POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies
Telecommunication Engineering

Master's Degree Thesis

Design and Development of a Honeypot Testing System

Supervisors	Candidate
Prof. MARCO MELLIA	ANDREA DOMENICO
Prof. LUCA VASSIO	MOURGLIA
Prof. IDILIO DRAGO	

December 2022

Summary

The permeation of connected electronic devices in everyday life has exponentially increased since the beginning of this century. Connected devices are constantly exposed to potential threats, that could lead to sensitive data leaks. Data is the gold of the twenty-first century, thus making this risk not only a security problem but a critical economical issue.

Modern IT systems are equipped with technological solutions that provide detection and protection against security risks. However, these systems do not produce comprehensive raw data about what they inspect. Complete information about the traffic toward a host could be useful to understand how malicious activities are performed. In order to carry out this task, honeypots can be employed.

Honeypots are tools intended to mimic the behavior of a real system, tricking an attacker into interacting with them and exploiting their apparent potential vulnerabilities. As honeypots are not real systems, it is important to assess how they respond with respect to an actual machine. As a matter of fact, the fidelity of the response is an important topic, because it participates in the ability of the honeypot to keep an attacker connected to it in pursuit of its goal. However, given its fictitious nature how can we assess and evaluate honeypot effectiveness?

The activity of testing systems by probing and attacking them to get information about their behavior and discover vulnerabilities is known as penetration testing. Penetration testing techniques are composed of different phases that contribute to the assessment of a system security. In this work, we followed some of the penetration testing phases such as analysis, gaining and maintaining access, to evaluate honeypot behavior with respect to real systems.

As a result, we developed a system called T-Hon capable of performing several attacks through the SSH protocol against different victims, managing the data collection and the target selection. The T-Hon system automates the attack process by controlling Metasploit, a framework for penetration testing activities, and managing the selection of the victim through a proxy, which is in charge of collecting all the traffic data and correctly forwarding the communication to the right recipient. The modular architecture of T-Hon allows to perform additions or modifications of a part of the system without affecting the others. Furthermore,

the system employs an organized and simple way of defining the parameters of the tests through a single configuration file. This feature allows to configure each attack individually while keeping all settings in one place. T-Hon collects and organizes the data in order to support a complete analysis. T-Hon has a wide number of victim scenarios as it allows to perform attacks on real machines, virtual machines, and Docker containers.

In order to evaluate T-Hon, we used Cowrie as a test subject. Cowrie is a medium interaction SSH and Telnet honeypot. The tests consisted in performing several attacks against it and real systems, thus comparing the results to discover potential issues and odd behaviors of the honeypot response. Attacks, or exploits, are scripts specifically designed to exploit vulnerabilities in order to gain access to a system. In this case, the chosen attacks aim at gaining access to the victim through the SSH protocol, retrieving a shell, and executing specific commands.

The evaluation of honeypot performance is a complex process. The presence of metrics that allow us to summarize the honeypot ability is of great importance. As a consequence, we propose a set of metrics that can be employed to recap the ability of the honeypot to mimic a real system, starting from the evidence collected through T-Hon. The analysis and comparison of the data collected by T-Hon during these attacks toward both honeypots and actual systems highlighted some important discrepancies. We identified potential issues in the Cowrie SSH connection protocol implementation, especially concerning the order in which messages are sent and how shell requests are managed. Furthermore, we were able to address one of the issues that we discovered in the previous phase. In this way, we deployed two versions of Cowrie, one with the issues solved and one without, into a real network for more than two months. In this way, exposing the two honeypots to real traffic interactions, we were able to understand how the modification we made affected the honeypot behaviors. This experiment reported some interesting results. The analysis of the recorded data highlighted how the modified honeypot performed better with respect to the other one in terms of the number of commands received in one session. Thanks to the aforementioned findings, we proved the value and usefulness of the developed tool.

Acknowledgements

Il mio percorso universitario non è stato dei più semplici. Il difficile impatto con il mondo accademico, la salute non sempre dalla mia parte e una pandemia sono stati grandi ostacoli da superare. Tuttavia, questi ultimi due anni hanno costituito una rinascita. Finalmente sono riuscito a ritrovare quell'entusiasmo e curiosità che avevano caratterizzato i miei anni all'Istituto Tecnico, un luogo che mi ha plasmato e dato una visione unica del mondo tecnologico.

Chi mi conosce sa che non amo esternare le mie emozioni e tanto meno i miei sentimenti. Questo, però, è un momento speciale, è il traguardo di un percorso, o meglio, un'importante tappa di un più grande progetto.

Nel mio percorso di studi, sono state tante le persone che mi hanno supportato ed aiutato, soprattutto nei momenti più complessi e difficili anche quando avevo l'impressione che il mondo intorno a me correva velocissimo ed io ero lì immobile, fermo sui libri. Ora è il momento di ringraziarle tutte ed in particolare vorrei spendere qualche parola in più per coloro che maggiormente hanno condiviso con me questi anni.

Ai miei nonni, ai miei zii, ai miei cugini ed in particolare ai miei genitori, con cui ho condiviso gioie e dolori di questo percorso di studi. Non lo dico spesso ma con gli anni mi sono reso conto di quanto sia stato fortunato ad avere una mamma ed un papà che hanno saputo lasciarmi scegliere la mia strada senza mai obbligarmi a prendere una via piuttosto che un'altra, ma rendendomi partecipe di un costante confronto che mi ha permesso di crescere personalmente rendendomi consapevole e autonomo.

A Chiara, la mia ragazza, che ha vissuto insieme a me, passo dopo passo, questo mio percorso. Mi ha supportato e sopportato quotidianamente con il suo entusiasmo anche quando ho fatto fatica a farlo io.

À ma famille canadienne, que même à distance, elle continue à m'être proche et que dans celle année, passée pour une bonne partie au froid, m'a fait découvrir un nouveau monde, dépassant me limites e en ouvrant mon horizon a des nouvelle expériences et projets.

All'Oratorio di Cavour e agli Amici di Babano, due realtà di volontariato che mi hanno formato e permesso di esprimere le mie passioni che hanno influenzato la

scelta del mio percorso di studi sin dalle scuole medie.

A Marco ed Alessandro, due punti di riferimento costanti con cui confrontarsi, costruire ed inventare!

Ai miei amici di sempre con cui da vent'anni condivido il mio tempo libero, con cui faccio volontariato e che mi hanno sostenuto ed aiutato quando sono stato in difficoltà, grazie GNOC!

Ai ragazzi di Smart Data con cui ho speso buona parte della stesura della tesi e che ringrazio per la loro grande disponibilità ed accoglienza. In particolare, un grazie speciale a Giulia M. per i suoi consigli e le sue guide.

Ai miei relatori I. Drago, L. Vassio, M. Mellia che mi hanno seguito costantemente in tutti questi mesi di lavoro. Un supporto formativo che mi ha guidato alla scoperta del mondo della ricerca e della cybersecurity.

A tutti coloro che hanno condiviso con me parte di questo percorso. Sono stati fondamentali nel raggiungimento di questo traguardo!

Grazie a tutti!

Table of Contents

List of Tables

List of Figures

Acronyms

CLI
Command Line Interface

VM
Virtual Machine

RPC
Remote Procedure Call

SSH
Secure Shell Protocol

OS
Operative System

IT
Information Technology

IDS
Intrusion Detection System

IPS
Intrusion Protection System

LIH
Low Interaction Honeypots

MIH
Medium Interaction Honeypots

HIH

High Interaction Honeypots

RL

Reinforcement Learning

IP

Internet Protocol

CVE

Common Vulnerabilities and Exposures

NVD

National Vulnerability Database

NIST

National Institute of Standards and Technology

CVSS

Common Vulnerability Scoring System

JSON

JavaScript Object Notation

TTY

Teletypewriter

API

Application Programming Interface

MSF

Metasploit Framework

CDF

Cumulative Density Function

Chapter 1

Introduction

Cybersecurity threats are becoming part of our daily life as the internet connection permeates every aspect of the day. Nowadays, the effects of cyber-attacks have a direct impact on the physical world, bringing them from the virtual environment of the internet into the real world [1]. The significance of this phenomenon is highlighted by the high economical cost that cyber-crime causes to companies. According to [2], the average data breach cost in 2020 was $3.86 million. This value increased remarkably year-over-year, reaching $4.24 million in 2021.

This issue could be even more critical in the next years as the number of devices connected to the internet is constantly increasing. As a matter of fact, according to Cisco Annual Internet Report (2018–2023), the number of connected devices will be three times the global population by 2023. Furthermore, over 70% of the global population will have mobile connectivity by then [3]. Consequently, as connected devices are becoming increasingly personal, they may expose to potential risks an enormous amount of sensitive data that could be used for unlawful purposes.

Cyber-threats are continuously evolving, becoming more sophisticated and tailored to the target, trying to exploit the latest vulnerabilities [4], making it even more important to maintain all electronic devices up to date with the latest security features available.

In order to increase the awareness about how attacks are deployed, it is important to collect accurate and high-quality data about how they are carried out. In this way, effective counteractions can be designed and set up. Intrusion Protection Systems (IPS), Intrusion Detection Systems (IDS), and antivirus solutions are often implemented into modern IT systems, but they lack the ability to provide security analysts with raw data about the traffic inspected by them [5], which can bring interesting insights regarding malicious activities.

The employment of honeypots can be very useful to accomplish this task, as they provide a powerful way to gather information. Honeypots are emulated virtual

machines that can act as a vulnerable one, attracting attackers to probe them. Thus, providing helpful data in a harmless way [6].

However, since honeypots are not real devices, it results particularly challenging for the developers to create a system that answers in a realistic way. This fact, leads to another important issue: how could we assess the fidelity of honeypots response? This is the main research question addressed by this work. Assessing honeypots response fidelity is significant as it brings to attention how the hoenypot is able to mimic the behavior of a real system. Moreover, this is directly connected to the ability of a honeypot to keep trapped the attacker [5]. Based on this criteria, honeypot systems can be divided into three different categories: LIH, MIH, HIH. Low Interaction Honeypots (LIH) provide little interaction with adversaries as they implement a small subset of OS functionalities. Medium Interaction Honeypots (MIH) provide a greater set of real OS features. High Interaction Honeypots (HIH) are completely functional systems that can be fully compromised by attackers [5].

In this work, we will concentrate on the second type, testing Cowrie, an open-source medium interaction SSH and Telnet honeypot [7]. SSH and Telnet are communication protocols widely used to access remote hosts. Nowadays, SSH higher security makes it the preferred one for that task, as it employs public-key cryptography to authenticate the client.

In order to evaluate honeypots behavior, it is necessary to compare their responses with those of a real system. This operation is based on the data collected from several attacks launched against honeypots and real systems. By analyzing the collected information it is possible to see how the honeypot responds to the attacker with respect to the behavior of a real system.

The analysis of data coming from attack execution is part of a security technique known as penetration testing. Penetration testing aims at providing proactive security protection highlighting security issues [8]. Generally, a penetration test is composed of five different phases: planning and reconnaissance, scanning, gaining access, maintaining access, and analysis [9]. For the scope of this work, we will concentrate on gaining and maintaining access, since we are not looking for potential vulnerabilities of the system under evaluation, but how it responds to different attacks. However, the first task of planning and reconnaissance still covers an important role. As a matter of fact, the search for attacks that are suitable for this task could be a challenging work since they have to be designed for the penetration testing framework and for the communication protocol in use.

As a result of this work, we designed a system called T-Hon that automates the attack process towards several victims, managing the log collection and the victim selection. Logs are collected and processed to make them readable and easier

to analyze. T-Hon handles the Metasploit framework to select and configure the attack. It manages the selection of the recipient by properly configuring a proxy which is in charge of collecting all SSH traffic between the attacker and the victim and forwarding it to the right recipient. T-Hon is based on a modular architecture composed of four blocks: attacker, proxy, victims, and manager. The attacker is the part in charge of deploying the attack to the victim. The proxy is responsible for collecting the traffic between the attacker and correctly forwarding the attack to the right victim. The victims are the recipients of the attack and they can be a real machine, a virtual machine, or even a Docker application. The last block is the manager, which is in charge of coordinating all the other blocks by selecting the victim and the attack to perform. It allows the user to define in advance all the settings for each attack and victim, without the need for user intervention during the test process. Furthermore, it organizes the data collection by grouping the gathered information in a structured way, simplifying the analysis process. The system project is open source and will be available at [10].

In order to evaluate how the honeypot performed with respect to a real system, in terms of how it is able to mimic the actual behavior of a true machine, we proposed some metrics that constitute a useful tool to summarize the honeypot performance. Thanks to T-Hon we were able to collect important data that allowed us to discover several issues in Cowrie SSH connection handling, shell gathering, and command execution. Based on these findings we assessed Cowrie performance through the proposed metrics. The found problems could prevent the attacker from completing the attack, or compromise its ability to verify its status. In order to understand how much such issues could affect the honeypot effectiveness, we addressed one of them by applying some modifications to the honeypot software. Then, we deployed two honeypots: one with the fixes and the other without. We collected about two months of data. The analysis of the gathered information offered some insightful results. The data collected by the fixed honeypot showed a higher number of commands per session and a reduction in the repeated attempts of executing certain commands. In this way, we were able to understand why the original honeypot received an important amount of reiterated commands. As a consequence, we proved that T-Hon is a valuable tool to gather data to assess honeypots work.

Chapter 2

Objectives and related work

The objective of this work is to create a system that allows information gathering about attacks towards honeypots and real systems. In particular, the system has to allow the automation of the attack process to perform several attacks without manual intervention. It should automate the management of data collection and victim selection. The gathered data will be of great value to evaluate the honeypot response compared to the ones of real systems. In this section we describe what are the objectives of our work, introducing the reasons that led to the creation of this work.

2.1 Objectives and research context

Honeypots are fictitious computer systems that aim at luring attackers into interacting with them. Honeypots are largely employed to study network security. The idea behind their use is to trap attackers and collect data about what actions they perform and avoid their interaction with production machines that could cause important damage and resource wasting [11].

As a consequence, the way honeypots respond to the attacker is a key feature of their effectiveness. Honeypots are not real computers, hence their answer could be different from the one of an actual system. Depending on how the client interacts with them, how complex is the operation required and how advanced is the honeypot, it may not be able to satisfy such requests. An unexpected request that could potentially lead to a faulty response may compromise the honeypot disguise resulting in being discovered [11].

Honeypot technology is very useful in understanding how malicious activities are carried out, however, it is important to understand how well these systems are able to mimic the real system behavior. The data collected by honeypots may be justified also by understanding how they responded to the attack, thus observing

the victim from the attacker point of view. The way honeypot responds could influence the attacker activity having consequences on the reality of its actions. Attackers may employ automated techniques to perform malicious activity on a large scale. Attacks are generally structured in steps. Each step of the attack carries out a certain action that leads to the prosecution of the offensive. If one of these actions does not receive positive feedback, the attack could be compromised and the data that could result may be corrupted or incomplete, thus making it difficult to understand.

Generally, researchers use honeypots to study SSH-based attacks [12], nevertheless there are honeypots designed specifically for different protocols and purposes. The Secure Shell protocol allows secure access to remote hosts and network appliances [13]. It is extensively employed in production environments [13], making it a common target for malicious activity attempts since it can grant access to a machine command line interface [14]. As a consequence, in this work, we will concentrate on the SSH protocol.

The SSH protocol is composed of three different layers, one running on top of the other, and they are listed as follows from the lower to the higher: transport, user authentication, and connection. The first one provides server authentication, confidentiality, and integrity and it generally runs over a TCP/IP connection. The second one is in charge of providing authentication to the client user. The third one allows the creation of logical channels that support functionalities such as interactive login sessions and remote command execution [15]. The third layer is of great interest, since, provides the feature of running commands on the remote host, and allows attackers to interact with the victim machine, upon successful authentication.

Data coming from the attacker is particularly useful in all security-related data analysis activities. However, in order to evaluate the response of a honeypot, it is necessary to perform comparisons between its behavior and the one of a real system under the same conditions. This means that it is essential to analyze data from the attacker point of view, by performing attacks toward both real systems and honeypots to understand how they act according to the received malicious commands. Based on these considerations, the need for a honeypot testing system came up.

The testing system should be able to automatically perform multiple tests toward real and mock systems, recording useful data for later analysis and comparisons. The analysis of the recorded data should be focused on highlighting differences among the responses of the victims. In particular, the analyst should look for indicators of honeypot fingerprints, which would lead to the distinction between a real system and a mock one. This kind of analysis enables the possibility to give the honeypot under testing an evaluation based on important parameters

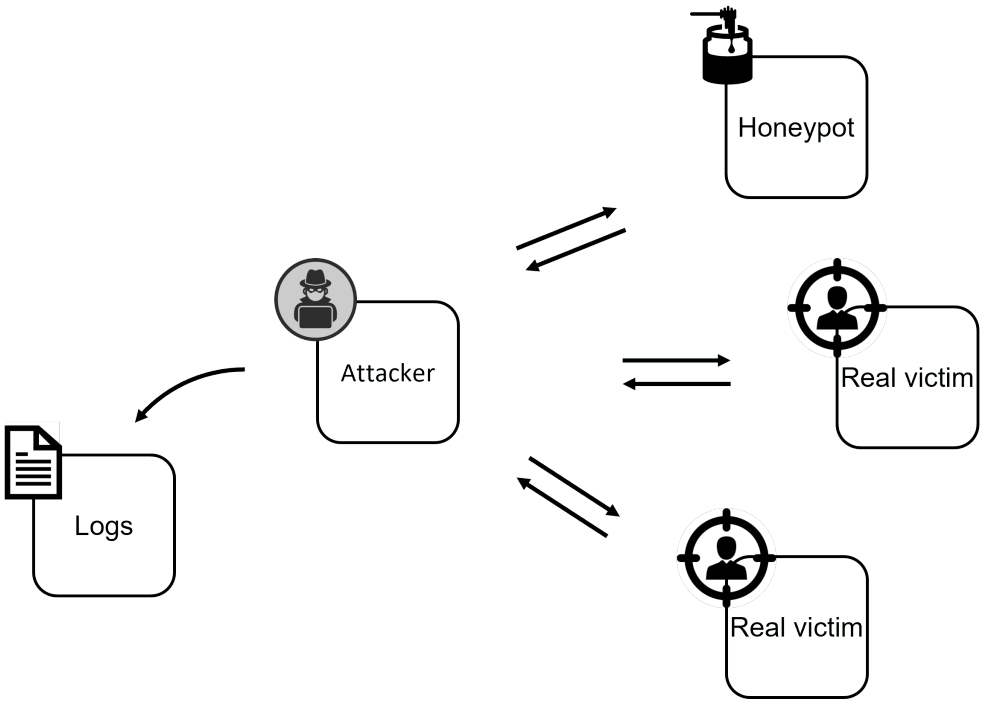

Figure 2.1: Honeypot testing scenario: an attack is deployed to multiple victims, honeypots, and real systems, in order to compare the results to find possible differences

and observation. For this reason, it is important to define metrics and evaluation methods that can generalize the assessment of a honeypot behavior.

Generally, the performance of a system can be defined as the ability of the system to meet specific user requirements and can be described as a Measure of Effectiveness (MoE) [16]. In this work, we will propose a set of metrics and parameters focused on assessing the fidelity of the honeypot response and how this relates to the behavior of the attacker.

Several penetration tools are available for testing systems vulnerabilities, however, the level of detail and the completeness of the provided data for the scope of this work still remain an unmet requirement. As a consequence, summarizing what is exposed in this paragraph and graphically reported in Figure 2.1, the ultimate goal of this work is to develop a honeypot testing system that provides useful data from the attacker perspective, that supports analysts in assessing the performance of the honeypot under investigation following specific evaluation metrics.

2.2 Related work

While there are extensive research and publications regarding the analysis of the data collected by honeypots, very few works are focused on the test and quality evaluation of such systems based on the comparison with real ones. Jason M. Pittman et al. [17] proposed an assessing tool used to reply to captured network traffic toward the honeypot with the goal of generating interactions and performing a measurement of effectiveness based on the metrics defined in [16]. The data analysis performed in [17] consists in comparing the honeypot response in the case of interaction with the traffic coming from the internet and the one generated by their developed tool.

Alexander Vetterl et al. [18] proposed a method to systematically fingerprint honeypot systems. This work covers fingerprinting techniques for three different protocols: HTTP, Telnet, and SSH. For what concerns the last one, they analyzed two different honeypots: Cowrie and Kippo. They employed specifically designed version strings and SSH2_MSG_KEXINIT packets to identify possible protocol deviation [19]. In [18] they highlighted how Kippo and Cowrie were able to respond similarly to OpenSSH, a popular implementation of the SSH protocol, however not exactly alike.

Evaluation metrics are important tools to state how effectively a honeypot performs in carrying out its task. In [16] a set of four metrics is proposed. The metrics are fingerprinting, data capture, deception, and intelligence. These metrics contribute to the definition of another parameter named Measure of Effectiveness (MoE). This parameter aims at stating how the honeypot performance complies with the requirements that were set by the final user.

In [5] a set of honeypot features that may influence honeypot performances are identified as follows: fidelity, scalability, adaptability, deployment strategy, resource type, monitoring, detection, and profiling. Despite, these features do not represent real metrics, they could constitute an interesting set of parameters to take into consideration to produce performance metrics.

Chapter 3

Background

In section 2.1 we presented the objectives of this work. In this section, we report the background information useful to understand and reach the presented goal.

In section 3.1 we report a background on honeypots paying particular attention to the different typologies and use cases. In section 3.2, we provide a background on penetration testing activity and techniques, presenting some of the most popular tools employed in the field.

3.1 Honeypots

Honeypots are relatively new devices that were first proposed at the beginning of this century. They were conceived as a tool to investigate hacker's activities and share insights about the analyzed information [20]. In these years, multiple definitions of what a honeypot is have been provided. Some are more focused on the use and application others on its structure. In [5] the authors define a honeypot as follows "A honeypot is an information system that includes two essential elements, decoys, and captors. It aims at using its information resources to attract unauthorized and illicit access with the purpose of security investigation".

This definition gives a great insight into the structure of a honeypot presenting two main components: decoy and captor. The decoy acts as bait to attract the attacker into probing and interacting with the honeypot. On the other hand, the captor is in charge of security-related activities. It is the element responsible for the collection of attack information such as the set of commands sent by an adversary during an attack [5].

Honeypots can be classified into different categories based on several characteristics. In [6], they provide a classification based on seven parameters: purpose, role, level of interaction, scalability, resource level, source code availability, and

applications. Nevertheless, the most commonly used classification is based on the purpose and level of interaction [21].

The purpose of a honeypot defines the ultimate goal of its use. Based on this feature, honeypots are divided into research and production. The first type is generally employed for data gathering while the second one is used for security defense purposes such as preventing an adversary from compromising a real system [6]. As a matter of fact, honeypots can be integrated into a production system in order to deceive an adversary into interacting with them and making it lose time and power while proper countermeasures are prepared and deployed [5].

On the other hand, research honeypots are very valuable systems as the data they collect can be used to analyze how malicious activities are performed. Large deployments of these devices can help monitor the network activities on a global scale [20].

The definition that we reported at the beginning of this section addresses the security investigation as the main use of honeypot capabilities. However, honeypots can be a proactive component of network security systems.

Production honeypots can be integrated into already deployed security systems to increase attack detection and reaction capabilities, whereas, from the prevention point of view, the honeypot is limited to slow down the attacker [21].

For what concerns the attack detection task, honeypots can complement the work of Intrusion Prevention Systems (IPS) and Intrusion Detection Systems (IDS). These security tools can report several false-positive or false-negative alerts, while honeypots consider all traffic as malicious, providing useful insights even in the presence of an unknown type of attack. Regarding the reaction task, honeypots can help security analysts understand how the attack was deployed and retrace the steps that led to such issues [21].

The way in which honeypots are able to relate with an adversary and provide it with correct information defines the level of interaction. This parameter allows a general classification of honeypots in four categories [6][5]:

- *Low Interaction Honeypots*: provide limited interaction with the attacker. They emulate a limited amount of services with very simple functionalities. This characteristic makes them easier to be uncovered by attackers.
- *Medium Interaction Honeypots*: provide a higher level of interaction with respect to LIH, however different implementations could provide a real operating system or an emulated one and an increased number of available services.
- *High Interaction Honeypots*: provide a high level of interaction thanks to the presence of a real operating system. This feature increases the complexity of the system since it could be completely compromised by an attacker.

- *Hybrid system*: composed of different honeypots providing distinct levels of interaction, merging the benefits of the listed above systems.

The level of interaction is somehow related to the fidelity of the honeypot as the higher the interaction level the higher the fidelity of the response [5]. The honeypot fidelity is of great interest for this work since its goal is to provide a system that could support the evaluation of this feature by providing analysts with comprehensive data.

One of the main challenges for a honeypot is to prevent its detection. From the moment an adversary detects the honeypot, all the benefits of such a system vanish since the attacker will leave the system without completing its task. The level of interaction of a honeypot is linked to its ability to be undetected. Depending on the limitations of the emulated services, it is easier for an adversary to understand if the system to which is connected is not a real one [21].

Creating honeypots with anti-detection features requires specific shrewdness because emulated services could behave slightly differently from real ones. For example, delays represent a critical point for honeypots, since the honeypot nature of capturing data could increase the time that would normally be required by a real system to perform a certain task. Intelligent honeypots based on machine learning approaches represent a good point in increasing honeypot stealthiness. For example, reinforcement learning honeypots can learn from previous attackers how to answer leading to an adaptation of the response based on the previous interactions [22].

As many honeypots are open-source tools, the source code is publicly available. Despite the advantage of community collaboration and contribution, they are subject to the regular analysis of adversaries who constantly look for signatures that could allow the identification of the honeypot. In the past years, several signatures were found, due to faulty honeypot behaviors. For example, an implementation error in Kippo, a popular SSH honeypot, allowed the creation of scripts capable of sending corrupted data that generated error messages that were not compliant with the protocol standard [1].

The Secure Shell protocol is widely used to access remote systems [13]. Because of this, it is a desirable target for adversaries, since a successful attack could grant access to a system being able to potentially compromise it. If on one hand, SSH is a common target for attackers, on the other hand, it is an important topic for security researchers. As a matter of fact, the analysis of SSH protocols is subject to intensive research [14]. This kind of activity is a perfect use case for SSH honeypots.

[1] https://www.rapid7.com/db/modules/auxiliary/scanner/ssh/detect_kippo/

Cowrie is a medium to high interaction SSH and Telnet honeypot and can be used as the base software for dynamic and intelligent honeypots. Cowrie was forked from the Kippo project representing, today, its modern evolution [6]. It was developed using Python programming language which allows the emulation of a UNIX system, providing a fictitious file system enabling document uploading and visualization. In addition, Cowrie can be employed in proxy mode enabling the possibility to monitor the activity of a malicious user on another system, increasing its level of interaction [7]. It supports full Docker deployment, simplifying its distribution [21].

3.2 Penetration testing

The penetration testing activity is an important task in the information security domain. The National Institute of Standards and Technologies defines in [23] this activity as "A test methodology in which assessors, typically working under specific constraints, attempt to circumvent or defeat the security features of a system".

The goal of this task is to evaluate a system in terms of risks linked to potential security breaches. A vulnerability assessment is an activity oriented to the discovery, classification, and analysis of system issues that could constitute a risk from a cybersecurity point of view. In penetration testing, this is only a component of the entire activity, since the discovered vulnerabilities can be used against the system under evaluation to understand what piece of information is possible to obtain from it and evaluate what is the cost for the test subject to be stolen of such information [24].

Vulnerabilities are subject of great interest and research for companies and government institutions. Vulnerabilities are identified through a code defined by the Common Vulnerabilities and Exposures [25]. Furthermore, the severity of vulnerability can be assessed following the Common Vulnerability Scoring System, which provides a set of rules to associate a qualitative score to this parameter. The CVSS classifies vulnerability severity from 0.0 to 10.0 [2], where 0.0 correspond to no vulnerability and 10.0 to a critical vulnerability. The creation of a piece of software that could exploit a particular issue could lead to the potential exposure of sensitive data or even worse consequences such as infrastructure impairment and financial losses.

As a matter of fact, companies and organizations may use penetration testing techniques to periodically assess the security of their information systems. Furthermore, they can take advantage of the gained information not only from a

[2]https://nvd.nist.gov/vuln-metrics/cvss

security and privacy point of view but as a means to increase the system awareness and understanding. The data collected during penetration testing is important to characterize the effectiveness of the deployed security measures and provide useful information to the entities responsible for the security maintenance [23].

Penetration testing is a structured activity. Several standards organizations provide guidelines and methodologies to perform comprehensive tests to guarantee the comparability and reproducibility of the results and ensure the collection of meaningful and high-quality data. An example of a standard is the Penetration Testing Methodologies and Standards (PTES). It reports the procedures that should be employed to carry out a penetration test [26].

Nevertheless, penetration testing involves some general steps that are common to all standards. According to [9], the general phases of a penetration test are:

1. Planning and reconnaissance
2. Scanning
3. Gaining access
4. Maintaining access
5. Analysis

The first phase is actually composed of two parts. The first part consists in devising and organizing the test including stating what are the objectives and the methodology. The second part consists in finding all useful intelligence that can help the assessors in setting up the problem.

The second phase is based on gathering information from the system under analysis by studying system-related data, to understand and discover system vulnerabilities [9]. This step can be performed actively or passively. In the first case, the assessors directly interact with the test subject to observe its reaction to certain probes. In the second case, data that can be obtained without interaction with the victim is analyzed [26].

In the third phase, penetration testers take advantage of the previously discovered vulnerabilities in order to gain access to the target system [9]. This phase is dependent on the previous one because it shows how the discovered vulnerabilities are effective to access the system [26].

While in the previous phase the objective was to exploit vulnerabilities to gain access to the test subject, in the fourth phase the goal is to remain present in the victim system for a certain period of time to perform further exploitation in the stealthier way possible. Generally, the goal is to create permanent access to the victim.

In the last phase, the data and the results obtained through the previous steps are analyzed. In this phase, a detailed report has to be produced to provide useful information to the relevant bodies. The final report has to include precise information about the discovered vulnerabilities, describing the methodologies that were employed to guarantee the repeatability of the tests [9].

In the end, it is crucial that penetration testers approach the system as a real attacker would and they have to define in a clear and exhaustive way the objectives of the penetration test paying particular attention to the test environment, attack surface, and goal, level of effort, and threat sources. Furthermore, a valid penetration test needs to provide proof and an explanation of the findings to quantify how the discovered vulnerability constitutes a risk for the test subject [23].

3.2.1 Penetration testing tools

As introduced in section 3.2, penetration testing is a very complex activity. As a consequence, it can be supported by specialized tools that are specifically designed to allow assessors to manage attacks, victims, and information gathering. In this section, we present different tools that are useful to perform and understand penetration testing.

Kali Linux

Information security activities require specific tools. In order to allow analysts to have a complete suite of instruments, specific operative systems have been developed. For what concerns penetration testing and forensic analysis, Kali Linux is the most popular OS. This is mainly due to the fact that it is Linux-based which guarantees stability and it comes with a preinstalled comprehensive set of security tools [27]. Among the tools that are preinstalled is worth mentioning Metasploit, which will be presented in section 3.2.1.

The Metasploit Framework

Penetration testers often employ frameworks that guide them into the testing process. One of the most used penetration testing frameworks is Metasploit [27]. Metasploit is a free and open-source project maintained by Rapid7. It is written in Ruby following a modular architecture. The framework is composed of the MSF Core library which extends the Ruby Rex library. The MSF Core library is then extended by the MSF Base library that supports the user interface. In order to interact with the Metasploit framework, a Command Line Interface and a

Web interface are provided. However, the CLI is preferable since it allows to take advantage of all framework available features [27].

Here we address some of the most important Metasploit features reported in [28] and [29]. The Metasploit framework is based on modules. Modules are tools that can be employed to carry out the tasks of a penetration testing procedure that we reported in section 3.2. For what concern the first two phases, the Metasploit framework provides auxiliary modules that are specifically conceived to support information gathering and vulnerability assessment tasks. For example, vulnerability scanners and service reconnaissance are available. These modules can provide useful information about what services are running and what ports they are using. In some cases, there are specific auxiliary modules conceived to discover what version of the service is installed on the victim machine. Others, such as the *Juniper SSH Backdoor Scanner* which assesses if that particular vulnerability is present on a certain host.

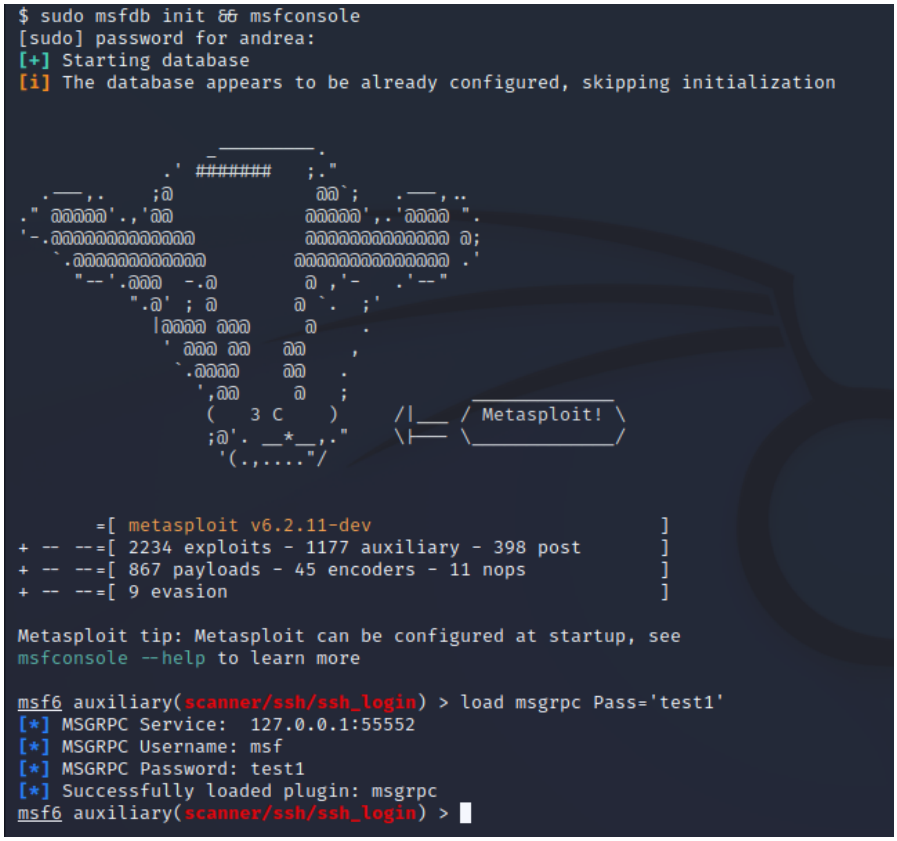

Figure 3.1: Metasploit Framework command line interface running the RPC API.

For the third phase, a complete library of modules called exploits is available.

Exploits are Ruby scripts that take advantage of known vulnerabilities to access a system. Once analysts access the system, it is possible to perform post-exploitation tasks and use techniques that will guarantee maintained access to the system, covered by the fourth phase of the penetration test procedure. For these tasks, Metasploit provides a set of payloads that help assessors in continuing the exploitation by acting on the exploited machine to achieve further objectives such as privilege escalation and backdoor installation.

An example of the payload is Meterpreter. Meterpreter is deployed through in-memory DLL injection and it allows to run specific commands aimed at further exploring the exploited machine. Furthermore, running completely in memory, it is stealthier compared to purpose-specific payloads. Another type of payload, which is often the default one, is the reverse shell. This payload allows to obtain a shell on the victim machine but exchanging the usual client/server paradigm, thus making the victim machine the client and the attacker the server. Since outgoing connections are less subject to firewall rules, it has more probability to succeed [27].

Metasploit provides a way to interface with it in order to externally automate its functionalities, by exposing a set of Application Programming Interfaces. The free version provides an RPC API that allows any other programming language able to manage HTTP-based Remote Procedure Call services such as Python. This API allows controlling Metasploit tasks externally with respect to the framework. Metasploit can run an RPC server to which it is possible to communicate following the API-specific format [29].

Metasploitable

Finding a machine that allows practicing and testing penetration testing procedures can be a challenging task. Metasploitable is a Virtual Machine that was designed specifically with a high number of vulnerabilities. This provides a secure environment in which to test attacks. Metasploitable is an Ubuntu Linux vulnerable machine. It can be seen as a test bench where it is possible to demonstrate common vulnerabilities. For instance, the employed passwords are very easy to guess, mainly default and commonly used passwords, and there are several services that provide back doors which can be exploited through Metasploit modules [29].

Chapter 4

T-Hon: testing honeypots

In this chapter, we analyze how the T-Hon system has been designed. The primary goal of this work is to create a tool that is able to automatically perform several attacks through the SSH protocol against honeypots and real systems, to obtain meaningful data concerning how honeypots and real systems respond. In this work we will focus on the SSH protocol, in particular, we will analyze the SSH connection protocol.

4.1 System requirements

In this section, we report the requirements of the system needed to achieve the objectives of this work reported in section 2.1. They can be summarized as follows:

- Automatic attack execution: attacks need to be executed without user intervention. Setup must be performed before attack execution and no on-the-fly modification should be necessary.
- Adaptability: the system must allow the user to set up all parameters of the attacks in a simple and organized way, allowing the employment of specific settings for each of them.
- Modularity: the system must allow and support any future integration or modification.
- Attack activity recording: the system has to collect, store and organize all the data generated by the attacks for the SSH connection protocol. The recorded data must be organized such that it should be easy to be processed and analyzed.
- Victim management: the system must be able to autonomously manage and select the victim to attack.

The recorded data about the attack activity can be later processed and analyzed making it possible to identify potential issues that could lead to incorrect behavior of the honeypots, thus making the attacker leave the system.

4.2 The architecture

In order to be compliant with the modularity requirement, the system architecture is based on four main components:

1. Attacker: it is in charge of deploying the attack to the victims
2. Proxy: it intercepts all the communications between the attacker and the victim recording them in a log file
3. Victims: they are the recipients of the attack
4. Manager: it is responsible for the management of the previously listed components

The graphical representation of the high-level architecture described above is reported in Figure 4.1. The manager oversees the activity of all the other components, while the proxy is placed in the middle between the attacker and the victim.

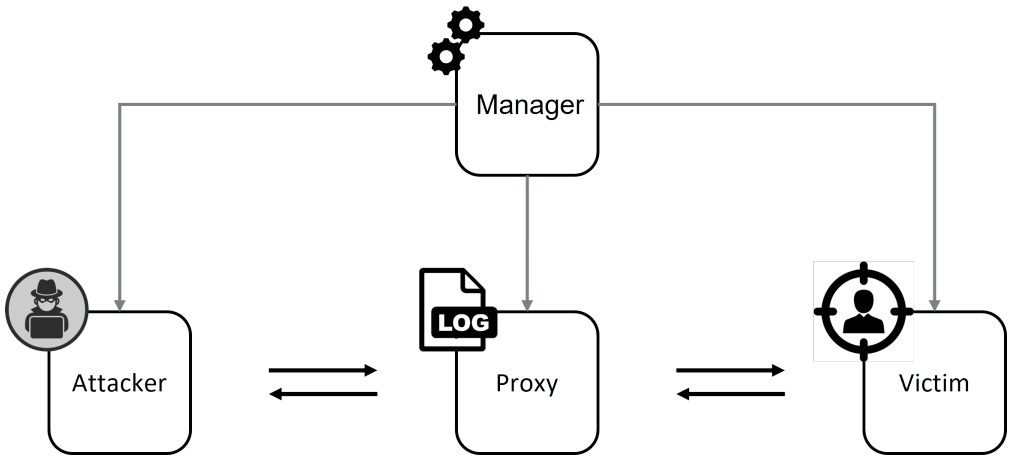

Figure 4.1: High level system architecture

4.2.1 The attacker

The attacker is in charge of deploying the attack to the victim. As represented in Figure 4.1, the attacker is not directly connected to the victim. However, from

its point of view, nothing changes. The attacker is implemented through the Metasploit framework, which was previously introduced in subsection 3.2.1.

The Metasploit framework provides a CLI that can be used to interact with it. Nevertheless, when it comes to penetration test automation, it is difficult to use it, as it is conceived for live interaction. However, in addition to the CLI, Metasploit provides an RPC API which can be used to control the attack process, hence making it possible to interact directly with the framework using scripting languages that are capable of communicating with an RPC server as a client. The RPC server can be easily started through the MSF CLI command reported in Figure 4.1.

```
1 $ load msgrpc Pass='password'
```

Figure 4.1: CLI command to start MSF RPC API

The Metasploit Framework is a key tool in the penetration testing domain. Alternatives to this tool are Burp and Core Impact. We decided to use Metasploit because it is free, it has a comprehensive library of exploit scripts and it exposes services that support attack automation. On the other hand, Burp and Core Impact free and trial versions are not as complete as Metasploit [30] [31] [29], hence they could not be easily integrated into an attack automation system. Furthermore, Metasploit comes with a complete database of exploit scripts, based on known vulnerabilities, covering a wide spectrum of operative systems, devices, and protocols.

The Metasploit framework is installed on a VM running Kali Linux. Kali Linux is an operative system conceived specifically for cybersecurity purposes. It embeds several security tools providing a comprehensive suite for penetration testing and other cybersecurity activities[1]. Metasploit is one of these tools.

Attack deployment

From the Metasploit point of view, the attack is a script written in Ruby, which is a general-purpose object-oriented programming language. The attack script takes advantage of the Metasploit framework to perform its task. As a matter of fact, Metasploit provides several modules that allow the exploit script to interact with communication protocols, operative systems, etc.

In our case, we will focus our test cases on the SSH protocol, hence the employed attack scripts are all based on known SSH vulnerabilities that aim at gaining access to a shell and executing specific commands to achieve a certain goal. For example, some scripts execute commands that allow the exploitation of specific

[1] https://www.kali.org/

vulnerabilities that would lead to a privileges escalation. The attack is carried out in three phases listed as follows:

1. Exploit selection: the attack based on a certain known vulnerability is selected.
2. Exploit configuration: parameters of the exploit such as victim port, username, and password are set.
3. Exploitation: the attack is deployed to the victim.

Among the three phases, the third one is the most important, since it physically carries out the attack. It is divided into multiple parts. In Figure 4.2 the phases of common attack exploitation performed on the SSH protocol are reported. These steps were identified by analyzing the attacks reported in Appendix A, however as reported in chapter 5 these phases may change in the order they are executed or they may not be all carried out depending on the attack goal.

Figure 4.2: General chain of operations of common Metasploit attack exploitation

Once the exploitation has ended, Metasploit will provide information about the success of the attack. Generally, the accomplishment of the attack is linked to the ability to carry out successfully all the phases reported in Figure 4.2 and listed as follows:

1. Connection opening: the SSH connection is opened with the victim machine.
2. Login: through the provided credentials or SSH key, the system executes the login and verifies its result.
3. Shell test: Metasploit can automatically check if it was able to obtain a shell in multiple ways. It can send to the victim the command "echo" followed by a set of random characters. If the shell was correctly obtained, the victim machine will respond exactly with the random set of characters that were generated by Metasploit. Otherwise, Metasploit will give negative feedback about the goal of gaining access to the shell. It can check for the SSH message stating the success of the request or it can execute other specific commands focused on the gathering of system information.
4. Command execution: in some cases, specific commands are sent after the shell has been obtained depending on the purpose of the attacks. Indeed, known

vulnerabilities of certain devices, such as routers, networking management systems, and networking OS, can be exploited to perform privilege escalation and obtain a shell with root privileges. This kind of operation can be particularly useful when the attacker is trying to install scripts or some piece of software on the victim machine that requires specific permissions.

5. Connection closing: the SSH connection is closed. However, if a shell is successfully obtained, a session is created maintaining a connection with the victim machine. The session can be either a shell or both a shell and a Meterpreter session. Meterpreter is a payload delivered to the victim machine that gives the ability to launch Metasploit modules which are not available in a standard shell[2].

The victim response is verified by Metasploit at each step to understand if the exploit gave the expected result. Metasploit provides several options that allow further information gathering about the attack. In particular, it is possible, concerning the SSH protocol, to see from the CLI a verbose log of all the SSH connection processes. However, not all attacks perform all the aforementioned tasks. For example, in some cases, depending on the attack goal, the *command execution* phase does not occur.

4.2.2 The proxy

The proxy is placed between the attacker and the victim. It is in charge of collecting logs about the traffic that passes through it and properly relaying the traffic from the attacker to the victim and vice versa.

The choice of using a proxy allows to make the log collection completely independent from the other parts of the system. In this way, the attacker and the victims can be changed, but the information-gathering block will always be the same, assuring comparability between different tests.

The proxy is implemented through Cowrie in proxy mode. Cowrie can be configured to act as an SSH proxy. This is possible because the proxy mode is completely independent from the honeypot one. Cowrie as proxy allows setting through its configuration file *cowrie.cfg* the correct victim to forward the incoming traffic. The proxy collects data at the connection level, recording all messages sent to the victim from the attacker and vice versa.

As reported in subsection 4.2.1, the attacker does not notice the presence of the proxy. However, it connects always to the proxy using its credentials, and then the

[2]https://docs.rapid7.com/metasploit/manage-meterpreter-and-shell-sessions/

proxy uses the victim credentials to connect to the right one in the back end. In this way, the attack configuration, in terms of credentials and recipient is always the same.

The proxy provides three types of log files:

1. TTY: binary file that records the interaction through SSH with the remote terminal. They are useful to reproduce the exact sequence of commands that we could see on the terminal.

2. LOG: text file where for each line an activity is recorded. It is a verbose log file, it contains extensive information about the messages exchanged in both directions (from the attacker to the proxy and vice-versa) and concerning the activity of the proxy system. Thanks to its high verbosity, it results easier to read for the human eye. However, it requires a considerable amount of processing.

3. JSON: text file formatted following the JSON standard. It is a list of objects in which information about the proxy activity and the messages received from the attacker are recorded. Each object is composed of distinct keys, each one containing a different field of a record.

Despite the JSON log file would be easier to process due to the fact that all fields of interest are already divided and directly accessible, we decided to analyze the LOG file. In this way, a thorough analysis can be performed, since we will have access to all the messages exchanged by the parties, including raw SSH payloads. This will be of great value during the analysis phase, because it allows us to understand the order in which messages are sent and what they contain, not only in terms of commands but also in terms of SSH protocol payload byte sequences.

The proxy runs in a Docker container. A container is a standard unit of software that allows running an application on different computer environments since it contains all its dependencies [3]. Thanks to this choice, the proxy can be run on any system able to execute the Docker engine, hence making it completely independent from the environment it is deployed. Furthermore, it opens the opportunity of using docker containers as victims, thus simplifying networking management and communication between the parties. Several Docker containers can be managed by Docker Compose, which is a tool that allows managing multi-container Docker applications[4].

There were two alternatives to this solution. The first one would have been to install the proxy natively on the same system of the manager. However, in

[3]https://www.docker.com/resources/what-container/

[4]https://docs.docker.com/compose/

that case, using victims running on the Docker container would have been more complicated, since the networking configuration should have exposed all the victims outside the docker environment. The second alternative would have been to install the proxy on another virtual machine, different from the one running the manager. Nonetheless, this solution would have made difficult the management of the proxy in terms of configuration files and the process of switching on and off the subsystem would have been heavier.

4.2.3 The victims

The victims are the recipients of the attack. They can be either honeypots or real systems. The basic setup should be made of at least one honeypot and one real system, to perform a meaningful comparison.

Victims can be either virtual machines or Docker containers, as mentioned in subsection 4.2.2. There is no limit to the number of victims that can be used. For each victim, a configuration file in the proxy has to be created. It contains all the necessary information that the proxy needs to correctly forward the attack:

- Port on which the SSH service is running
- IP address or Docker service name
- Username to perform SSH login
- Password to perform SSH login

These configuration files will be automatically selected by the manager to switch the victim of the test.

4.2.4 The manager

Given the complexity of the architecture and the number of actors involved, it is necessary to have some kind of supervisor that oversees and controls the testing process. The subsystem in charge of this task is the manager. The manager is a Python program that automates and orchestrates the testing process. This part is of great importance since it organizes how attacks are carried out, the victim selection, and the log collection. In Figure 4.3 a detailed representation of how the parts of the system interact among them. In this figure, we can observe the coordinating action performed by the manager subsystem on all the other parts.

Manager settings

The manager is set up through a configuration file written following the JSON standard. Configuration files are useful tools since they allow to modify parameters

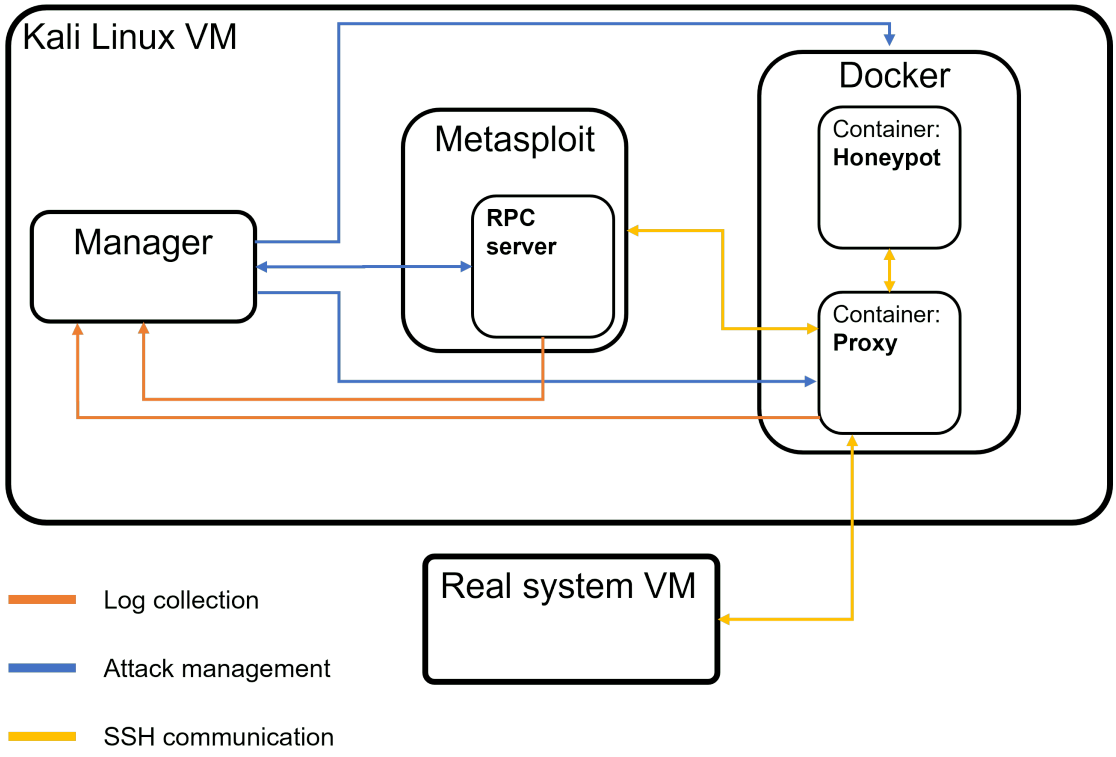

Figure 4.3: System architecture highlighting how subsystem parts interact.

without having to adjust hard-coded information in the program code. JSON files are a great format for computer use, because they can be parsed into a Python dictionary, that can be easily accessed and manipulated.

The configuration file presents the following keys:

- exploits: a list of JSON objects containing the exploits that we want to deploy. For each of them, exploit-specific settings can be specified.
- conf_files: a list of the configuration file names of the victims. Each of these files corresponds to a victim.
- test_bed_conf: a list of general settings that apply to all exploits. They can be overridden for a specific exploit by specifying the same one in the exploit-specific settings. In this part, it is specified the number of trials to be performed which is the number of times an attack has to be repeated.
- rpc_psw: password of the Metasploit RPC server
- rpc_port: port of the Metasploit RPC server

Interaction with the Metasploit framework

The manager is connected to the Metasploit framework thanks to its Remote Procedure Call API. Through this service, it is possible to interact with Metasploit, using the same set of commands accessible through the CLI. We employed a Python library specifically designed to interact with this API, called Pymetasploit3 [5]. This library allows to interface with MSF modules, sessions, and consoles [5].

Unfortunately, this library does not provide asynchronous event handling. Hence, we have to perform tasks synchronously with the Metasploit Framework as they were manually run from the console. Despite this issue, the library provides the possibility to read the MSF console and to interface with it as we were using it manually. The library ensures that the console is read without losing information by buffering the data that has not been read yet. Furthermore, by reading the MSF console, we are constantly aware of the status of the attack, and the manager can act accordingly and record the result.

Victim selection

As mentioned before, the manager is in charge of the victim selection task. Victims are specified in the manager configuration file. The victim configuration file has to be set in the proper proxy folder. However, the proxy configuration file needs to feature always the same name: *cowrie.cfg*. For this reason, in the same folder of the main configuration file, there are other copies of the same one, named with a meaningful name representing the victim and containing its specific settings. Every time the victim needs to be changed, the manager copies the proper configuration file content into the main one.

The proxy reads the configuration file only once at start-up. For this reason, every time the victim is changed, the proxy subsystem needs to be restarted. However, since it runs on a Docker container, it requires very little time to start up and it is easy for the manager to control it.

Attack management

The attack selection process is performed by reading from the configuration file the list of exploits that has to be deployed. Each attack is different, as its goal can change from one to another. For this reason, for each attack, specific settings can be applied. An example of JSON object used to describe an attack is reported in Figure 4.2. Each object contains two keys:

[5] https://github.com/DanMcInerney/pymetasploit3

- name: uniquely identifies the attack script in the Metasploit Framework library. For what concerns exploits, it must report the exact path from the exploit folder to the script location without its extension. For all the other modules, it is necessary to specify the path from the modules folder to the script location without its extension.

- parameters: lists of JSON objects containing the name of the attack-specific settings and their value.

```
{
    "name": "linux/ssh/microfocus_obr_shrboadmin",
    "parameters": [
        {
            "name": "PASSWORD",
            "value": "admin"
        },
        {
            "name": "USERNAME",
            "value": "root"
        }
    ]
}
```

Figure 4.2: Example of JSON object for exploit settings

Log collection

Logs collection is an important task because it allows a comprehensive analysis of the results. As mentioned in subsection 4.2.2, the proxy produces three types of log files. In order to simplify the data collection process, the manager collects the JSON and LOG files and it regroups them in one single JSON document. This file is the output of the entire test. In this document, a list of JSON objects is reported, one for each deployed attack with the following keys:

- victim: the name of the victim that has been tested in the format "trial number"_"victim name"

- exploit: the copy of the JSON object describing the attack reported in the system settings file

- success: boolean value reporting if the attack was able to retrieve a shell (true) or not (false)

- timestamp: the timestamp corresponding to the moment the attack started
- tty: the name of the binary file containing the the TTY logs
- msf_logs: the Metasploit console output during the attack
- logs: the proxy logs from the JSON file
- logs.log: the proxy logs from the LOG file

Logs analysis is not performed directly by the T-Hon system, but it is done in a later time using data analysis techniques.

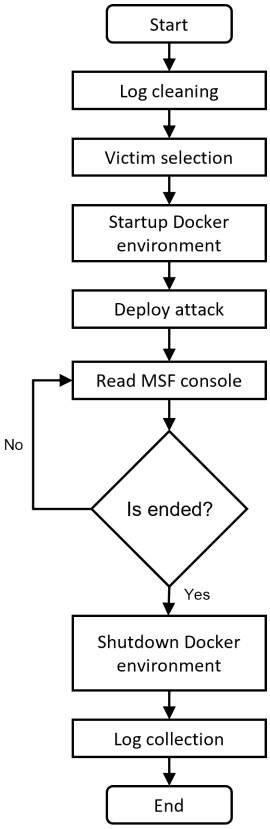

Figure 4.4: Attack management flow chart

4.3 Testing process

The testing process is a complex procedure since it has to assure that all the data is correctly collected. For each trial and for each victim, the procedure described in the flow chart reported in Figure 4.4 is performed.

The procedure starts by cleaning old log files. Now, the victim selection is performed as explained in section 4.2.4. Then, the docker environment is started up. In this way, the proxy and all the other systems running on Docker containers are initialized. Later, the attack is deployed. The system waits for it to finish and in the meantime continues to read the Metasploit framework console to be aware of the attack status, logging each output. Once the execution of the Metasploit attack script is terminated, the docker environment is shut down. At this point, the log files produced by the proxy are collected and organized into the JSON file that will contain all the results of all the performed attacks.

4.4 Requirement compliance

At the beginning of this chapter, we reported a set of key requirements for the development of the system. In this section, we will verify that those requirements were met by the adopted system design. The list of requirements and the explanation of how this design meets them is reported in table Table 4.1.

Requirement	**Solution**
Automatic attack execution	We developed a module that automates the attack process managing the Metasploit Framework
Adaptability	The system employs a file that reports all the parameters for its general functioning and specific settings related to the attacks to be deployed
Modularity	The system is based on a distributed architecture, making easier to modify or integrate new features and modules
Attack activity recording	The system records all the SSH connection protocol traffic and it organizes the collected data for an easy and complete analysis
Victim management	The system is able to automatically select the victim to attack

Table 4.1: T-Hon system requirements compliance

Chapter 5

Case studies

Using the data collected through T-Hon, we try to assess the fidelity performance of Cowrie responses and highlight possible discrepancies compared to real operative systems. We performed several attacks against three different victim systems: Cowrie, Ubuntu, and Metasploitable. In this way, we will evaluate how effective T-Hon is in providing useful and meaningful data. In section 5.1 we describe the methodology employed to carry out these tests. In this section, we provide some metrics that are useful to quantify the performance of the honeypot in terms of its ability to behave as an actual system and we try to use them to evaluate test results. In section 5.3 we report three cases that show how T-Hon is a valuable tool for collecting data from honeypot tests.

5.1 Methodology

In this section, we will describe how the test bed is set up and what information have to be gathered to perform a comprehensive test. In order to perform comprehensive tests and according to the work objective, we will employ some of the penetration testing activity phases reported in section 3.2. These tests are not focused on discovering new vulnerabilities per se, since we will employ existing attacks created to exploit known security issues, hence the scanning phase will not be performed.

Nevertheless, for what concerns the other steps, in the planning and reconnaissance phase, all the attacks were gathered and a set of victims was chosen as reported in subsection 5.1.2 and subsection 5.1.1. Furthermore, according to the previous choices, T-Hon settings were defined as reported in subsection 5.1.3. Then, the third and fourth phases, respectively gaining access and maintaining access, were carried out following the steps provided by the attacks if implemented. The last phase is the object of section 5.3, where the results of the previous phases are analyzed to highlight differences in the victim responses.

As reported in subsection 4.2.2, the data collected refers to the SSH connection protocol. The logs will contain the commands sent by the attacker and the relative victim response. Furthermore, they will provide detailed information about the SSH connection protocol messages. This feature enables a thorough inspection of the exchanged payload.

5.1.1 Victims

In this work, we will perform attacks through the Secure Shell Protocol protocol towards three different victims:

1. Cowrie 2.4.0
2. Metasploitable 2.6.24-16-server
3. Ubuntu 22.04 LTS

The honeypot under evaluation is a state-of-the-art system called Cowrie which is a Linux-emulated SSH honeypot. In order to compare the results, the other victims have to be Linux based and support the SSH protocol. The choice of Metasploitable was driven by the interest in having a test subject that could help us understand how a completely vulnerable device would respond. As reported in section 3.2.1, Metasploitable is an intentionally vulnerable machine conceived for validation and demonstration purposes.

On the other hand, we needed a real system that could represent a common host. For this reason, Ubuntu was chosen, since it features a very high diffusion, accounting for 33.9% of the Linux market [32]. The structure of the test bed and its architecture is described in chapter 4.

5.1.2 Attacks selection

As reported in subsection 5.1.1, the systems under testing are all Linux based and support the SSH protocol. Hence, the attacks to deploy need to be compatible with the system employed in the tests. For this reason, the attack research was focused on finding scripts dedicated to SSH vulnerabilities or to vulnerable systems that could be exploited through the SSH protocol based on the Linux operative system.

The Metasploit Framework provides a complete library of exploit scripts that can be used against a host, including specific attacks on SSH. Several other scripts [1] could be found online exploiting known security issues of devices whose SSH implementation features some vulnerabilities. However, most of them are written

[1] https://0day.today

in Python and are not directly integrable into the Metasploit Framework, hence we decided to focus on the attacks available in the framework database. Metasploit library[2] contains a set of modules specifically designed for the SSH protocol as reported in Table 5.1. Exploits modules are scripts designed to take advantage of known vulnerabilities to perform malicious tasks on the victim machine. Fuzzers modules are scripts that inject random and unexpected data into programs or stacks to discover bugs and faulty behaviors. Scanners are modules conceived to look for characteristics of the victim in order to gather useful information [28]. For these tests, we selected 14 exploits and 1 scanner among the available ones. The higher number of chosen exploit modules with respect to the scanners is due to the fact that we are interested in the interaction the attacker has with the system not only at the login but even once this has succeeded, to see the commands it sends to accomplish its malicious task.

Module type	**Quantity**
Exploits	14
Fuzzers	4
Scanners	14

Table 5.1: Number of Metasploit modules specifically created for the Secure Shell protocol

Among the 14 exploits, only 5 were able to successfully perform login to the victim. The remaining 9 do not use a username and password to log in but they rely on SSH keys.

5.1.3 T-Hon settings

The T-Hon system is set up through its configuration file *settings.json*, to execute each of the attacks once against each victim. The list of the attacks and settings that are exploit-specific, such as username and password, are specified in the configuration file. Since the attacker always interacts with the proxy first, the credentials are always related to the proxy.

For each victim, a different proxy configuration file is created and specified in the T-Hon configuration file as well as the proxy port, the proxy IP address, and the RPC port and password needed for the connection to the Metasploit Framework. In the proxy configuration file *cowrie.cfg* the credential to log into the three victims specified in subsection 5.1.1 are automatically set. T-Hon will set the

[2]https://github.com/rapid7/metasploit-framework/tree/master/modules

proper proxy configuration file before starting the attack deployment without any kind of user intervention. Depending on the attack goal, if a shell is obtained, then the command "exit" is sent from the attacker to the victim to terminate the session. This command is not part of the attack script, but it is used by T-Hon to record if commands can be correctly sent to the victim machine once a shell is gathered.

5.2 Test goals

As previously reported in chapter 2, the objective of this work is to create a system that will provide analysts with useful data to evaluate the quality of the honeypot response. This is done by performing attacks through the SSH protocol against honeypots and real systems, thus comparing the collected responses to highlight differences and potential errors.

In order to understand how attacks are deployed to the victims we summarize a set of steps that can give an overall evaluation of how further the attacker is able to go in the interaction with the victims. These steps have been identified among the attacks listed in Appendix A. The steps, reported in Figure 5.1, are numbered as follows:

1. Connection opening: capability of establishing an SSH connection with the attacker.

2. Login: capability of correctly performing login operations.

3. Shell gathering: capability of providing a shell.

4. Execution of attack-specific commands: capability of executing commands sent by the attacker.

Despite all the attacks having the first two steps, not all of them have step number 4 since their goal is only to get access to the victim system and gather a shell. Furthermore, as reported in Figure 5.1, in some case the command execution is performed before the shell gathering since the execution request is embedded in the *channel request* message specifying "exec" in the type field in the payload and the command to be executed.

5.2.1 Metrics

Defining metrics to determine honeypots performance is a hard task since it is complex to quantify their performance. As we reported in chapter 2, in the literature [16] there are definitions of some metrics based on:

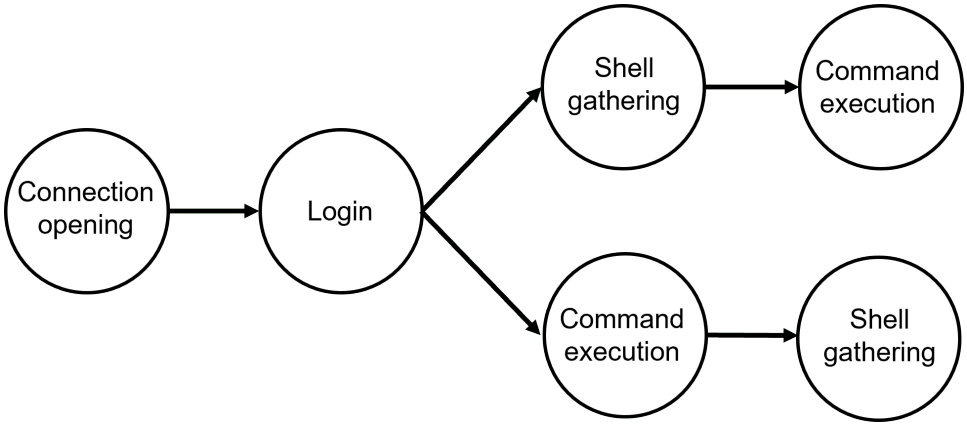

Figure 5.1: Attack steps for performance evaluation

1. Fingerprinting: the ability of the honeypot in identifying changes in its environment or in the services it provides.
2. Data capture: honeypot ability to collect high-quality data from the attacker.
3. Deception: honeypot ability to deceit an attacker into continuing the interaction with it.
4. Intelligence: the level of interaction and machine learning techniques implemented in the honeypot.

Fingerprinting is a metric that can be estimated by looking at the honeypot from a client's point of view. In that case, the honeypot would probe a set of victims to gather information, such as understanding what services are available and what ports are open. However, in this work, we are looking at the honeypot from the attacker's point of view. The intelligence metric can be seen more as a classification parameter, stating the level of automation and adaptability of a honeypot based on the implementation employed. In this case, this can be easily inferred from Cowrie specifications. Data capture can be employed to evaluate the quality of the honeypot collected information. It can be based also on the ability of the honeypot to recognize commands and categorize them.

Deception is a fundamental parameter in evaluating honeypot performance. How the honeypot is able to trick an attacker into continuing the interaction with it is a key factor as it allows the collection of further information concerning how attacks are performed. This last metric can be evaluated thanks to the data collected by T-Hon. The ability of a honeypot to deceive an adversary is directly connected to the fidelity of its response. The higher the ability of the honeypot to behave as a real system the higher the probability that an attacker will continue its task since

it does not notice that it is dealing with a mocked system. This is very important especially when attackers are automated machines since they expect responses following protocol standards and if this is not the case, the attacker will leave without completing its attack or trying several times without success, thus affecting the completeness and quality of the collected data. To evaluate the deception of a honeypot we propose to use an evaluation method based on how many steps of a certain attack the honeypot is able to complete. In order to have a ground truth comparison, we suggest applying the same procedure to a real system under the same conditions. In this way, it will be clearer how real and mock systems perform when subject to the same attack.

This kind of evaluation can be employed when the environment in which the test is performed is a controlled one, as the one in which T-Hon will be used. However, how could we perform a similar evaluation in the case of a deployed system opened to interaction with the external world? In this case, an important parameter to take into consideration is the number of distinct commands that a honeypot receives in an SSH session. This measurement is insightful since it gives an idea about the interaction between the honeypot and the attacker. If an attacker sends multiple distinct commands in a session, it is probable that the honeypot is responding in an expected way, as a real system would do. Nevertheless, this parameter cannot be employed as an absolute score since we do not know which are all the steps of the attack in advance. For this reason, it is essential to deploy different systems under the same conditions to perform a meaningful comparison.

As previously highlighted, fingerprinting can be used by honeypots to understand the environment in which they are. However, as reported in [18], this technique can be employed against honeypots to understand if there are some signatures that would give proof of interacting with a mock system. Discovering such evidence is of great importance to maintain the effectiveness of a honeypot. The signature of a honeypot is hard to quantify, as it is actually an error or odd behavior in the way the system works with respect to a real one. As a matter of fact, the presence of a signature could drastically influence all the other parameters, since it would decrease the fidelity of the response and inhibit the deception capacity since the honeypot could be detected by attackers looking for a discovered signature. Nevertheless, in order to give an evaluation of such an important feature, we will report the number of issues that could potentially bring to the discovery of a signature. This cannot be done in an absolute way, it is necessary to analyze the honeypot response with respect to a real system. For this reason, it is important to perform tests against both honeypots and actual systems to have a baseline result to compare with the other.

Summarizing what until now reported, we propose in the context of this work,

the following parameters as metrics to evaluate the performance of the honeypot under testing:

1. Attack steps score: is the ratio between the number of steps that the adversary is able to complete on the victim and the total number of steps of a known attack as reported in the following formula:

$$AS_{score} = \frac{NS_{success}}{NS_{total}}$$

where AS_{score} is the attack steps score, $NS_{success}$ is the number of successfully completed steps and NS_{total} is the total number of steps of the attack.

2. Attack commands score: it is the ratio between the number of received input line commands that are correctly executed by the victim and the total number of commands in a known attack, as reported in the following formula:

$$AC_{score} = \frac{NC_{success}}{NC_{commands}}$$

where AC_{score} is the attack commands score, $NC_{success}$ is the number of successfully executed commands and NC_{total} is the total number of commands of the attack.

3. Received commands: number of commands received during an SSH session.

4. Signature: number of issues that could potentially constitute loss of deceitfulness.

The first three parameters are linked since the fact that an attacker continues in its activity is related to the number of commands it sends and to the ability of the victim to correctly execute them. However, depending on the environment in which tests are carried out, the attack may be not known a priori, making it impossible to estimate the first parameter. For the real victims, the signature parameter is not applicable since they employ a real implementation of the service.

5.3 Results

In section 5.1 we reported the methodology that we employed to perform the tests. In this section, we present the data collected through T-Hon in real test cases. We analyze the results of three different attacks of the fifteen modules that were identified in subsection 5.1.2 and reported in appendix Appendix A. We present the findings of these attacks since they are the most representative of the value of T-Hon. For each of them, we discuss how the honeypot responded compared to the other two real systems, highlighting possible discrepancies between the three systems and providing a quantification of the metrics reported in subsection 5.2.1.

5.3.1 Attack 1: Cisco UCS scpuser

This exploit is based on a known default password of the Cisco UCS Director, which is a heterogeneous platform for private cloud Infrastructure as a Service [33].

This vulnerability is identified by [34], introduced in section 3.2 with the CVE 2019-1935 and it got a base score in the CVSS of 9.8 out of 10. As reported by [34], the vulnerability was caused by an undocumented default password and incorrect permission settings of a documented default account. As we can notice by looking at the CVSS score, this vulnerability was an important security issue, because a successful exploit could grant the attacker full read and write access to the system database.

The Metasploit exploit based on this vulnerability is contained in the MSF library in the *cisco_ucs_scpuser.rb* file. According to the steps identified in paragraph section 5.2 only steps 1, 2, and 3 are carried out since its goal is just to gain access to the system. After the attacker performed a successful login, it requests a shell by specifying it in the *channel request* message. Then, it tests the victim for the presence of a shell by sending two times the command "echo" followed by a string of random characters. The attacker verifies that the string it sent is the same one it received.

In Table 5.2, is reported the number of SSH messages that are exchanged between the attacker and the victims and vice versa. As we can see, it is clear that there are some important differences in how the systems under analysis respond to the same attack. Cowrie sent three times the number of messages to the attacker with respect to the other two, while the two real systems responded in a similar way in terms of the number of messages.

Direction	**Cowrie**	**Metasploitable**	**Ubuntu**
Attacker to victim	3	6	6
Victim to attacker	33	8	10

Table 5.2: Number of SSH messages for each victim for the Cisco UCS scpuser attack

In Table 5.3 we reported the number of messages in both directions (from the attacker to the victim and from the victim to the attacker) divided by type for each of the victims. By dividing the SSH messages by type, it is even more explicit the discrepancies among victim responses. For what concerns the honeypot Cowrie, the reception of one message from the attacker generated 31 *channel data* messages. However, the number of *channel success* and *channel open confirmation* messages is the same.

On the other hand, there are few differences between the two real systems. Ubuntu sends one *channel data* message more than Metasploitable, due to the fact that once the connection opens and the shell is requested, Ubuntu sends welcome information, while Metasploitable does not. The last difference between these two is that Ubuntu sends a global request with the specification *hostkeys-00@openssh.com* which is an optional OpenSSH extension that allows the server to inform the client about its protocol host keys after a successful authentication[3]. The *channel close*, *channel eof*, *channel window adjust*, and *channel global request* are not sent in the Cowrie case. In this case study, the number of messages sent by the attacker is different in the Cowrie case with respect to the other. This is due to the fact that the attacker was not able to continue its task as much as on the real systems.

	A→C	**C→A**	**A→M**	**M→A**	**A→U**	**U→A**
CHANNEL DATA	1	31	3	2	3	3
CHANNEL OPEN	1	-	1	-	1	-
CHANNEL REQUEST	1	-	1	1	1	1
CHANNEL OPEN CONFIRMATION	-	1	-	1	-	1
CHANNEL SUCCESS	-	1	-	1	-	1
CHANNEL CLOSE	-	-	1	1	1	1
CHANNEL EOF	-	-	-	1	-	1
CHANNEL WINDOW ADJUST	-	-	-	1	-	1
GLOBAL REQUEST	-	-	-	-	-	1

Table 5.3: Number of SSH messages divided by type for Cisco UCS scpuser attack. A: Attacker, C: Cowrie, M: Metasploitable, U: Ubuntu

Payload analysis

In order to understand why there is such an important difference between the messages sent by Cowrie with respect to the other two, we inspected messages payload and sequences. Among the different messages that both the honeypots and real systems have sent at least once, the *channel open confirmation* message shows some peculiarities.

[3] `https://cvsweb.openbsd.org/src/usr.bin/ssh/PROTOCOL?annotate=HEAD`

Victim	Byte sequence
Cowrie	\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x80\x00
Metasploitable	\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00
Ubuntu	\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00

Table 5.4: Byte sequence of the *channel open confirmation* message payload bytes sequence sent from the victims for the Cisco UCS scpuser attack

As we can observe from Table 5.4, there is only one byte that changes in the sequence compared to the others. According to the SSH connection protocol standardized by the RFC 4254 [15], the highlighted bytes refer to the initial window size, which specifies how many bytes of data can be sent on the channel without having to modify it. From a protocol point of view, this does not constitute an issue. The *channel success* messages are the same for all victims.

Connection analysis

Further insights, can be obtained by inspecting the sequence of SSH messages sent between the parts. In Figure 5.2 and Figure 5.3 are reported the SSH connection diagrams of the attacks towards Metasploitable and Cowrie, respectively. In Figure 5.2, it is possible to notice the three *channel data* messages corresponding to the two "echo" commands and the "exit" command sent from T-Hon. In Figure 5.3, we can notice only the first *channel data* message corresponding to the first "echo" command. As we can see, in the Metasploitable case, the victim responded to the attacker with a *channel window adjust* message, while in the cowrie case this does not happen. Then, in the first case reported in Figure 5.2, the attack continues as expected giving a successful result, while in the second case reported in Figure 5.3, the victim continues to send unexpected data to the attacker, thus preventing it to continue its task.

Again, we performed a control on the payload sent in the *channel data* messages from Cowrie to the attacker and we saw that the honeypot responded to the attacker shell test by sending one character for each message as shown in Figure 5.4, instead of sending the entire string back as one. This is the reason for the high number of messages from Cowrie to the attacker reported in Table 5.3. On the other hand, the two real systems behave as expected. The results of this test suggest that Cowrie has problems in handling the shell request.

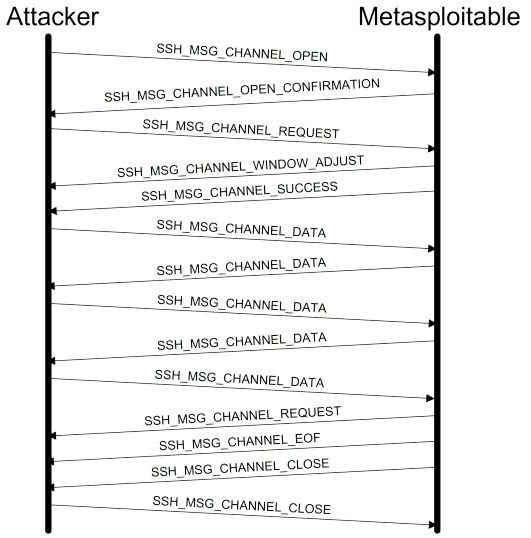

Figure 5.2: SSH connection between the attacker and Metasploitable for the Cisco UCS scpuser attack

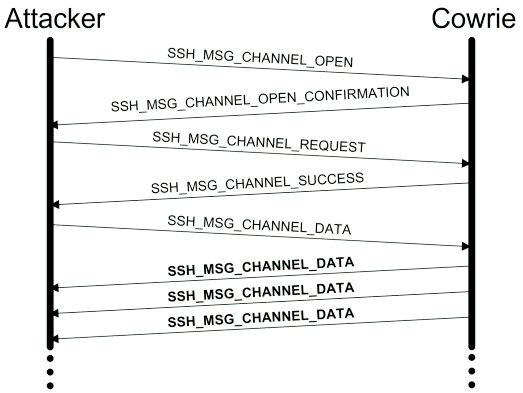

Figure 5.3: SSH connection between the attacker and Cowrie for the Cisco UCS scpuser attack

```
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01e'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01c'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01h'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01o'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01 '\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01G'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01R'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01u'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01Q'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01d'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01s'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01n'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01w'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x011'\n",
- b'\\x00\\x00\\x00\\x00\\x00\\x00\\x00\\x01S'\n",
```

Figure 5.4: Extract of the log reporting the particular Cowrie behavior when responding to the "echo 'randomBytes'" command for the Cisco UCS scpuser attack

Metric evaluation

In Table 5.5 the evaluation of the metric is reported. Figure 5.5 shows the steps that the attack performs and the result in carrying them out for the three different

victims. Consequently, the attack deployed toward cowrie was able to perform two tasks out of three, since due to some faulty behavior in the "echo" command handling, the attacker was not able to verify the presence of the shell. For what concerns the other two victims, the attack succeeded in carrying out all steps.

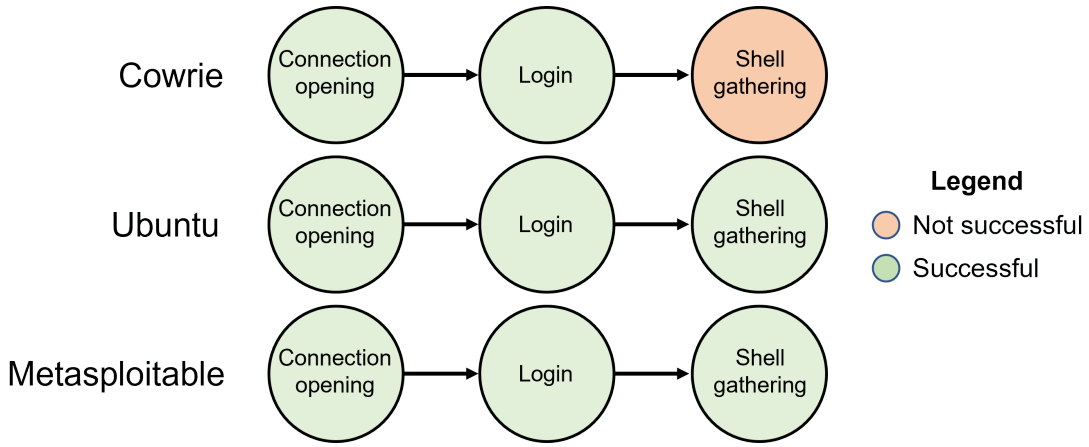

Figure 5.5: Attack result for the three victims for the Cisco UCS scpuser attack

As a consequence given the three steps of the attack, the attacker was able to successfully carry out two of them. In this way, Cowrie obtained an attack steps score of 66%. The number of input line commands employed in the attack was 2. However, if the shell was correctly verified then, the T-Hon system would have sent the "exit" command. Consequently the total number of commands would have been 3. However, the honeypot did not succeed in performing any of them obtaining an attack command score of 0 %. For what concerns the number of received commands, due to the honeypot inability to correctly execute them, it received only 1 input line command. In the end, we identified 2 issues that could compromise Cowrie deceitfulness:

1. A predefined initial window size value in the *channel open confirmation* message

2. An odd response to the command "echo", consisting in sending one byte per *channel data* message instead of the entire string

Consequently, the signature parameter assumes the value 2. In conclusion, the overall performance of the honeypot on the Cisco UCS scpuser attack is poor. Even if the attack steps score is pretty high, it was not able to correctly execute any shell verification command.

Direction	Cowrie	Metasploitable	Ubuntu
Attack steps score	66 %	100 %	100 %
Attack commands score	0 %	100 %	100 %
Received commands	1	3	3
Signature	2	n.a.	n.a.

Table 5.5: Metrics evaluation for the victims under the Cisco UCS scpuser attack

5.3.2 Attack 2: SSH Login

The SSH Login is an auxiliary module that is used to perform SSH logins on a list of machines. This module is not classified as an exploit since it does not take advantage of a known vulnerability. Consequently, it is inserted into the scanner categories since it can scan a certain number of hosts trying to log into them with a list of credentials. This kind of tool could be useful to test several victims trying to see if some of them can be entered with common unsecured credentials. However, in the module description it is reported the CVE 1999-0502 which refers to the fact that the Linux system may have an unsecured password, that can be easily guessed.

This module is contained in the MSF library in the *SSH_login.rb* file. Despite the fact that it aims only at performing the login, it does execute a command on the remote machine. The command is "id", which prints the system identifications for the logged user[4]. Furthermore, it presents a particular feature called *Gather Proof* that aims at verifying that the access was actually carried out successfully. This is performed by trying to identify the system type that was accessed by analyzing id information before opening a Metasploit session requesting a shell.

This module carries out all the four steps identified in section 5.2, in the order 1, 2, 4, 3. The module starts by performing login, and request the execution of the id command. Then, if the *Gather Proof* option is set to *true*, the previous step is repeated. At this point, if the login had a positive result, independently from the result of the *Gather Proof* process, a shell is requested, and if successful a Metasploit session is opened. Furthermore, if the previously reported step is acknowledged, the T-Hon will send the "exit" command.

In Table 5.6 the total number of SSH messages exchanged between victims, the attacker and vice-versa are reported. The number of exchanged SSH messages is

[4]https://www.ibm.com/docs/en/aix/7.1?topic=i-id-command

fairly similar among all victims. However, Cowrie presents slightly more messages sent to the attacker than the other two victims.

Direction	Cowrie	Metasploitable	Ubuntu
Attacker to victim	10	10	10
Victim to attacker	23	20	21

Table 5.6: Number of SSH messages for each victim for the SSH login attack

In Table 5.7 the number of SSH messages generated by the attacker and the victims divided by type is reported. The number of messages from the attacker is equal for all victims. This suggests that it was able to run the same actions on all of them. The number of *close*, *request*, and *open confirmation* messages sent to the attacker is the same for all victims. However, as in the case reported in subsection 5.3.1, the Ubuntu victim sends a *global request* message, while all the other victims do not. For what concerns the *data* message, Ubuntu and Metasploitable responded with the same number of messages. On the other hand, Cowrie responded with 11 *data* messages.

	A→C	C→A	A→M	M→A	A→U	U→A
CHANNEL CLOSE	3	3	3	3	3	3
CHANNEL DATA	1	11	1	2	1	2
CHANNEL OPEN	3	-	3	-	3	-
CHANNEL REQUEST	3	3	3	3	3	3
CHANNEL OPEN CONFIRMATION	-	3	-	3	-	3
CHANNEL SUCCESS	-	3	-	3	-	3
CHANNEL EOF	-	-	-	3	-	3
CHANNEL WINDOW ADJUST	-	-	-	3	-	3
GLOBAL REQUEST	-	-	-	-	-	1

Table 5.7: Number of SSH messages divided by type for the SSH login attack. A: Attacker, C: Cowrie, M: Metasploitable, U: Ubuntu

Payload analysis

In order to understand the reason for the differences reported in Table 5.7, we inspected the messages payloads and, later, the sequence in which they were sent. This analysis will focus on the victims responses since the attacker sent always the same payloads.

As reported in paragraph subsection 5.3.1, the *open confirmation* messages present some peculiarities. For the first and the third *open confirmation* messages, the byte sequence are the same for all victims except one byte related to initial window size that has the same behavior highlighted in yellow in tables Table 5.4 and Table 5.8. Also in this case, Cowrie has a different value for the initial window size with respect to the other real systems. The second *open confirmation* message, presents another peculiarity. The fourth byte reserved for the sender channel number, as defined in [15], is set to 1 while in the other victim responses are set to 0. The fact that Cowrie chooses another channel to respond, it is not a protocol issue. As reported in [15], the number referring to a channel may be different between the two connection ends. However, it is interesting to report that it has a different behavior with respect to real systems.

For what concerns the *request*, *close*, *success* messages, their payload is the same one for all the victims.

Victim	Byte sequence
Cowrie	\x00\x00\x00\x01\x00\x00\x00\x01\x00\x02\x00\x00\x00\x00\x80\x00
Metasploitable	\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00
Ubuntu	\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x00

Table 5.8: Byte sequence of the second *channel open confirmation* message payload sent from the victims for SSH Login attack. Highlighted in green are the bytes related to the sender channel and in yellow are the bytes related to the initial window size

Connection analysis

In order to gain more information to understand the differences reported in Table 5.8 we analyzed the sequence in which the messages are sent during the SSH connection. In Figure 5.7 and Figure 5.6, the SSH connection diagram for Cowrie and Metasploitable respectively are reported.

The Ubuntu connection diagram is very similar to the Metasploitable one except for the *global request* message that we previously highlighted and reported in Table 5.7. The diagrams report the first SSH connection that is established during

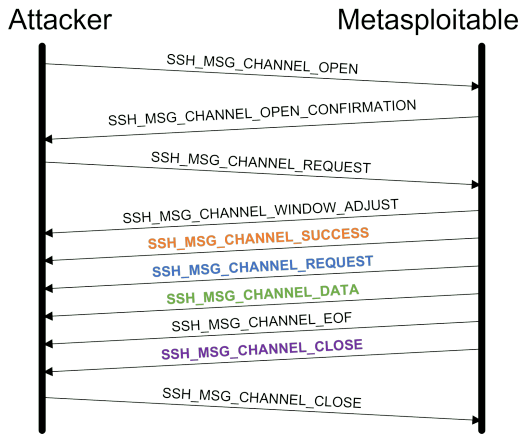

Figure 5.6: First SSH connection between the attacker and Metasploitable for the SSH login attack

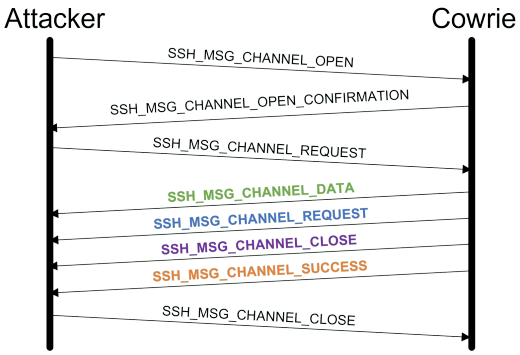

Figure 5.7: First SSH connection between the attacker and Cowrie for the SSH login attack

the attack. The second connection is identical to this one from the diagram point of view. This is due to the fact that the Gather Proof option was set to true. The first three messages are the same and in the same order for both victims.

Nevertheless, starting from the fourth, we can notice some differences. In the Metasploitable case, a *channel adjust* message is sent while in the Cowrie case, it is not. Then the *channel success* message is sent. In the Cowrie diagram, a *channel data* is sent. In this case, the attacker requests to execute the command "id" specifying *exec* in the request type payload field, through a channel-specific request. In this type of request, it is possible to specify in the payload in the *want proof* byte if the recipient has to answer it or not. In this case, the attacker requested a response. According to [15], after a channel-specific request with the *want proof* set to true, the recipient can answer in three ways:

1. *channel success*: the request is correctly recognized and supported by the channel.

2. *channel failure*: the request is not recognized or it is not supported by the channel.

3. Request-specific continuation messages.

The third option seems to be the one chosen by Cowrie since it responds to the attacker command directly with its result. However, a *channel success* message is sent, but in a sequence which is odd, since it is transmitted after the *channel close* message is sent. The *channel EOF* is missing in the Cowrie connection diagram.

However, according to [15], a party can send the *channel close* message without sending it before the *channel EOF*, thus, from a protocol point of view, this does not constitute an issue. Anyway, the fact that both real systems responded in the same way but differently from Cowrie, could provide some clues that there may be some protocol implementation issues.

The second SSH connection is the same as the one reported in Figure 5.7 and Figure 5.6. There are some more differences in the connection that is created to request a shell. After the shell is obtained, the T-Hon system sends the "exit" command to close it. For what concerns Cowrie, we found the same issue reported in subsection 5.3.1. The command is correctly received, however, Cowrie sends back the "exit" string, one character per packet even if this was not requested. These results further highlight a possible problem in the management of the shell request.

Metric evaluation

In Table 5.9 the metric evaluation is reported. The final result of the test in terms of steps carried out by the attacker is reported in Figure 5.8.

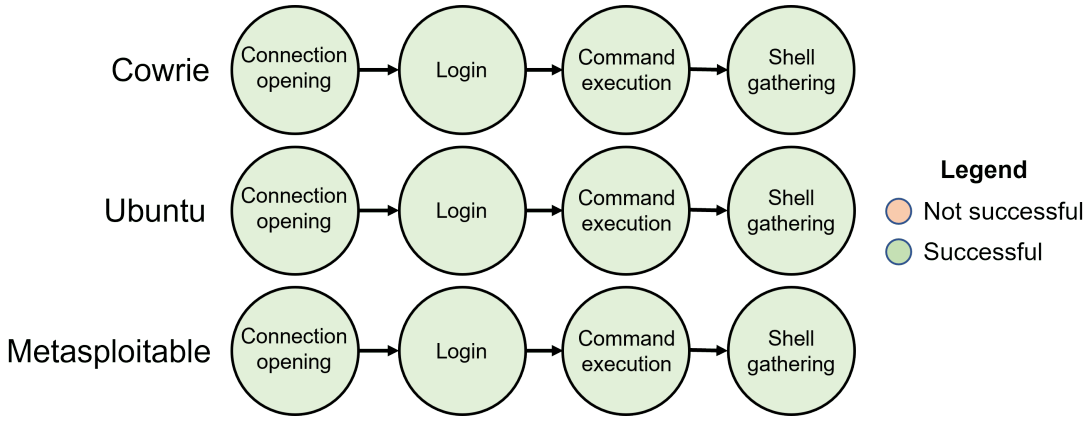

Figure 5.8: Attack result for the three victims for the SSH login attack

The outcome is marked as successful since the command is correctly executed from the result point of view and, most importantly, the shell is correctly gathered. As a matter of fact, the attacker does not verify the presence of the shell by performing some kind of test as in subsection 5.3.1. In this way, the shell is correctly obtained, however, several odd behaviors have been found. Consequently, given the four attack steps, the attacker was able to successfully carry out all of them. Hence, Cowrie obtained an attack steps score of 100 %. The number of input line commands employed in the attack were 2 corresponding to the two "id" command sent in two different connections, and considering also the "exit" command sent by T-Hon, we have a total number of commands sent equal to

3. However, the honeypot was able to correctly execute only the first two, thus obtaining an attack command score of 75 %. Since the first two commands were successfully executed and the shell successfully obtained, the T-Hon system sent the "exit" command, which was not correctly executed. Nevertheless, the number of received commands is 3. For what concerns the signature parameter, we identified 4 issues that could cause Cowrie to lose its deceitful ability:

1. A predefined initial window size in the *channel open confirmation* message, which was already highlighted in subsection 5.3.1.

2. An odd response to the command "exit", in the same way, reported in subsection 5.3.1.

3. A particular order in which responses are sent to the attacker.

4. The use of a different channel even if not necessary.

As a consequence, the signature parameter assumes the value 4.

Direction	**Cowrie**	**Metasploitable**	**Ubuntu**
Attack steps score	100 %	100 %	100 %
Attack commands score	75 %	100 %	100 %
Received commands	3	3	3
Signature	4	n.a.	n.a.

Table 5.9: Metrics evaluation for the victims under the Cisco UCS scpuser attack

In conclusion, the overall performance of the honeypot on the SSH login attack, for what concerns the first two parameters is good. However, even if the shell was correctly obtained, the "exit" command was not successfully executed. Furthermore, the number of issues that could constitute a problem from the signature point of view is high.

5.3.3 Attack 3: Quantum vmPRO backdoor

Quantum vmPRO is a backup and recovery application for data protection. It interacts with any operative system of any virtual machine [35]. The attack is based on a known command "shell-escape" which constitutes a hidden backdoor that allows obtaining a bash shell with full root permissions, even if the user has no administrative privileges. This module belongs to the exploit category and it is contained in the MSF library in the *quantum_vmpro_backdoor.rb* file.

According to the steps reported in section 5.2, this attack performs all the identified steps. The attacker connects to the victim, then performs the SSH login procedure. If successful, the attacker continues by requesting the execution of the backdoor command "shell-escape". If the command is correctly executed then a shell is requested. If the shell is correctly gathered, T-Hon will send the "exit" command.

The number of SSH messages exchanged between the victims is reported in Table 5.10. In this case, Cowrie sent fewer messages than the other two real systems. On the other hand, Metasploitable and Ubuntu exchanged a similar number of messages. The number of messages sent from the attacker is the same for all three victims.

Direction	**Cowrie**	**Metasploitable**	**Ubuntu**
Attacker to victim	3	3	3
Victim to attacker	5	7	8

Table 5.10: Number of SSH messages for each victim for the SSH login attack

In Table 5.11, the number of messages in both directions (from the attacker to the victim and from the victim to the attacker) divided by type for each of the victims are reported. First of all, the number of *channel close*, *channel open*, *channel request*, and *channel success* is coincident for all three victims. The number of *channel eof*, *channel extended data* and *channel window adjust* are equal for the two real systems: Ubuntu and Metasploitable, while they are not present in the Cowrie case. Finally, the *global request* is only present for Ubuntu as in the case reported in subsection 5.3.1 and subsection 5.3.2.

Payload analysis

As a consequences of the previous observations, we inspected the messages payloads, focusing on the ones that were at least sent once from all victims, to ensure a useful comparison. We will focus on the victims messages since for what concerns the attacker, they are always alike. First of all, the *channel close* and *channel success* messages have the same payloads for all victims. Also in this case, the *channel open confirmation* message presents the same behavior reported in Table 5.4. However, the *channel request* message displays some interesting insight as shown in Table 5.12. As we can see, there is a difference in the last byte. This message sent the result of the previously requested command execution. Since the backdoor command does not exist on any of the victims, bash returned the exit code 127 ($7F_{16}$) that

	A→C	C→A	A→M	M→A	A→U	U→A
CHANNEL CLOSE	1	1	1	1	1	1
CHANNEL OPEN	1	-	1	-	1	-
CHANNEL REQUEST	1	1	1	1	1	1
CHANNEL DATA	-	1	-	-	-	-
CHANNEL OPEN CONFIRMATION	-	1	-	1	-	1
CHANNEL SUCCESS	-	1	-	1	-	1
CHANNEL EOF	-	-	-	1	-	1
CHANNEL EXTENDED DATA	-	-	-	1	-	1
CHANNEL WINDOW ADJUST	-	-	-	1	-	1
GLOBAL REQUEST	-	-	-	-	-	1

Table 5.11: Number of SSH messages divided by type for the SSH login attack. A: Attacker, C: Cowrie, M: Metasploitable, U: Ubuntu

corresponds to the "command not found" error[5]. Both Metasploitable and Ubuntu append the exit code to the end of the message, however Cowrie does not, even if from the logs the command is not found.

Victim	Byte sequence
Cowrie	\x00\x00\x00\x00\x00\x00\x00\x0bexit-status\x00\x00\x00\x00\x00
Metasploitable	\x00\x00\x00\x01\x00\x00\x00\x0bexit-status\x00\x00\x00\x00\x7f
Ubuntu	\x00\x00\x00\x01\x00\x00\x00\x0bexit-status\x00\x00\x00\x00\x7f

Table 5.12: Byte sequence of the victims *channel request* message payload bytes sequence for Quantum vmPRO backdoor attack

[5]https://www.gnu.org/software/bash/manual/bash.html

Connection analysis

In Figure 5.9 and Figure 5.10, the SSH connection diagrams for Metasploitable and Cowrie are reported. Connection diagrams are a useful tool to understand the dynamics of the messages exchanged between attacker and victim. The Ubuntu connection diagram is equivalent to the one of Metasploitable with the only difference that in its case a *global request* message is sent before the *channel open* message. The cowrie message sequence displays the same particular behavior reported and analyzed in subsection 5.3.2. However, in this case, there is one more difference. Both Metasploitable and Ubuntu use a *channel extended data* message to send the result of the executed command, which is transmitted in the *channel request* message. As reported in [15], a party can use a *channel extended data* message to transfer data with the possibility of specifying the data type in a reserved payload field.

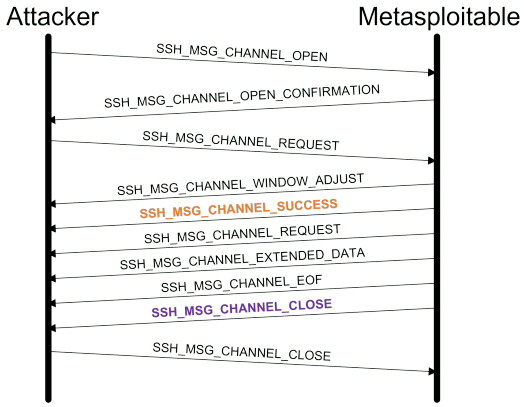

Figure 5.9: SSHconnection diagram between the attacker and Metasploitable for the Quantum vmPRO backdoor attack

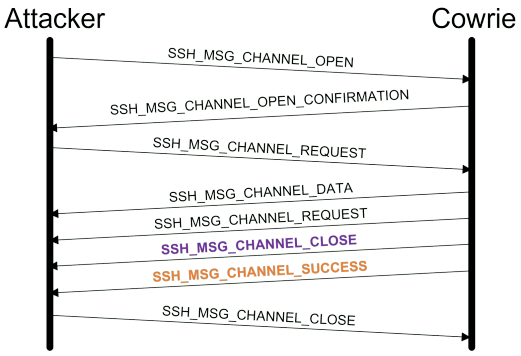

Figure 5.10: SSH connection diagram between the attacker and Cowrie for the Quantum vmPRO backdoor attack

Metric evaluation

In Table 5.13 the evaluation of the metrics proposed in subsection 5.2.1 is reported. The overall result of the attack divided by steps is reported in Figure 5.11.

In this case study, any of the victims was able to correctly execute the requested command, consequently, they were not able to provide a shell. This is due to the fact that all machines were not running the Quantum vmPRO application, thus they were not vulnerable to this type of attack. The overall attack steps score is 50 % since the attacker was able to perform only 2 steps out of 4. The attack

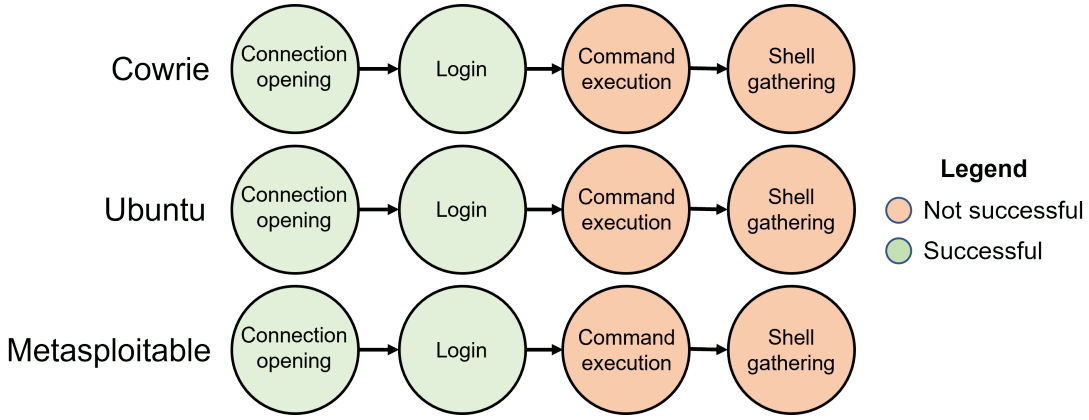

Figure 5.11: Attack result for the three victims for the Quantum vmPRO backdoor attack

command score is 0 % since any victim was able to give the correct result to the command execution. They received only 1 command from the attacker since, given the inability to retrieve a shell, T-Hon could not send the "exit" command. For what concerns the signature parameter, we identified 3 potential issues concerning the mocking capability of Cowrie:

1. The bash exit code is not reported to the attacker
2. A predefined initial window size as highlighted in subsection 5.3.1
3. A particular order in which responses are sent to the attacker as highlighted in subsection 5.3.2

As a consequence, the value of the signature parameter is 3.

Direction	Cowrie	Metasploitable	Ubuntu
Attack steps score	50 %	50 %	50 %
Attack commands score	0 %	0 %	0 %
Received commands	1	1	1
Signature	3	n.a.	n.a.

Table 5.13: Metrics evaluation for the victims under the Quantum vmPRO backdoor attack

In conclusion, the overall performance of the honeypot, compared to the one of the real systems, is good. However, the presence of several potential issues that could make Cowrie discoverable, does decrease its effectiveness.

5.4 Case studies review

In this section, we summarize the results that we highlighted in sections subsection 5.3.1, subsection 5.3.2, and subsection 5.3.3. First of all, thanks to the data collected through the T-Hon system we were able to identify several discrepancies between the honeypot behavior and the real systems responses. This was possible because T-Hon methodically deployed the attacks to all victims, collecting and organizing the resulting data. Analyzing the data concerning the three attacks that we reported as case studies we were able to highlight the following issues:

- Cowrie does not use a *channel window adjust* message as the real systems under analysis do. However, it seems to have by default a window value in the *channel open confirmation* message.
- When a shell is requested using a specific *channel request* message, Cowrie is not able to manage command execution. It responds to commands sending back one character for each message.
- When the attacker requests to execute a command through a specific *channel request* message, the order in which the messages resulting from the command execution are sent is peculiar.
- When a command fails its execution, the error code is not sent back to the attacker.
- Cowrie chooses different channels to respond to the attacker with respect to real systems.

The fact that we were able to identify all these issues proves the usefulness of T-Hon in supporting the evaluation of honeypot effectiveness.

Chapter 6

Work application

In this chapter, we will present the results of the analysis of the data collected from a real honeypot deployment. We applied some modifications to Cowrie to fix an odd behavior concerning the order in which SSH messages are sent that was highlighted in subsection 5.3.2. Then, a honeypot with the fixes and one without were deployed and exposed to real traffic. In this way, we were able to show how the T-Hon data supported the honeypot fixes and how they contributed to the collection of real attack data compared to the ones collected without the corrections.

6.1 Honeypot modification

As shown in Figure 5.7, the order in which the SSH messages are sent presents a particular characteristic. The *channel success* message is sent after the *channel close* message and most importantly, after the data requested in the attacker *channel request* is sent. Even if this could be accepted from the standard [15] point of view, since after a *channel request* a *channel data* message could be sent reporting the data requested in the attacker message, it is not accepted by the attacker. As a matter of fact, Metasploit is not able to verify if the shell is really a valid one, thus failing the Gather Proof procedure. Despite this behavior, the attack is successful since a shell is correctly retrieved.

In order to avoid this, a fix has been applied to the honeypot. This modification allows reordering the way SSH messages are sent. In this way, after the attacker *channel request* has been sent, the first message in response is *channel success*, then *channel data* message. Consequently, the Gather Proof procedure is successful.

In order to verify the result of the alterations that we made, we tested the fixed version of Cowrie with SSH login, which is the same attack that brought to the discovery of this particular behavior as reported in subsection 5.3.2.

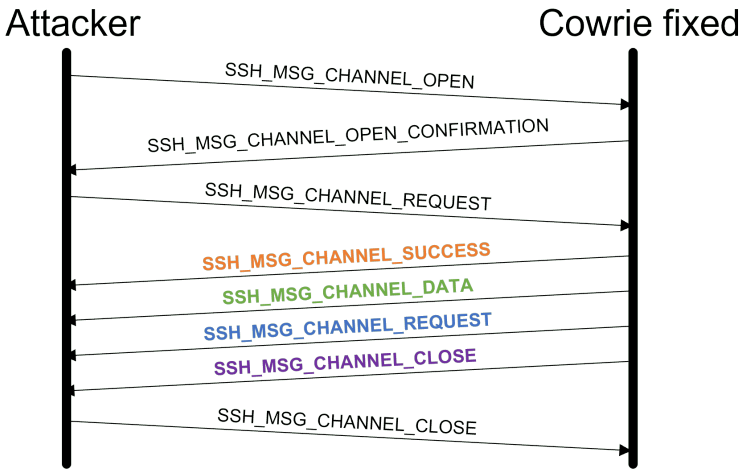

Figure 6.1: SSH connection between the attacker and the fixed version of Cowrie for the SSH Login attack

In Figure 6.1 the SSH connection diagram between the fixed version of Cowrie and the attacker for the SSH Login attack is reported. With respect to Figure 5.7, it is clear that the order in which response messages are sent is changed. Now, the SSH connection diagram is much more similar to the one of a real system under the same attack as the one shown in Figure 5.6.

6.1.1 Deployment

In order to understand how the modification that we made to the original version of Cowrie could affect its performance, we deployed two versions of the honeypot:

1. Cowrie original: standard version of Cowrie without any modification.

2. Cowrie fixed: version of Cowrie with the modification reported in section 6.1.

The two honeypots were installed into an already deployed system on a /16 network of the Politecnico di Torino. According to [36] in this network a /23 network was isolated and multiple /29 networks belonging to the same /24 network were employed. Each of the /29 networks features 8 IP addresses. 16 of these /29 networks were reserved for L-7 and L-4 responders i.e. honeypots. The two honeypots were deployed on two different /29 networks positioned in the middle of the /24 IP addresses range. The two honeypots were exposed to real SSH connections coming from the internet. The structure of this setup aims at making the two honeypots work under the same conditions, without influencing each other.

6.2 Analysis of the collected data

In this section, we present the data collected by the two deployed honeypots as described in subsection 6.1.1 and we will analyze and compare how the two honeypots behaved, through the quantification of the retrieved data and the graphical representation of meaningful characteristics. The data was collected for a period of 70 days, from 31st August 2022 to 8th November 2022.

6.2.1 Data quantification

In Table 6.1 a quantification of the collected data is reported, divided by honeypot deployment: fixed and original. The column *Started SSH handshakes* reports the number of initiated SSH handshakes, independently from the fact that they may have failed. The fixed Cowrie had about 16% less initiated handshakes with respect to the original one. However, the number of distinct IPs that interacts with both of them is fairly similar. This means that the number of attackers that interact with the two honeypots is almost the same.

Honeypot	**Started SSH handshakes**	**Distinct IPs**	**SSH login attempts**	**Commands received**	**Distinct commands received**
Original	2065102	21276	1605840	3991662	231193
Fixed	1724566	21460	1260015	3664297	232432

Table 6.1: Quantification of collected data divided by honeypot version: fixed and original

The fact that the number of handshakes is lower in the fixed one could be caused by the modification that we applied. Several times, attacks are carried out in an automated way using, for example, bots. If the attack is not successful, the automated system could decide to repeat it until a certain stopping criterion is met, such as a fixed number of attempts. This observation is supported by looking at the number of received commands. On one hand, in the case of the original version of Cowrie, this number is higher with respect to the fixed one. On the other hand, observing the number of distinct received commands, the number is quite similar. This means that both honeypots received almost the same number of distinct commands, but in the original version some of them are sent multiple times. This could justify the idea that in the original version, the same command may be sent multiple times because it does not give the expected result, while in the

fixed version this is not necessary. The number of login attempts is quite different between the two versions. The original version received 27% more attempts with respect to the fixed one. Again, supposing that there is a large interaction with automated attackers, the original version may need more attempts before the attacker stops.

6.2.2 SSH connections and IP addresses

In Figure 6.2, the number of started SSH handshakes is plotted for each day for both the original and fixed versions of Cowrie. The number of handshakes grows during the first 40 days of deployment reaching its peak around October 10$^{\text{th}}$. In the remaining days, it decreases to then have a smaller peak during the last days of deployment.

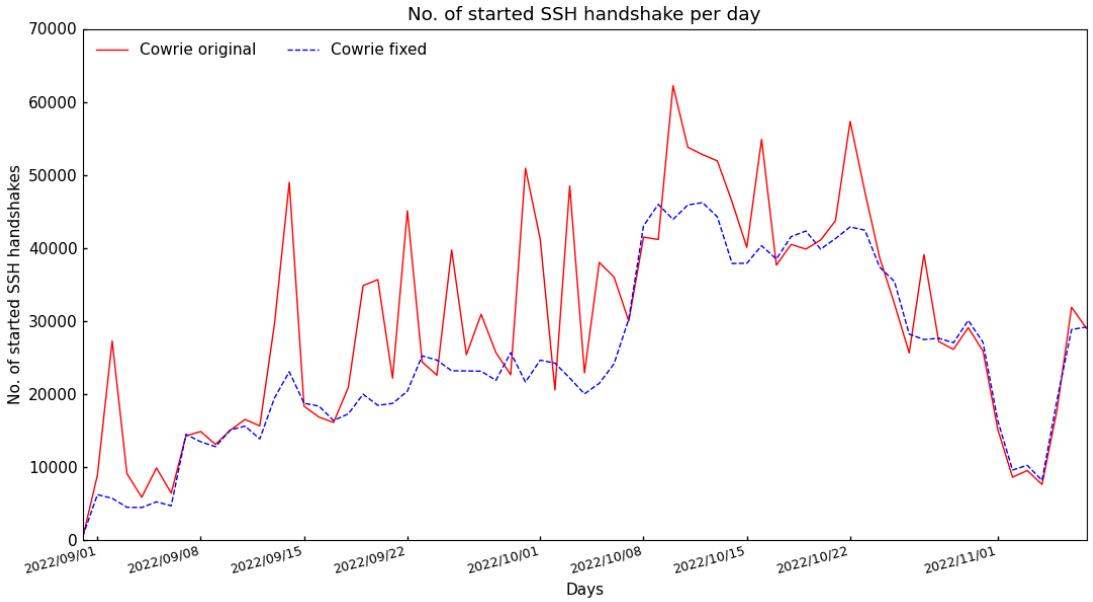

Figure 6.2: Number of started SSH handshakes per day in the case of the original and fixed version of Cowrie

At first sight, the dashed line representing the fixed version of Cowrie is more smoothed out with respect to the red one, representing the original Cowrie version. The peaks that are reported in the red line may represent the repeated attempts of establishing a connection and maybe execute line commands. In the surrounding area of those peaks, the dashed line presents much smaller peaks. This could represent the fact that in the original version of Cowrie, the attacker is not able to continue its task and it tries multiple times to perform it. Whereas, in the fixed version, it is more likely to succeed. This does not imply that the attack will

achieve its final goal, however, it is more likely to perform some more tasks that could potentially be blocked in the next steps.

In Figure 6.3 the number of distinct IP addresses that started an SSH handshake with both original and fixed Cowrie is reported. The two lines are fairly similar as they follow the same trend. The number of IP addresses that interacts with both honeypots is very low at the beginning of the experiment. Starting from September 6^{th}, the number increased drastically reaching a peak on September the 29^{th}. Then it decreases to reach a steady trend until the end of the time window. As remarked in Table 6.1, the number of distinct IP addresses is very similar among both Cowrie deployments. This suggests that the data captured is comparable and the two victims are subject to the same environment since the number of attackers that interact with the two honeypot versions is almost the same.

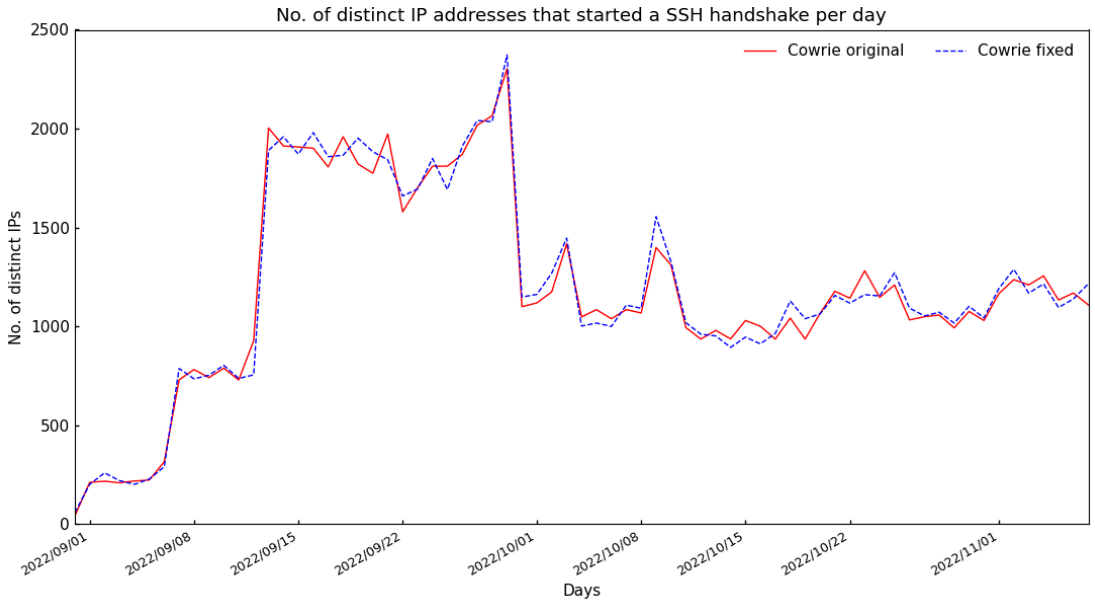

Figure 6.3: Number of distinct IP addresses that started an SSH handshake per day in the case of the original and fixed version of Cowrie

6.2.3 Commands received

In Figure 6.4, the Cumulative Density Function for the sessions as a function of the number of input line commands is shown. The CDF is a statistical tool that represents the probability that the random variable under consideration takes a value less or equal to the one reported on the x-axis. In this case, the random

variable represents the number of commands received by the attacker for each SSH session.

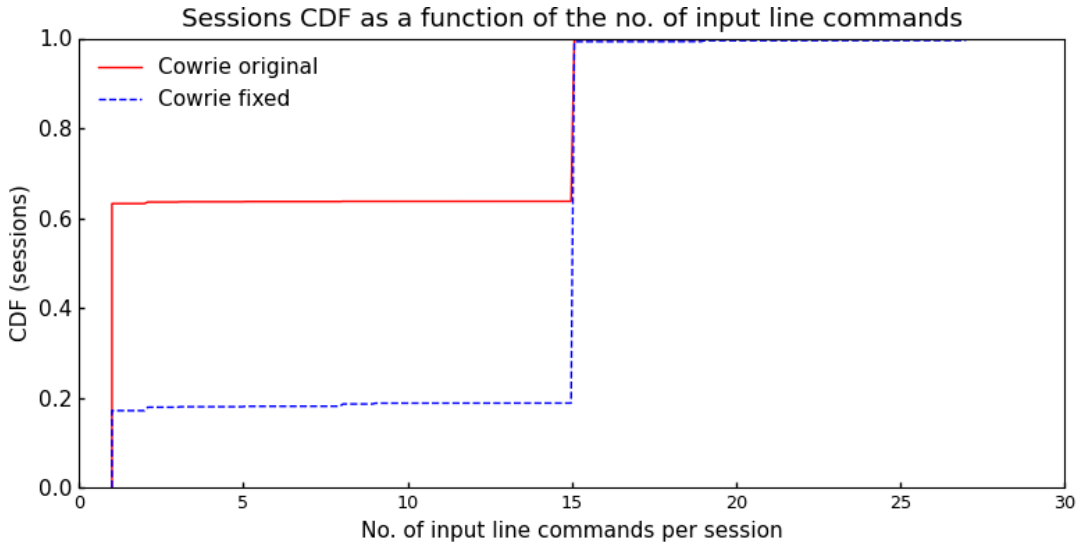

Figure 6.4: Session Cumulative Density Function as a function of the number of input line commands per session

Looking at the chart reported in Figure 6.4, over the 60% of SSH sessions with the original version of Cowrie registered only one command. This result is drastically decreased in the case of the fixed version of Cowrie. From this finding, we can infer that the number of SSH sessions with just one command is much smaller for the fixed version of Cowrie. The fact that below the 20% of sessions with the fixed version of Cowrie reports only one command, suggests that the attacker is able to send more commands to the fixed honeypot, as it may be able to continue its procedure for a longer number of steps compared to the original version of Cowrie. Few sessions can be noticed for a number of commands corresponding to 2, 3, 5, 8, 9, 19, 23, and 27, where the fixed version of Cowrie reports a slightly higher percentage of sessions with respect to the original one.

In Figure 6.5, the probability density function of the SSH sessions as a function of the number of commands sent in each of them is reported. This chart completes the representation reported in Figure 6.4, making clearer the difference between the two honeypots behavior. For what concerns SSH sessions with just one command, the same considerations made for Figure 6.4 are standing. Furthermore, we can notice a peak on fifteen line commands per SSH session corresponding to the vertical ascending line around the same number of line commands shown in Figure 6.4. In this case, around 80% of the SSH sessions between the attacker and the fixed

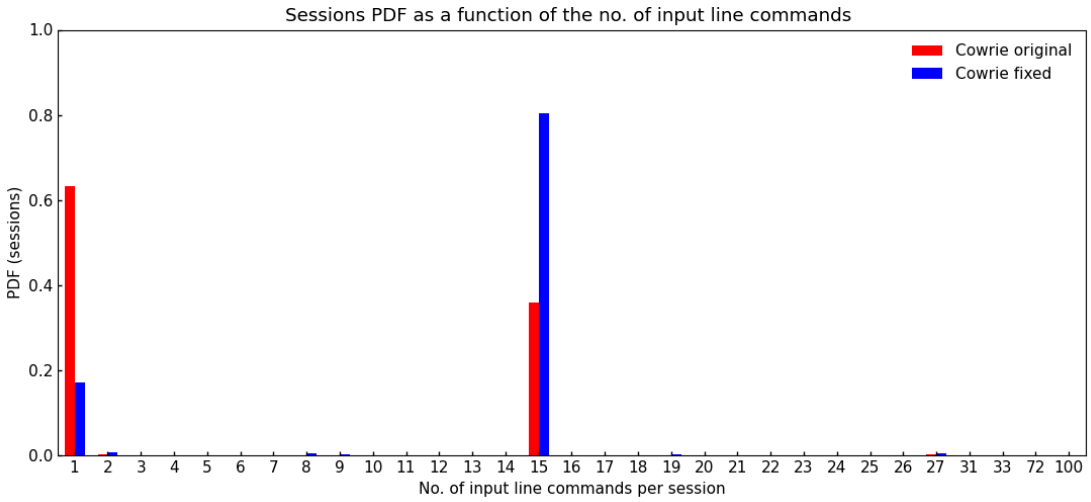

Figure 6.5: Probability Density Function of SSH sessions as a function of the number of input line commands

version of Cowrie registered 15 commands, while less than 40% of SSH sessions reached the same number with the original version.

In Figure 6.6, the number of sessions as a function of the number of input line commands is reported. The scale of the y-axis is logarithmic. We can observe a high difference in the number of sessions containing just one command between the original version of cowrie and the fixed one. The big discrepancies reported in Figure 6.4, especially concerning sessions with fifteen commands, here are not reported. However, this is due to the fact that, generally, the fixed version of Cowrie has more input line commands per session, meaning that the attacker that is not stuck on the first command, as in the original version of Cowrie, can continue the attack, increasing the number of commands per sessions and allowing the honeypot to collect more information. For this reason, as reported in Figure 6.4, it is more likely that the fixed version of Cowrie will receive a higher number of input line commands.

In tables Table 6.2 and Table 6.3 the four most popular first commands are reported, respectively for the original and fixed version of Cowrie. These are the commands that the attacker sends as soon as the SSH login phase is successfully completed. For each of them, the number of SSH sessions in which they were recorded is reported, helping quantify divergences among the number of sessions in which they were employed. As a matter of fact, the difference between the number of sessions in which the commands have been recorded is large, especially between the most popular one and the others. For what concerns the original version of

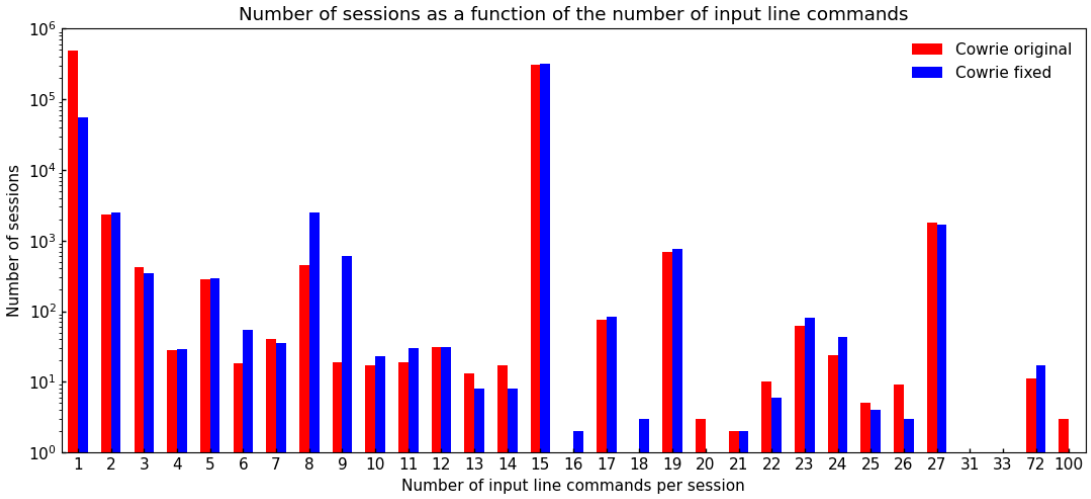

Figure 6.6: Number of sessions as a function of the number of input line commands

cowrie, the most popular command was sent 55% more with respect to the second most popular command and 23 more times than the third one. For what concerns the fixed version of Cowrie, the most popular command was sent 12 times more than the second one and 19 times more with respect to the third one.

Rank	**Cowrie original received commands**	**No. of SSH sessions**
1°	echo -e "\x6F\x6B"	366417
2°	cat /proc/cpuinfo | grep name | wc -l	235417
3°	uname -a	15800
4°	cd /tmp; rm -rf wget*; wget http://179.43.175.5/wget.sh; curl -O http://179.43.175.5/wget.sh; chmod 777 wget.sh; ./wget.sh server; sh wget.sh server	11938

Table 6.2: Most popular first input line commands received by the original version of Cowrie and the relative number of sessions in which they were recorded

In Table 6.2 the first command prints on the terminal the string that follows the command "echo". In this case, the string is written using the ASCII codes representing the letters "o" and "k". The "-e" parameter is used to tell the echo command to escape the backslash characters used to identify the ASCII bytes.

The second command tries to understand the technical specification of the computer on which is run, in particular, it reports the specifications of all the processors or cores installed on the machine. The command filters the previously obtained result by name, which yields the name of each processor installed. Then it uses the "wc" utility with the option "-l" to count the number of lines that were generated with the previous command steps. In this way, it is possible to retrieve how many processors are installed on the computer.

The third command "uname -a" is used to get the name of the operative system that the computer on which the command is executed is running. The "-a" option allows showing all the information about the OS such as the system version, machine ID, and the release number of the OS [1]. This information could be used to assess the system characteristics to understand how the machine can be exploited by using, for example, an exploit based on vulnerabilities that are typical of that specific OS version.

The fourth command is more complex than the previous one. First of all, it moves to the "tmp" directory. Then, it downloads a bash file from a specific IP address using two different tools: "wget" and "curl", it changes access permission with the "chmod" command followed by the code "777" which aims at granting execution, read and write privileges to groups, owner and public[2]. Then it executes the downloaded file passing the parameter "server".

For what concerns the commands reported in Table 6.3 for the fixed version of Cowrie, the first three correspond respectively to the second, third, and fourth most popular commands reported in Table 6.2 for the original version of Cowrie. The fourth command reported in Table 6.3 executes "uname -a", which behavior was previously described. Then it runs the "lspci" command which is used to gather information about PCI buses and connected devices if any [3]. The result is filtered and some words of interest are highlighted in color. Then, a request is sent to a specific IP address through the command line tool curl[4] and the response is saved into a file that is downloaded and executed with "perl".

By comparing Table 6.2 and Table 6.3, we can highlight that the most popular command in the original version of Cowrie is no more considered in the first four most common commands recorded by the fixed version. However, the other commands still remain in the ranking, shifting from one position toward the first place. Furthermore, the fourth command reported in Table 6.3 is the fifth most

[1] https://www.ibm.com/docs/en/aix/7.2?topic=u-uname-command

[2] https://www.ibm.com/docs/en/aix/7.2?topic=c-chmod-command

[3] https://man7.org/linux/man-pages/man8/lspci.8.html

[4] https://curl.se

Rank	Cowrie fixed received commands	No. of SSH sessions
1°	cat /proc/cpuinfo | grep name | wc -l	236688
2°	uname -a	19637
3°	cd /tmp; rm -rf wget*; wget http://179.43.175.5/wget.sh; curl -O http://179.43.175.5/wget.sh; chmod 777 wget.sh; ./wget.sh server; sh wget.sh server	12200
4°	uname -a;lspci | grep -i –color 'vga|3d|2d';curl -s -L http://39.165.53.17:8088/iposzz/dred -o /tmp/dred; perl /tmp/dred	4087

Table 6.3: Most popular first input line commands received for the fixed Cowrie version and the relative number of sessions in which they were recorded

popular command for the original Cowrie version. The number of sessions in which the same command is used in both versions is similar, making comparable the two cases. Furthermore, the fact that the most popular commands received by the two honeypots, except for the first one in the original Cowrie case, highlights how the two honeypots were subject to the same environment.

This finding is particularly important because it shows how the changes that we made to Cowrie impacted its performance. Since the only difference between the two versions is the order in which they send SSH messages, the fact that the most popular command in the original version is no more reported in the top four command ranking of the fixed one means that the changes that we made allowed the attacker to continue its task without remaining stuck on the first command. Given the high amount of sessions in which that command was sent, it is legitimate to assume that the smoothed trend reported in Figure 6.2 for the fixed version of Cowrie, is due to the lack of that command at the top of the ranking. However, that command is still present in 78 sessions.

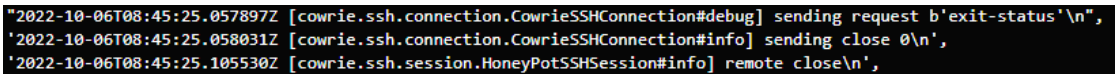

Figure 6.7: Extract from the log of the fixed version of Cowrie

Inspecting one of the sessions in which this command was sent to the fixed version of Cowrie, we noticed that once the command sent by the attacker is executed the connection closes without issues as shown in Figure 6.7. In the case

of the original version of Cowrie, once the honeypot responds to the command, the connection is lost as the attacker leaves without following the protocol standard as reported in Figure 6.8. This means that the way the original version of Cowrie answers to SSH messages from the attacker is not accepted by the attacker, as we saw in subsection 5.3.2.

```
"2022-10-06T00:00:01.156329Z [cowrie.ssh.connection.CowrieSSHConnection#debug] sending request b'exit-status'\n",
'2022-10-06T00:00:01.156458Z [cowrie.ssh.connection.CowrieSSHConnection#info] sending close 0\n',
"2022-10-06T00:00:01.224971Z [HoneyPotSSHTransport,822437,103.170.246.117] Got remote error, code 11 reason: b'Bye Bye'\n",
'2022-10-06T00:00:01.225227Z [cowrie.ssh.transport.HoneyPotSSHTransport#info] connection lost\n']
```

Figure 6.8: Extract from the log of the original version of Cowrie

6.3 Result review

In this case, it is not possible to employ all the metrics that were used in section 5.3 since we do not know in advance what were the goals of the attacks and the steps to achieve them. As a consequence, the attack steps score cannot be evaluated. For the same reason, we cannot evaluate the attack commands score. In a similar way, the signature parameter cannot be determined since we do not have data concerning real system behavior under the same conditions.

However, since the issue concerning the order in which messages are sent has been fixed, the signature parameter for the fixed version of Cowrie under the SSH login test is now decreased. Even if this modification does not affect the other metrics in that test, it does have a big effect on the collected data in the real deployment scenario. The fixed version of Cowrie performed better according to the received commands metric with respect to the original one, as highlighted in the data presented in section 6.2. As reported in Figure 6.5 and Figure 6.4, it is clear that the number of input line commands per session is higher for the fixed version of Cowrie with respect to the original one.

Thanks to the inspection that the data collected through T-Hon we were able to improve Cowrie effectiveness. As a matter of fact, this small modification to the honeypot allowed to remove from the collected data an important amount of commands that were not useful to gain information about how attacks were carried out.

Chapter 7

Conclusions and future work

The objective of this work was to design and develop a honeypot testing system. The system needed to be able to automatically perform attacks toward a group of victims. Then, the data from those attacks had to be collected to provide analysts with comprehensive information regarding how the victims responded to the deployed attacks. The focus of this data was the SSH connection protocol since this carries the input line commands sent by the attacker, which are of great interest to understand how malicious activities are performed. Furthermore, a set of metrics and parameters were proposed to evaluate and quantify the honeypots performance from the response fidelity point of view.

In order to meet the previously listed goals, the T-Hon system has been developed. T-Hon is an attack and data collection automation tool based on four blocks: the attacker, the proxy, the manager, and the victims. The four blocks work together under the coordination of the manager to select the victim to test, attack it, and collect the resulting data. The attacker block is based on the Metasploit Framework, a known tool in the penetration testing domain. It provides extensive support to the penetration testing activity. In the context of this work, we are interested in its complete library of exploits based on known vulnerabilities and its Remote Procedure Call Application Programming Interface. The library is useful to find attacks suitable to any test case and the RPC API is important to allow external interaction with the attack process and its automation. The proxy block was used to separate the data collection and the victim selection tasks from the other blocks. In this way, the proxy can be changed according to the kind of data that the test is focused on without affecting the other blocks. The proxy module is based on Cowrie in proxy mode deployed in a Docker container. The victims block regroup all the systems that will be the recipients of the attacks. They can be honeypots or real systems and they can be deployed as virtual machines, real machines or Docker containers. The manager block is a Python program that is in charge of

orchestrating all the tasks of the other blocks. It is set up through a configuration file that reports the attacks that have to be deployed, their specific settings, and the victims recipient of the attacks. At the end of the test session, T-Hon will provide a comprehensive JSON file reporting the logs and configurations of all the deployed attacks. The architecture of T-Hon allows adapting the system to the type of attacks, protocols, and victims that is required to inspect.

A set of attacks has been selected from the Metasploit library and three victims were selected: Cowrie in honeypot mode, Metasploitable, and Ubuntu. In this way, we could evaluate how T-Hon is able to provide useful data to assess how the honeypot behaves compared to real operative systems. We reported three case studies featuring three different attacks. In these studies, we analyzed the T-Hon collected data to highlight discrepancies in the different victim cases. After the data collection phase, we assessed the honeypot performance using the proposed metrics. Inspecting data from the attacker point of view, gave us a unique perspective of how the honeypot behaves. Furthermore, it provided us with the opportunity to discover some particular behaviors. In all three cases, we were able to identify issues that could be exploited to fingerprint the honeypot, as there are concerns regarding the way Cowrie manages the shell requests, how commands are executed and the SSH protocol implementation. Exploiting these issues, a malicious user could potentially recognize the honeypot among real systems, thus making it lose its deceiptability.

Thanks to the data collected by T-Hon and then analyzed in the study cases, we were able to identify honeypot issues and we decided to address one of them and fix it. Then, we inspected the changes in the collected data of the modified honeypot with respect to the original one that resulted to be significant. Among the issues identified in the study cases, we decided to solve the one regarding the implementation of the SSH protocol, specifically concerning the order in which the SSH connection protocol messages are sent. We applied modifications to the Cowrie software to make it behave similarly to the real systems that we tested. Then, the two versions of the honeypot, the fixed and the original one, were deployed in a real network for more than two months. Later, the data collected by both systems were analyzed and compared. Both versions of Cowrie were subject to the same environment. However, the fixed version of Cowrie performed much better in terms of the number of input line commands received per session, reducing in a considerable way the number of repeated attacks using the same input line command. This suggested that in the fixed version of Cowrie, attackers are able to execute more commands with respect to the other one. In the end, T-Hon proved to be a valuable tool to test honeypots and provide meaningful data that can be used to inspect their behaviors to gather precious insights and information.

Given the current development state of the T-Hon system, we propose some additional studies and future work that could allow to enhance the system capabilities. The data collected by T-Hon is focused on the SSH connection protocol. However, it would be interesting to perform the same kind of testing at a lower level inspecting the transport and authentication layer protocols. Furthermore, we propose to employ T-Hon, with the appropriate modifications, to inspect other types of honeypots, featuring different protocols. Another interesting feature that could be added to the system is an automated data analysis block, that could provide an evaluation of the proposed metrics after that all tests have been carried out. T-Hon was used with attacks identified in the Metasploit Framework library. We suggest to perform a deeper research for new attack scripts and to find a straightforward way to integrate them into the Metasploit Framework.

In this work, the data collected by T-Hon was used to identify odd behaviors in honeypot operations. As introduced in section 3.1, honeypots can be integrated with intelligent machine learning techniques that allow them to modify their behavior according to the way the attacker interacts with them. This kind of honeypot can learn from the received attacks and real system behavior how to respond to incoming interactions. In this way, we propose to adopt T-Hon as a training system, that can be employed to deploy several attacks toward intelligent honeypots in order to train them and, as a consequence, test their performance. The attacks could be the very same received during real environment deployments and resubmitted in controlled circumstances. This possible use of T-Hon could decrease the time needed to train such systems. Furthermore, it would be possible to employ ready-to-use attacks available in various libraries or by creating ad-hoc interactions, without waiting long periods of time for the collection of live data.

Appendix A

List of Linux SSH attacks

In this appendix, we report the list of SSH attacks that we selected for the scope of this work and available in the Metasploit Framework library [1] [2] reporting their description[3].

Table A.1: List of SSH Linux attacks

Begin of Table	
Name	Description
ceragon fibeair known privkey	Ceragon ships a public/private key pair on FibeAir IP-10 (FibeAir IP-10 is Ceragon's next-generation carrier-grade wireless Ethernet solutions family) devices that allow passwordless authentication to any other IP-10 device. Since the key is easily retrievable, an attacker can use it to gain unauthorized remote access as the "mateidu" user.

[1] github.com/rapid7/metasploit-framework/tree/master/modules/exploits/linux/ssh

[2] github.com/rapid7/metasploit-framework/tree/master/modules/auxiliary/scanner/ssh

[3] https://www.infosecmatter.com

Continuation of Table A.1

Name	Description
exagrid known privkey	ExaGrid (Tiered Backup Storage) ships a public/private key pair on their backup appliances to allow passwordless authentication to other ExaGrid appliances. Since the private key is easily retrievable, an attacker can use it to gain unauthorized remote access as root. Additionally, this module will attempt to use the default password for root, 'inflection'.
cisco ucs scpuser	This module abuses a known default password on Cisco UCS Director. The 'scpuser' has the password of 'scpuser', and allows an attacker to log in to the virtual appliance via SSH. This module has been tested with Cisco UCS Director virtual machines 6.6.0 and 6.7.0. Note that Cisco also mentions in their advisory that their IMC Supervisor and UCS Director Express are also affected by these vulnerabilities, but this module was not tested with those products.
ibm drm a3user	This module abuses a known default password in IBM Data Risk Manager. The 'a3user' has the default password 'idrm' and allows an attacker to log into the virtual appliance via SSH. This can be escalated to full root access, as 'a3user' has sudo access with the default password. At the time of disclosure, this was an 0-day, but it was later confirmed and patched by IBM. Versions <= 2.0.6.1 are confirmed to be vulnerable.
loadbalancerorg enterprise known privkey	Loadbalancer.org ships a public/private key pair on Enterprise virtual appliances version 7.5.2 that allows passwordless authentication to any other LB Enterprise box. Since the key is easily retrievable, an attacker can use it to gain unauthorized remote access as root.
mercurial ssh exec	Mercurial is a distributed revision control tool. This module takes advantage of custom hg-ssh wrapper implementations that don't adequately validate parameters passed to the hg binary, allowing users to trigger a Python Debugger session, which allows arbitrary Python code execution.

Continuation of Table A.1

Name	Description
microfocus obr shrboadmin	This module abuses a known default password on Micro Focus Operations Bridge Reporter. The 'shrboadmin' user, installed by default by the product has the password of 'shrboadmin' and allows an attacker to log in to the server via SSH. This module has been tested with Micro Focus Operations Bridge Manager 10.40. Earlier versions are most likely affected too. Note that this is only exploitable in Linux installations.
quantum dxi known privkey	Quantum ships a public/private key pair on DXi V1000 2.2.1 (appliances for backup) appliances that allow passwordless authentication to any other DXi box. Since the key is easily retrievable, an attacker can use it to gain unauthorized remote access as root.
quantum vmpro backdoor	This module abuses a backdoor command in Quantum vmPRO (software for VM backup). Any user, even one without admin privileges, can get access to the restricted SSH shell. By using the hidden backdoor "shell-escape" command it's possible to drop to a real root bash shell. This module has been tested successfully on Quantum vmPRO 3.1.2.
solarwinds lem exec	This module exploits the default credentials of SolarWinds LEM (security information and event management system). A menu system is encountered when the SSH service is accessed with the default username and password which is "cmc" and "password". By exploiting a vulnerability that exists on the menuing script, an attacker can escape from a restricted shell. This module was tested against SolarWinds LEM v6.3.1.
symantec smg ssh	This module exploits a default misconfiguration flaw on Symantec Messaging Gateway. The 'support' user has a known default password, which can be used to login to the SSH service, and gain privileged access from remote.

Continuation of Table A.1	
Name	Description
vmware vdp known privkey	VMware vSphere Data Protection (backup and restore) appliances 5.5.x through 6.1.x contain a known ssh private key for the local user admin who is a sudoer without a password.
vyos restricted shell privesc	This module exploits command injection vulnerabilities and an insecure default sudo configuration on VyOS (open-source network OS) versions 1.0.0 <= 1.1.8 to execute arbitrary system commands as root. VyOS features a restricted-shell system shell intended for use by low-privilege users with operator privileges. This module exploits a vulnerability in the telnet command to break out of the restricted shell, then uses sudo to exploit a command injection vulnerability in /opt/vyatta/bin/sudo-users/vyatta-show-lldp.pl to execute commands with root privileges. This module has been tested successfully on VyOS 1.1.8 amd64 and VyOS 1.0.0 i386.
f5 bigip known privkey	F5 ships a public/private key pair on BIG-IP appliances that allow passwordless authentication to any other BIG-IP box. Since the key is easily retrievable, an attacker can use it to gain unauthorized remote access as root.
ssh login	This module will test ssh logins on a range of machines and report successful logins. If you have loaded a database plugin and connected to a database this module will record successful logins and hosts so you can track your access.
End of Table	

Bibliography

[1] Adrienne LaFrance. *Cyberwar Is Officially Crossing Over Into the Real World*. May 2017. URL: https://www.theatlantic.com/technology/archive/2017/05/cyberwar-is-officially-crossing-over-into-the-real-world/526860/ (cit. on p. 1).

[2] *Cost of a Data Breach Report 2021*. Tech. rep. IBM Corporation New Orchard Road Armonk, NY 10504: IBM Corporation, July 2021 (cit. on p. 1).

[3] *Cisco Annual Internet Report (2018–2023)*. Tech. rep. Chesney House, 96 Pitts Bay Road, Pembroke HM 08, Bermuda: Cisco, Mar. 2020 (cit. on p. 1).

[4] *The McAfee Consumer Mobile Threat Report*. Tech. rep. 6220 America Center Drive, San Jose, CA 95002: McAfee, Feb. 2022 (cit. on p. 1).

[5] Wenjun Fan, Zhihui Du, David Fernandez, and Victor A Villagra. «Enabling an Anatomic View to Investigate Honeypot Systems: A Survey». eng. In: *IEEE systems journal* 12.4 (2018), pp. 3906–3919 (cit. on pp. 1, 2, 7–10).

[6] Javier Franco, Ahmet Aris, Berk Canberk, and A. Selcuk Uluagac. «A Survey of Honeypots and Honeynets for Internet of Things, Industrial Internet of Things, and Cyber-Physical Systems». eng. In: *IEEE Communications surveys and tutorials* 23.4 (2021), pp. 2351–2383 (cit. on pp. 2, 8, 9, 11).

[7] Cowrie. *Cowrie Documentation*. Oct. 2022. URL: https://cowrie.readthedocs.io/en/latest/index.html (cit. on pp. 2, 11).

[8] Ahmad Salah Al-Ahmad, Hasan Kahtan, Fadhl Hujainah, and Hamid A. Jalab. «Systematic Literature Review on Penetration Testing for Mobile Cloud Computing Applications». In: *IEEE Access* 7 (2019), pp. 173524–173540. DOI: 10.1109/ACCESS.2019.2956770 (cit. on p. 2).

[9] Mohd Nizam Zakaria, Poon Ai Phin, Nurfarahin Mohmad, Saiful Adli Ismail, Mohd Nazri Kama, and Othman Yusop. «A Review of Standardization for Penetration Testing Reports and Documents». In: *2019 6th International Conference on Research and Innovation in Information Systems (ICRIIS)*. 2019, pp. 1–5. DOI: 10.1109/ICRIIS48246.2019.9073393 (cit. on pp. 2, 12, 13).

[10] OffSec Services Limited. *Metasploit unleashed*. Nov. 2022. URL: https://www.offensive-security.com/metasploit-unleashed/ (cit. on p. 3).

[11] Christos Dalamagkas, Panagiotis Sarigiannidis, Dimosthenis Ioannidis, Eider Iturbe, Odysseas Nikolis, Francisco Ramos, Erkuden Rios, Antonios Sarigiannidis, and Dimitrios Tzovaras. «A Survey On Honeypots, Honeynets And Their Applications On Smart Grid». In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. 2019, pp. 93–100. DOI: 10.1109/NETSOFT.2019.8806693 (cit. on p. 4).

[12] Solomon Z. Melese and P.S. Avadhani. «Honeypot System for Attacks on SSH Protocol». eng. In: *International journal of computer network and information security* 8.9 (2016), pp. 19–26. ISSN: 2074-9090 (cit. on p. 5).

[13] Jurgen Schonwalder, Georgi Chulkov, Elchin Asgarov, and Mihai Cretu. «Session resumption for the secure shell protocol». In: *2009 IFIP/IEEE International Symposium on Integrated Network Management*. 2009, pp. 157–163. DOI: 10.1109/INM.2009.5188805 (cit. on pp. 5, 10).

[14] Melike Baser, Ebu Yusuf Guven, and Muhammed Ali Aydin. «SSH and Telnet Protocols Attack Analysis Using Honeypot Technique: Analysis of SSH AND TELNET Honeypot». eng. In: IEEE, 2021, pp. 806–811 (cit. on pp. 5, 10).

[15] *The Secure Shell (SSH) Connection Protocol*. The Internet society, Jan. 2006. URL: https://www.rfc-editor.org/info/rfc4254 (cit. on pp. 5, 37, 42–44, 48, 51).

[16] Jason M Pittman, Kyle Hoffpauir, Nathan Markle, and Cameron Meadows. «A Taxonomy for Dynamic Honeypot Measures of Effectiveness». eng. In: (2020) (cit. on pp. 6, 7, 31).

[17] Jason M Pittman, Kyle Hoffpauir, and Nathan Markle. «Primer – A Tool for Testing Honeypot Measures of Effectiveness». eng. In: *arXiv.org* (2020). ISSN: 2331-8422 (cit. on p. 7).

[18] A Vetterl and R Clayton. «Bitter harvest: Systematically fingerprinting low- and medium-interaction honeypots at internet scale». eng. In: 12th USENIX Workshop on Offensive Technologies, WOOT 2018, co-located with USENIX Security 2018, 2018 (cit. on pp. 7, 33).

[19] A Vetterl, R Clayton, and I Walden. «Counting outdated honeypots: Legal and useful». eng. In: Proceedings - 2019 IEEE Symposium on Security and Privacy Workshops, SPW 2019, 2019 (cit. on p. 7).

[20] L. Spitzner. «The Honeynet Project: trapping the hackers». In: *IEEE Security & Privacy* 1.2 (2003), pp. 15–23. DOI: 10.1109/MSECP.2003.1193207 (cit. on pp. 8, 9).

[21] Lukas Zobal, Dusan Kolar, and Radek Fujdiak. «Current State of Honeypots and Deception Strategies in Cybersecurity». eng. In: *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. Vol. 2019-. IEEE, 2019, pp. 1–9. ISBN: 9781728157634 (cit. on pp. 9–11).

[22] Michail Tsikerdekis, Sherali Zeadally, Amy Schlesener, and Nicolas Sklavos. «Approaches for Preventing Honeypot Detection and Compromise». In: *2018 Global Information Infrastructure and Networking Symposium (GIIS)*. 2018, pp. 1–6. DOI: 10.1109/GIIS.2018.8635603 (cit. on p. 10).

[23] *Assessing Security and Privacy Controls in Information Systems and Organization*. Tech. rep. Gaithersburg: NIST, Jan. 2022 (cit. on pp. 11–13).

[24] Georgia Weidman. *Penetration Testing*. eng. No Starch Press, 2014. ISBN: 1-59327-564-1 (cit. on p. 11).

[25] *Common Vulnerabilities and Exposures (CVE) Numbering Authority (CNA) Rules*. Common Vulnerabilities and Exposures Program. 2016 (cit. on p. 11).

[26] Kevin Allen Lee; Cardwell. *Advanced penetration testing for highly-secured environnements : employ the most advenced pentesting techniques and tools to build highly-secured systems and environnements*. eng. Packt Publishing, 2016. ISBN: 1-78439-581-1 (cit. on p. 12).

[27] Abhinav Singh, Nipun Jaswal, Monika Agarwal, and Daniel Teixeira. *Metasploit Penetration Testing Cookbook: Evade Antiviruses, Bypass Firewalls, and Exploit Complex Environments with the Most Widely Used Penetration Testing Framework, 3rd Edition*. eng. Birmingham: Packt Publishing, Limited, 2018. ISBN: 9781788623179 (cit. on pp. 13–15).

[28] OffSec Services Limited. *Metasploit unleashed*. Nov. 2022. URL: https://www.offensive-security.com/metasploit-unleashed/ (cit. on pp. 14, 30).

[29] Rapid7. *Metasploit Documentation*. Oct. 2022. URL: https://docs.metasploit.com (cit. on pp. 14, 15, 18).

[30] HelpSystems. *Core Security*. Oct. 2022. URL: https://www.coresecurity.com/products/core-impact#resources (cit. on p. 18).

[31] PortSwigger Ltd. *PortSwigger*. Oct. 2022. URL: https://portswigger.net/burp (cit. on p. 18).

[32] Truelist. *Linux Statistics*. Oct. 2022. URL: https://truelist.co/blog/linux-statistics/ (cit. on p. 29).

[33] Cisco. *UCS director*. Oct. 2022. URL: www.cisco.com/c/en/us/products/servers-unified-computing/ucs-director/index.html (cit. on p. 35).

[34] National Insitute of Standards and Technology. *National Vulnerability Database*. Oct. 2022. URL: `https://nvd.nist.gov` (cit. on p. 35).

[35] *Quantum vmPRO User's Guide*. Quantum Corporation. 2015 (cit. on p. 45).

[36] F. Soro, I. Drago T. Rescio, D. Giordano M. Mellia, Z. B. Houidi T. Favale, and D. Rossi. «Enlightening the Darknets: Augmenting Darknet Visibility with Active Probes». In: (2021) (cit. on p. 52).