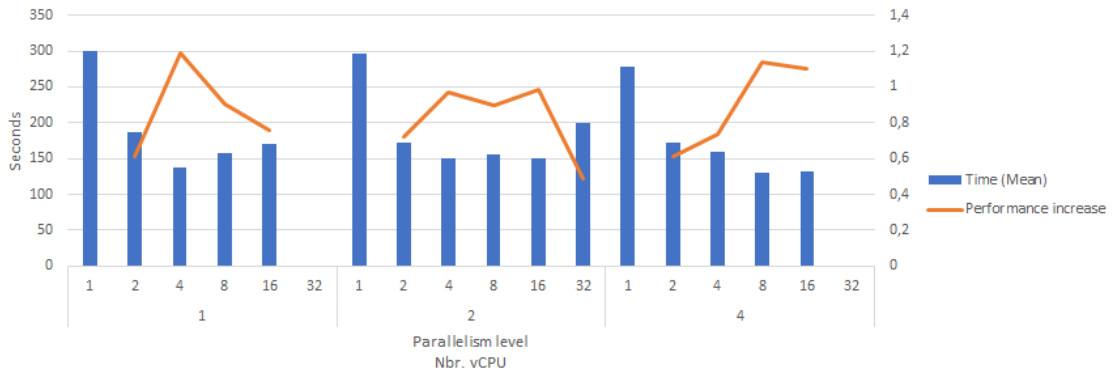


FIGURE 5.8: Nuclei - Performance metrics

As previously referred, each Module was ran with its default configurations and, from their public information, *httprobe* and *Nuclei* run with 20 and 10 concurrent processes by default, respectively. Comparing results from both Modules, they suggest that each one has a better parallelism level, independent from the other. However, further testing is needed once the Modules were tested in isolation.

5.2 Pipeline Performance

This test intended to measure the application's performance in the scenario of multiple pipelines running simultaneously. We resorted to the same three types of instances and executed the referred pipeline, *Subfinder*, *httprobe* and *Nuclei*, in sets of ten simultaneous

runs. Due to resources limitations imposed by Google, we could not have three 4 vCPU instances operating simultaneously.

The graphics (5.9, 5.10, 5.11) illustrate the obtained results. On the left axis we have the mean execution time, on the bottom axis the parallelism level and for each of these levels we find the readings for each scenario: 1, 2 or 3 instances.

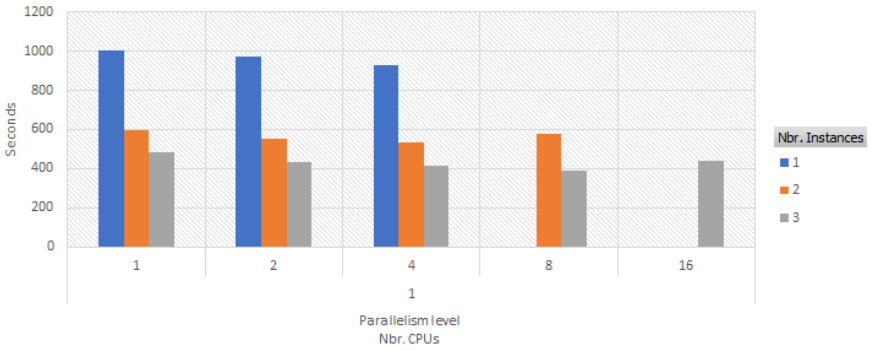


FIGURE 5.9: Pipeline - 1 vCPU

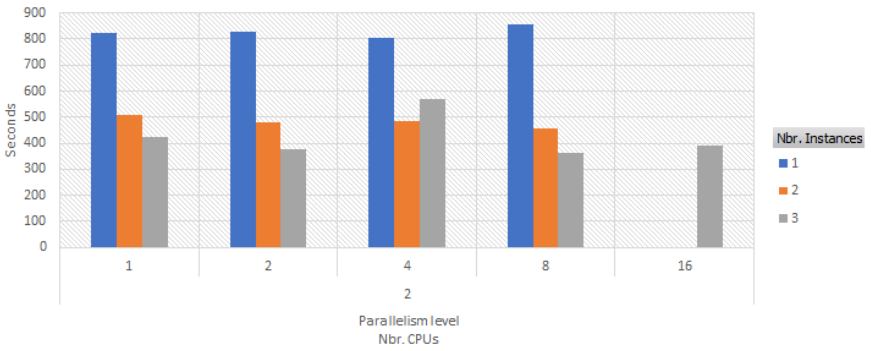


FIGURE 5.10: Pipeline - 2 vCPU

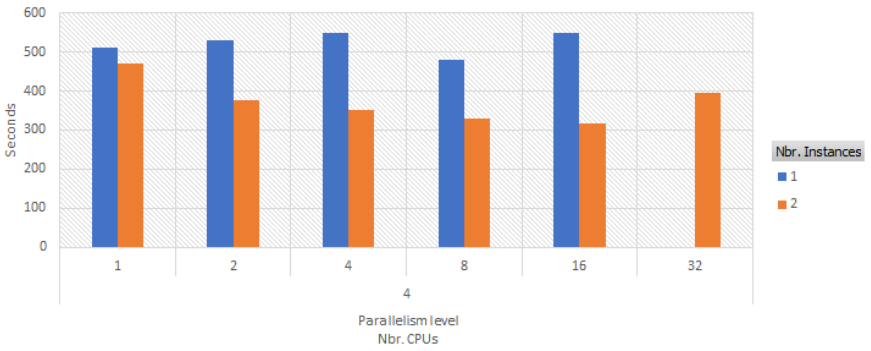


FIGURE 5.11: Pipeline - 4 vCPU

From the results, we observe that there is a performance gain as we scale up to more instances or when increasing computation resources. Furthermore, raising the parallelism level contributed to a performance gain in 60,7% of the times.

As we can observe from the results (table 5.3) some tests did not succeeded as the instances entered in a blocking state. When the parallelism level was set too high for the available resources and the 10 pipelines were executing simultaneously, the instances stopped processing and became unresponsive.

Nbr Instances		1		2		3	
Nbr vCPU	Parallelism level	Time	StdDev	Time	StdDev	Time	StdDev
1							
	1	1006,25	28,60	597,54	71,87	484,49	54,51
	2	974,17	38,90	555,24	36,11	433,27	80,81
	4	926,23	24,92	533,59	91,39	412,96	72,36
	8	*a*)	-	577,10	78,91	391,57	48,50
	16	*a*)	-	*a*)	-	436,86	59,40
2							
	1	823,36	29,24	509,15	34,88	425,56	31,13
	2	828,05	29,61	480,85	90,75	379,01	57,19
	4	805,93	40,38	485,23	59,58	569,61	159,67
	8	856,44	25,72	456,80	86,31	362,15	72,81
	16	*a*)	-	*a*)	-	390,12	69,97
4							
	1	512,43	27,79	470,21	43,12	-	-
	2	530,20	47,20	376,46	65,67	-	-
	4	547,43	80,60	350,85	58,71	-	-
	8	481,11	28,59	329,53	37,14	-	-
	16	548,71	24,38	318,01	38,68	-	-
	32	*a*)	-	395,51	74,91	-	-

TABLE 5.3: Pipeline Performance - Test results
a) instance blocked

Considering the execution time for tests with the same parallelism level, we observe a consistent reduction when adding instances, with one exception. The graphics 5.12, 5.13 and 5.14 illustrate the time reduction in the pipeline execution taking as a reference the value of the same parallelism level with one less instance. The reduction range from approximately 5% to almost 50%.

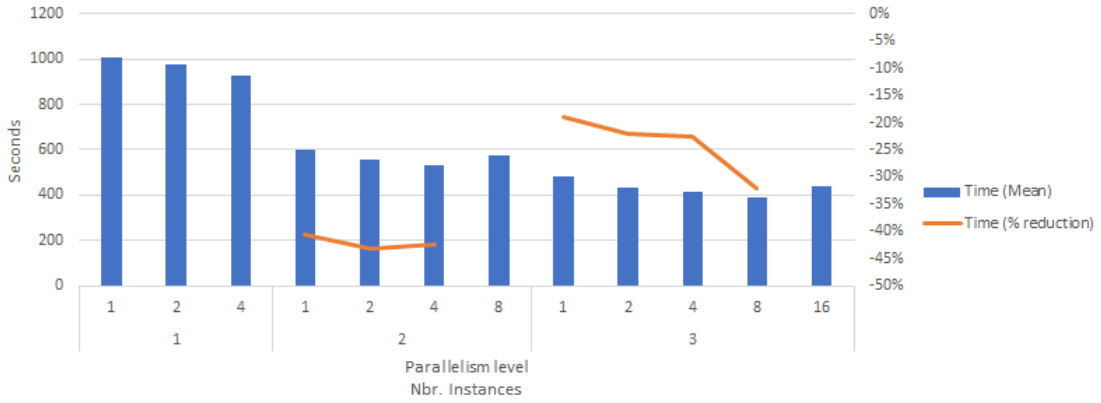


FIGURE 5.12: Pipeline Performance - 1 vCPU

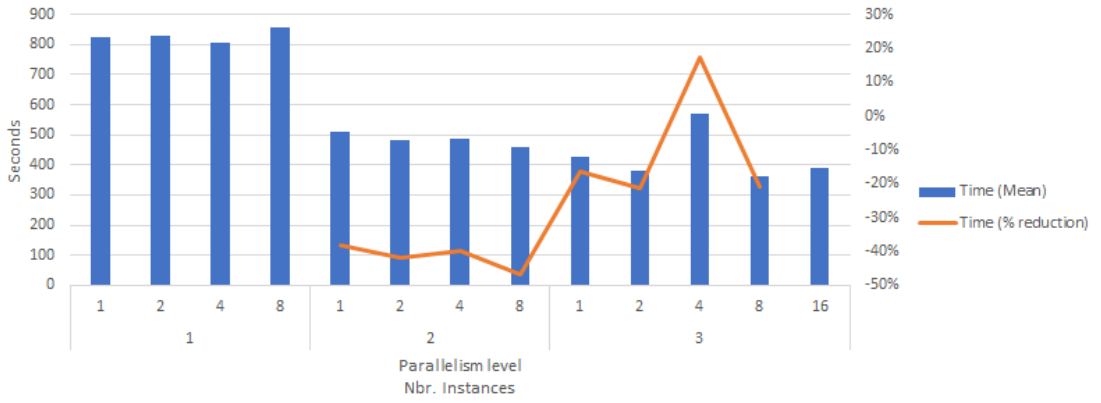


FIGURE 5.13: Pipeline Performance - 2 vCPU

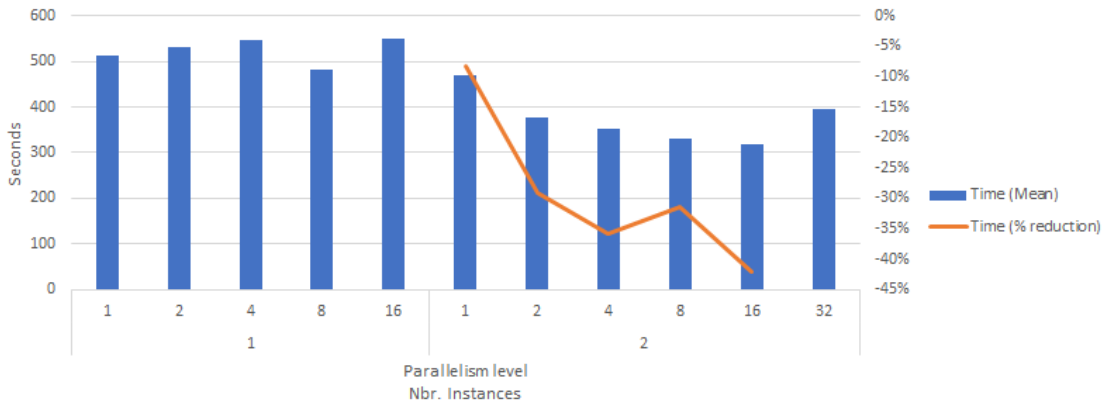


FIGURE 5.14: Pipeline Performance - 4 vCPU

Departing from the conclusion of the previous section about a differentiated approach concerning the parallelism level and the computation resources, we performed a new group of tests setting the level to 16 in *httprobe* and to 4 in *Nuclei* to run sets of 10 simultaneous pipelines, based on the results obtained in the Module's test. When testing on instances with 1 virtual CPU, we could not finish the tests successfully. Once more, some of the instances entered in a blocking state, even stopping to send the heartbeat at regular intervals, which caused the broker to assume the connection was lost.

Nbr. Instances	**1**		**2**		**3**	
Nbr. vCPU	**Time**	**StdDev**	**Time**	**StdDev**	**Time**	**StdDev**
1	*a*)	-	*a*)	-	*a*)	-
2	899,17	33,01	542,03	115,62	395,84	130,88
4	498,85	33,98	287,66	53,48	-	-

TABLE 5.4: Pipeline - Test results

a) instance blocked

Although not revealing a consistent improvement, the results on table 5.4 point to a relative success in the setups with more available resources.

Chapter 6

Conclusion

The test's results, although preliminary, point to a successful approach as far as the overall system's performance is concerned. Some points should, however, be considered:

- Network
- Computation resources
- Module startup overhead
- Task distribution

Network conditions are very relevant, mainly considering that almost all the tools that an user may want to add will interact with the target through the network. As such, a poor network connection will degrade significantly the system's performance.

We can also infer that computational resources play a significant role. Comparing *httprobe* with *Nuclei*, where the second has a processing phase of the responses to the requests, we denote that the parallelism level has to be lower in order to avoid a resource depletion.

Although not measured during these tests, we assume that there is some time elapsed between a module call and the moment it actually starts processing requests. This elapsed time may be sufficient to process a certain amount of requests, for which further testing should be done calibrate the parallelism level according to this aspect.

Concerning task distribution, the created mechanism tied to CPU usage at a certain moment in time should be improved. We realised that, sometimes, the task distribution was not as balanced as expected and, although accepting the CPU metrics were correct

at the moment they were produced, the global task distribution may have been impaired once it relies in just one element with a considerable amount of volatility.

The performed tests point to a real advantage, as far as the consumption of time is concerned, when the workload is distributed by a pool of instances. Additionally, paralleling module's execution increases the performance, although aspects like default configurations, computing resources and processing needs shall be taken in account. Despite the improvement needs identified, OrchRecon provides an efficient framework for reconnaissance activities that helps to leverage the use of tools that a researcher may choose by providing a way of orchestrating them in a pipeline approach.

6.1 Future Work

OrchRecon, at the present, is a prototype to approach a message-oriented framework that enables tool orchestration, while maintaining a perspective in performance. Our option for parallel execution seems aligned with that perspective, although some increase may be obtained if we resort to another language like Go or Rust. Moreover, some architecture options may be rethought, like performing the Module's parallel execution in distinct instances unlike the actual implementation.

Concerning the gathered data and its storage, some other approaches should be equated, namely the implementation of a data structure that allows to an easier correlation between all the collected elements and an appropriate database engine that supports it.

As far as error recovery is concerned, the actual mechanism is very incipient and shall be improved. Although the majority of the incorporated tools have their own mechanism, an effective approach must be outlined within the application logic.

As a prototype, some functionalities were not properly addressed, for which a GUI is the next logical improvement, as well as reporting capabilities.

Bibliography

[1] ENISA, "Vulnerabilities and exploits," last accessed 01/06/2020. [Online]. Available: https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/vulnerabilities-and-exploits

[2] A. Kakareka, "Chapter 31 - what is vulnerability assessment?" in *Computer and Information Security Handbook*, 3rd ed., J. R. Vacca, Ed. Boston: Morgan Kaufmann, 2013, pp. 483 – 494. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128038437000314

[3] ETSI, *ETSI TR 103 305: CYBER; Critical Security Controls for Effective Cyber Defence*, ETSI Std., 2015. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/103300_103399/103305/01.01.01_60/tr_103305v010101p.pdf

[4] *CIS Controls*, Center for Internet Security Std., 2019. [Online]. Available: https://www.cisecurity.org/controls/

[5] J. T. C. I. J. . . I. technology — Subcommittee SC 27 — IT Security techniques, *ISO/IEC 27002:2013*, ISO/IEC Std., 2013.

[6] "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec." [Online]. Available: http://data.europa.eu/eli/reg/2016/679/oj

[7] "Directive (eu) 2016/1148 of the european parliament and of the council of 6 july 2016 concerning measures for a high common level of security of network and information systems across the union." [Online]. Available: https://eur-lex.europa.eu/eli/dir/2016/1148/oj

[8] "Regulation (eu) 2019/881 of the european parliament and of the council of 17 april 2019 on enisa (the european union agency for cybersecurity) and on information and communications technology cybersecurity certification and repealing regulation (eu) no 526/2013 (cybersecurity act)." [Online]. Available: http://data.europa.eu/eli/reg/2019/881/oj

[9] ETSI, *ETSI TR 103 456: CYBER; Implementation of the Network and Information Security (NIS) Directive*, ETSI Std., 2017. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/103400_103499/103456/01.01.01_60/tr_103456v010101p.pdf

[10] "Center for internet security," last accessed: 25/05/2020. [Online]. Available: https://www.cisecurity.org/

[11] C. for Internet Security, "Mapping and compliance," last accessed 02/06/2020. [Online]. Available: https://www.cisecurity.org/cybersecurity-tools/mapping-compliance/

[12] ETSI, *ETSI TR 103 305-1: CYBER; Critical Security Controls for Effective Cyber Defence; Part 1: The Critical Security Controls*, ETSI Std., 2018. [Online]. Available: https://www.etsi.org/deliver/etsi_tr/103300_103399/10330501/03.01.01_60/tr_10330501v030101p.pdf

[13] J. T. F. T. Initiative, *NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations*, NIST Std., 2015. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final

[14] ——, *NIST Special Publication 800-53A Revision 4: Assessing Security and Privacy Controls in Federal Information Systems and Organizations*, NIST Std., 2014. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-53a/rev-4/final

[15] R. M. Elie Saad, Matteo Meucci, *OWASP Web Security Testing Guide*, owasp.org Std., 2020. [Online]. Available: https://github.com/OWASP/wstg/releases/download/v4.1/wstg-v4.1.pdf

[16] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, *NIST Special Publication 800-115: Technical Guide to Information Security Testing and Assessment*, NIST Std., 2008. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-115/final

[17] P. S. S. C. Penetration Test Guidance Special Interest Group, *PCI Data Security Standard — Information Supplement — Penetration Testing Guidance*, PCI Std., 2017.

[18] M. Barceló and P. Herzog, "The open source security testing methodology manual," Institute for Security and Open Methodologies, Tech. Rep.

[19] I. I. Amit, *The penetration testing execution standard*, Std., 2014, last accessed 11/06/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Main_Page

[20] *Penetration Test Guidance*, FedRAMP - Federal Risk and Authorization Management Program Std., 2017. [Online]. Available: https://www.fedramp.gov/assets/resources/documents/CSP_Penetration_Test_Guidance.pdf

[21] Common vulnerability scoring system. [Online]. Available: https://www.first.org/cvss/

[22] Cve - common vulnerabilities and exposures. [Online]. Available: https://cve.mitre.org/

[23] Cwe - common weakness enumeration. [Online]. Available: https://cve.mitre.org/

[24] I. I. Amit, *Intelligence Gathering - The penetration testing execution standard*, Std., 2014, last accessed 17/06/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Intelligence_Gathering

[25] J. Faircloth, "Chapter 2 - reconnaissance," in *Penetration Tester's Open Source Toolkit*, 4th ed., J. Faircloth, Ed. Boston: Syngress, 2017, pp. 31 – 106. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128021491000026

[26] Maltego. Last accessed: 22/06/2020. [Online]. Available: https://www.maltego.com/

[27] The harvester. Last accessed: 22/06/2020. [Online]. Available: https://github.com/laramies/theHarvester

[28] Netcraft. Last accessed: 22/06/2020. [Online]. Available: https://www.netcraft.com/

[29] Nmap. Last accessed: 22/06/2020. [Online]. Available: https://nmap.org/

[30] J. Faircloth, "Chapter 3 - scanning and enumeration," in *Penetration Tester's Open Source Toolkit*, 4th ed., J. Faircloth, Ed. Boston: Syngress, 2017, pp. 107 – 149. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128021491000038

[31] I. D. Ceukelaire. Abusing autoresponders and email bounces. Last accessed 17/06/2020. [Online]. Available: https://medium.com/intigriti/abusing-autoresponders-and-email-bounces-9b1995eb53c2

[32] ——. Hundreds of internal servicedesks exposed due to covid-19. Last accessed 17/06/2020. [Online]. Available: https://medium.com/@intideceukelaire/hundreds-of-internal-servicedesks-exposed-due-to-covid-19-ecd0baec87bd

[33] I. I. Amit, *Vulnerability Analysis - The penetration testing execution standard*, Std., 2014, last accessed 30/06/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Vulnerability_Analysis

[34] Owasp top ten 2017. Last accessed: 4/7/2020. [Online]. Available: https://github.com/OWASP/Top10/raw/master/2017/OWASP%20Top%2010-2017%20(en).pdf

[35] D. B. F. Deborah J. Bodeau, Catherine D. McCollum, "Cyber threat modeling: Survey, assessment, and representative framework," *The Homeland Security Systems Engineering and Development Institute*, 2018. [Online]. Available: https://www.mitre.org/sites/default/files/publications/pr_18-1174-ngci-cyber-threat-modeling.pdf

[36] I. I. Amit, *Threat Modeling - The penetration testing execution standard*, Std., 2014, last accessed 25/06/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Threat_Modeling

[37] ——, *Exploitation - The penetration testing execution standard*, Std., 2014, last accessed 09/07/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Exploitation

[38] S. A. Rahalkar, *Certified Ethical Hacker (CEH) Foundation Guide*. Springer, 2016.

[39] I. I. Amit, *Post Exploitation - The penetration testing execution standard*, Std., 2014, last accessed 26/07/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Post_Exploitation

[40] R. Messier, "Penetration testing basics," *Berkeley, CA: Apress*, 2016.

[41] I. I. Amit, *Reporting - The penetration testing execution standard*, Std., 2014, last accessed 27/07/2020. [Online]. Available: http://www.pentest-standard.org/index.php/Reporting

[42] Injection flaws. Last accessed: 13/7/2020. [Online]. Available: https://owasp.org/www-community/Injection_Flaws

[43] Summarizing known attacks on transport layer security (tls) and datagram tls (dtls). Last accessed: 19/7/2020. [Online]. Available: https://tools.ietf.org/html/rfc7457

[44] Xml external entity (xxe) processing. Last accessed: 19/7/2020. [Online]. Available: https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing

[45] Xxe injection. Last accessed: 19/7/2020. [Online]. Available: https://portswigger.net/web-security/xxe

[46] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0," W3C, RFC, 2013. [Online]. Available: https://www.w3.org/TR/REC-xml/

[47] Access control vulnerabilities and privilege escalation. Last accessed: 21/7/2020. [Online]. Available: https://portswigger.net/web-security/access-control

[48] Broken access control. Last accessed: 21/7/2020. [Online]. Available: https://owasp.org/www-community/Broken_Access_Control

[49] Code review guide. Last accessed: 21/7/2020. [Online]. Available: https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf

[50] C. Linhart, A. Klein, R. Heled, and S. Orrin. Http request smuggling. Last accessed: 22/7/2020. [Online]. Available: https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

[51] Cross-site scripting. Last accessed: 22/7/2020. [Online]. Available: https://portswigger.net/web-security/cross-site-scripting

[52] Insecure deserialization. Last accessed: 24/7/2020. [Online]. Available: https://portswigger.net/web-security/deserialization

[53] Deserialization of untrusted data. Last accessed: 24/7/2020. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data

[54] Exploiting insecure deserialization vulnerabilities. Last accessed: 24/7/2020. [Online]. Available: https://portswigger.net/web-security/deserialization/exploiting

[55] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[56] Containerization explained — ibm. Last accessed: 18/08/2020. [Online]. Available: https://www.ibm.com/cloud/learn/containerization

[57] What is a container? — app containerization — docker. Last accessed: 18/08/2020. [Online]. Available: https://www.docker.com/resources/what-container

[58] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[59] G. F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed systems: concepts and design*. Pearson Education, 2012.

[60] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 3rd ed. Pearson Education, 2018.

[61] J. T. C. I. J. . . . I. technology — Subcommittee SC 33 — Distributed application services and ITU-T, *ISO/IEC 10746-1:1998*, ISO/IEC Std., 1998.

[62] Cloud computing - statistics & facts — statista. Last accessed: 01/09/2020. [Online]. Available: https://www.statista.com/topics/1695/cloud-computing/

[63] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[64] Protocol buffers — google developers. Last accessed: 05/10/2020. [Online]. Available: https://developers.google.com/protocol-buffers/

[65] 0mq - the guide. Last accessed: 06/10/2020. [Online]. Available: http://zguide.zeromq.org/

[66] The bug hunter's methodology. Last accessed: 12/10/2020. [Online]. Available: https://www.youtube.com/watch?v=gIz_yn0Uvb8

[67] Github - jhaddix/tbhm: The bug hunter's methodology. Last accessed: 12/10/2020. [Online]. Available: https://github.com/jhaddix/tbhm

[68] Github - findomain. Last accessed: 12/10/2020. [Online]. Available: https://github.com/Findomain/Findomain

[69] Github - silverpoision/rock-on. Last accessed: 12/10/2020. [Online]. Available: https://github.com/SilverPoision/Rock-ON

[70] Github - epi052/recon-pipeline. Last accessed: 12/10/2020. [Online]. Available: https://github.com/epi052/recon-pipeline

[71] Intrigue - intelligent attack surface management. Last accessed: 12/10/2020. [Online]. Available: https://intrigue.io/

[72] Github - intrigueio/intrigue-core. Last accessed: 12/10/2020. [Online]. Available: https://github.com/intrigueio/intrigue-core

[73] Documentation - spiderfoot. Last accessed: 12/10/2020. [Online]. Available: https://www.spiderfoot.net/documentation

[74] Github - smicallef/spiderfoot. Last accessed: 12/10/2020. [Online]. Available: https://github.com/smicallef/spiderfoot

[75] Github - j3ssie/osmedeus. Last accessed: 12/10/2020. [Online]. Available: https://github.com/j3ssie/Osmedeus/

[76] What is Spear Phishing? Definitions and Risks. Last accessed: 19/10/2020. [Online]. Available: https://www.kaspersky.com/resource-center/definitions/spear-phishing

[77] K. Kritikos, K. Magoutis, M. Papoutsakis, and S. Ioannidis, "A survey on vulnerability assessment tools and databases for cloud-based web applications," *Array*, vol. 3, p. 100011, 2019.

[78] Cloud computing services — google cloud. Last accessed: 07/10/2020. [Online]. Available: https://cloud.google.com/

[79] Sqlalchemy - the database toolkit for python. Last accessed: 08/10/2020. [Online]. Available: https://www.sqlalchemy.org/

[80] Instance groups — compute engine documentation — google cloud. Last accessed: 12/10/2020. [Online]. Available: https://cloud.google.com/compute/docs/instance-groups/

[81] Acronis - bug bounty program — hackerone. Last accessed: 19/10/2020.

[82] projectdiscovery/subfinder. Last accessed: 19/10/2020. [Online]. Available: https://github.com/projectdiscovery/subfinder

[83] tomnomnom/httprobe. Last accessed: 19/10/2020. [Online]. Available: https://github.com/tomnomnom/httprobe

[84] projectdiscovery/nuclei. Last accessed: 19/10/2020. [Online]. Available: https://github.com/projectdiscovery/nuclei