Master of Science in Computer Engineering

Master Degree Thesis

Attack propagation and response in digital service chains

Supervisors:
Prof. Fulvio Valenza

Candidate:
Ten. Camilla Piccini

Co-Supervisors:
Dott. Daniele Bringhenti
Ing. Matteo Repetto

Anno Accademico 2023/2024

Abstract

This thesis deals with the detection of multi-step attack and strategies to mitigate them within Digital Service Chains (DSCs), a framework increasingly adopted by Digital Service Providers (DSPs) to integrate various digital components like IoT, software and cloud infrastructure. While DSCs offer significant advantages in terms of scalability and flexibility, they also introduce security challenges, making difficult to isolate and mitigate threats.

To address these challenges, this work proposed an innovative approach based on TAMELESS (Threat & Attack ModEL Smart System) framework. TAMELESS is designed to detect threats by determine if a system is secure through the study of its components and their security properties.

An important element of this thesis is the inclusion of Common Vulnerabilities and Exposure (CVEs) within the multi-step attack scenario. The work also proposes a threat model for CVEs that can be used to automate the analysis of vulnerabilities in the TAMELSS framework.

The modifications made to TAMELESS involve focus exclusively on the cyber aspect of threats, integrating CVEs to reinforce vulnerability analysis and add functionalities for managing patches. The patch management ensure that once vulnerabilities are identified and patched, they are neutralized, reducing the risk of threats.

The obtained result demonstrates the efficiency of the modified TAMELESS framework in identifying threats, analyzing attack patch and mitigating the risk associated with multi-step attacks.

This work provides an advanced approach to make DSCs more secure in an interconnected environment, offering a robust framework for future developments in threat detection and response strategies.

Acknowledgements

Questa tesi segna la fine di un lungo ed impegnativo percorso di studio che ha portato alla realizzazione di molti obbiettivi sia di crescita personale che professionale.
Per il raggiungimento di questo traguardo vorrei ringraziare, innanzitutto, il mio relatore, il Prof. Fulvio Valenza, per la sua competenza e la sua grande disponibilità in tutto il percorso. Vorrei ringraziare anche i correlatori, il Dott. Daniele Bringhenti e l' Ing. Matteo Repetto, per il loro costante supporto e i loro ingenamenti ed aiuti durante tutto il lavoro di tesi.
Vorrei poi ringraziare tutta la mia famiglia, in particolare mia mamma Erika, mio babbo Massimiliano e mia sorella Arianna che mi hanno sempre supportato in questi anni di duro lavoro, rendendo anche i momenti peggiori più leggeri. Mi hanno sempre fatto sentire la loro presenza nonostante la distanza fisica e di questo gli sono molto grata.
Ringrazio anche i miei nonni, i miei zii, mio cugino e tutte le persone che mi hanno dimostrato la loro vicinanza in questi anni.
Infine l'ultimo ringraziamento va al mio fidanzato Christian, che mi è sempre stato vicino anche nei momenti di sconforto, aiutandomi a credere in me stessa e nelle mie capacità.

Contents

List of Figures **6**

1 Introduction **9**

1.1 Document Structure 10

2 Backgroud **13**

2.1 Digital Service Chains (DSC) 13

2.2 Multi-step attack 15

2.3 Common Vulnerability Exposure (CVE) 17

2.4 Attack graph 20

2.5 REST API 22

3 Threat Modelling **25**

3.1 Example of multi-step attack 25

3.2 Analisys of involved CVEs 28

3.3 Threat description 30

3.3.1 Structure 30

3.3.2 Implementation 32

4 TAMELESS **35**

4.1 Hybrid Threat Model 36

4.2 Code 38

4.2.1 Rule.p 38

4.2.2 System.p 39

4.3 RESTful Web Service in TAMELESS 40

4.3.1 SystemService.java 41

5 Thesis overwiev **45**

5.1 Thesis objectives 45

5.2 Phases of Work 46

6 Optimization of TAMELESS **49**

6.1 Modification to Focus on Cyber Domain 49

6.2 Inclusion of CVEs 52

6.3 Patch Managment 54

7 Validation **57**

7.1 Validation of Cyber-focused Rule Modifications 58

7.2 Validation of CVEs-focused Rules 63

7.3 Validation of Rules focused on Patch 68

8 Conclusion **75**

8.1 Future works 76

Bibliography **77**

List of Figures

2.1 An example of DSC [1] 13
2.2 An example of multi-step attack [2] 16
2.3 The CVE Lifecycle 18
2.4 Attack Graph 21
2.5 Generation Methodologies 22

4.1 TAMELESS workflow 35

7.1 Attack Graph Example 1 64
7.2 Attack Graph Example 2 68
7.3 Attack Graph Example 3 73

Chapter 1

Introduction

The evolution of digital technologies has radically changed the way digital service providers (DSPs) work. As digital technology advances and connections between various components grow, Digital Service Providers (DSPs) are increasingly using Digital Service Chains (DSCs) to connect different digital services to provide solutions to new business needs. They increasingly adopt external services provided by third-party providers, creating these Digital Service Chains (DSCs) that offer significant opportunities in terms of scalability, flexibility, and innovation.

DSCs are creating by combining various digital components like IoT devices, software, cloud/edge infrastructures and services, which interact to provide integrated solutions.

The interconnection between services introduce new cybersecurity challenges. For example they makes difficult to isolate and mitigate threats, as an attack on a single component can propagate through the entire service chain.

Advanced attack techniques, like lateral movement, exploit these interconnections to penetrate deeper into systems and access critical resources.

These attacks are focused on the more vulnerable targets and less protected components in DSCs, which are easier to exploit. Once attackers have successfully reached these initial targets, they move laterally through the network, moving from one compromised components to another. Through this process attacker finally reach the primary target, which is more protected.

Traditional security solutions are no longer sufficient to effectively defend against increasingly sophisticated and multi-phase attacks. An integrated approach is necessary, combining advanced detection techniques, automated response, and threat intelligence

management, also taking into account the sequential nature of attacks.
Protecting digital service chains requires a new paradigm of protection that is adaptive, integrated, and capable of addressing the complex challenges of a multi-tenant and interconnected environment. [3]
The aim of this thesis work is to study and analyze an existing framework and modify it to make more functional in this new interconnected environment, focusing only on cyber aspect and involving CVEs (Common Vulnerabilities and Exposures).
The analyzed framework is TAMELESS, which is already optimized in threat analysis but not in this new environment.

1.1 Document Structure

This thesis has been divided into following chapters:

- **Chapter 2**: covers the theory behind the thesis, in particular Digital Service Chain (DSC), Multi-Step Attack, Common Vulnerability Exposure (CVE) and Attack Graph. We have also introduced some theory about REST that can be usefull for some future works described in the last chapter.
- **Chapter 3**: focuses on threat modelling, including the elaboration of three multi-step attack examples, analysis of involved CVEs and the development of a threat description useful in future works .
- **Chapter 4**: studies TAMELESS, software that has been analysed and modified during thesis development. There is the description of the most important files that we have analyzed during our work.
- **Chapter 5**: outlines the goals of the thesis and development methodology, so the phase of work.
- **Chapter 6**: describes the changes made on TAMELESS to implement to achieve thesis objective. This chapter is divided into three sections, each of them represent one goals achieved in the thesis.
- **Chapter 7**: presents the validation of all the implementations described in the previous chapter. Also this chapter is divided into three section, each focused on one goals of the thesis.

- **Chapter 8**: is the final chapter that concludes the thesis showing the entire work and discussing possible future works.

Chapter 2

Backgroud

2.1 Digital Service Chains (DSC)

Digital Service Chains (DSCs) is composed by a series of interconnected digital services and components that work together to provide complete solutions to complex business requirements. These services are provided by different Digital Service Providers (DSPs) and communicate to each other using standardized protocols for web services.
One common use of DSCs is in Smart Building or Smart Cities. Here, various devices, infrastructures and applications, managed by different provider, work together to offer different services.[3]
In figure 2.1 we can see an example of DSC, in particular of a smart cities.

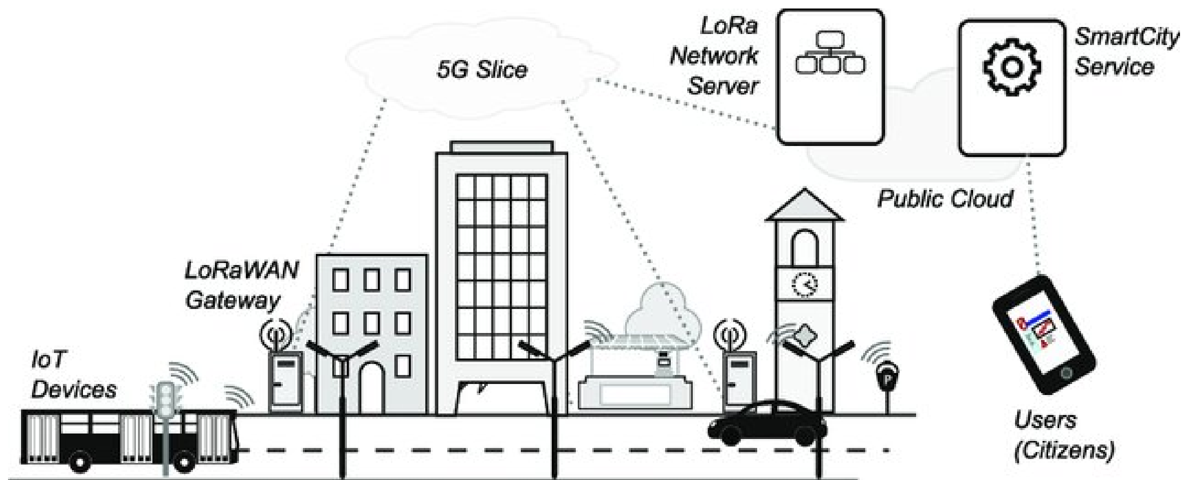

Figure 2.1: An example of DSC [1]

The complexity of modelling DSC comes from their structure, in particular the various interactions between applications and infrastructures that are managed by different owners. The factors contributing to this complexity include:

- Heterogeneity: DSCs comprehends many services and technologies each with each own protocols and standards.
- Flexible Composition: the ability to dynamically compose and recompose services adding layers.
- Multi-Ownership: different components are often owned and managed by different entities.
- Dynamic Topologies: the structure of DSCs can change frequently due to replication, scaling, or migration.[3]

Challenges to Protect Digital Service Chains

The run-time detection of threats and attacks in Digital Service Chains presents several challenges.
One of the main issues is fragmented operations. The multi-ownership nature of DSCs limits visibility into each subsystem, making difficult to correlate multi-stage attacks across different domains and identify the entire attack chain, resulting in a focus on single attacks and not on the chain.
Additionally, the dynamic service topologies make threat detection more difficult. Operations such as scaling, replication, migration and service replacement frequently change the composition and topology of services. This reduces the effectiveness of detection algorithms that rely on static configurations, causing gaps in monitoring and security coverage.
Multi-ownership and externalization reduce privacy and confidentiality. As data and resources are shared among various entities, implementing dynamic trust mechanisms and strict access controls is necessary to protect data while facilitating collaboration among different Digital Service Providers.
The adoption of lightweight environments, such as containers and serverless computing makes more difficult the deployment of traditional security devices. Without an abstraction layer it becomes challenging to implement uniform monitoring and response policies.

Finally, evolving attack patterns are a significant threat. The dynamic configurations of DSCs make them especially vulnerable to adaptive attack strategies, highlighting the need for more advanced detection and response mechanisms.[3]
In summary, while Digital Service Chains offer numerous benefits in terms of flexibility and functionality, they also introduce substantial challenges in terms of security and management. All the challenges described before create a complex security landscapes and this new scenario requires new advanced security mechanism to ensure protection to DSCs.

2.2 Multi-step attack

Multi-step attack represents an advanced and sophisticated type of cyber threat, which is characterized by a series of organized steps focused on compromising a target system. They exploit multiple vulnerabilities across various systems and are designed to bypass traditional security measures by unfolding gradually and across different segments of the network.
Multi-step attacks are particularly prevalent in digital service chains due to their complex and layered nature, as we have introduce in section 2.1. In DSCs the interconnections between services and systems create a larger and more heterogeneous attack surface, offering attackers numerous entry points and pathways for propagation.
A typical multi-step attack include the following steps:

- Initial Compromise: gain initial access to the target system, normally starting by accessing the most vulnerable link and then move to more important target. An attacker can use different techniques such as phishing, exploiting software vulnerabilities or using social engineering techniques.

- Establish Foothold: maintain persistent access to the compromised system through installation of malware or backdoors.

- Privilege Escalation: gain higher-level access permission exploiting privilege escalation vulnerabilities or misconfigurations.

- Internal Reconnaissance: study and understand the network layout and locate high-value targets, perform this by mapping the network and identifying additional targets.

- Lateral Movement: move across the network in order to reach the final target, such as database or critical infrastructure. Techniques used to do this depend on the previous step and on the target, for example it is possible to use compromised credentials.
- Achieving the Object: this is the final step, so fulfill the attacker's primary intent, which can be stealing sensitive data, deploying ransomware, or disrupting services.

Each of these steps highlights the precision required to execute such attacks effivciently and demonstrates the methodical approach attackers take to penetrate and exploit target networks.[3]
These attacks are designed to avoid traditional security measures by exploiting vulnerabilities at different stages and this make the detection and prevention challenging. It will be necessary to implement various defensive strategies compared to traditional attacks. These may include deploying defense-in-depth strategies to provide multiple layers of protection or using machine learning systems and anomaly detection to identify unusual activities that could indicate a multi-step attack. We can see an example of multi-step attack in Figure 2.2.

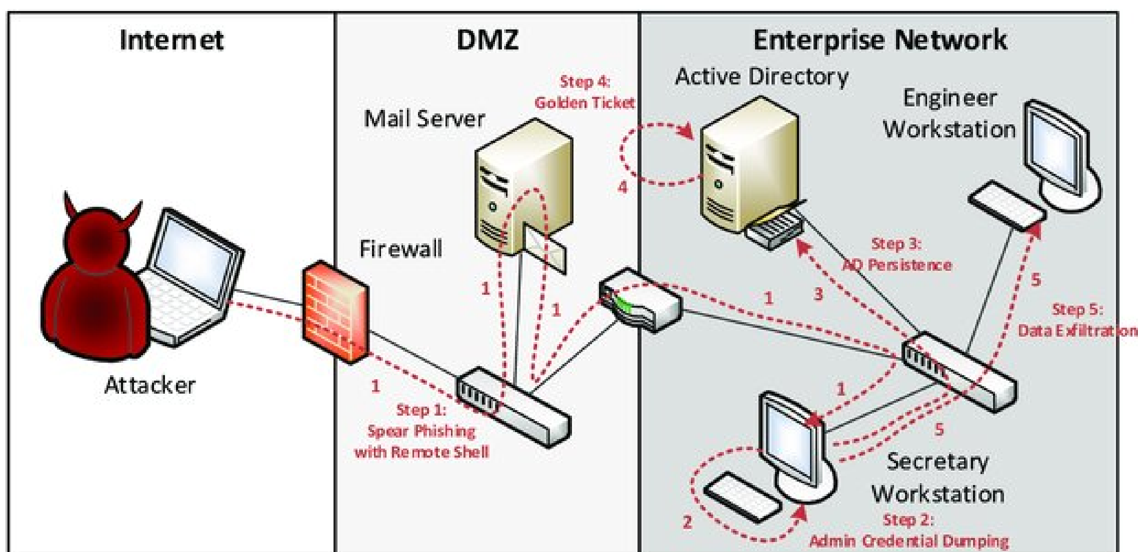

Figure 2.2: An example of multi-step attack [2]

The attack in figure is composed by seven different steps:

1. The attacker first sends a phishing email to the secretary of an employee. He can find the secretary's email address on internet.

2. When the secretary opens the phishing email, a malware is executed on their computer. This allow the attacker to establish a remote shell on the secretary's computer and he gains access to the device and the company's network.

3. After, the attacker uses a spear-phishing technique to convince a network administrator to log into the compromised secretary's computer.

4. The attacker now manages to obtain the domain credentials of the network administrator.

5. The attacker can use gained credential to get access to company's central server. This enables the attacker to issue a "golden ticket", a type of authentication token.

6. The golden ticker permits the attacker to gain access to the workstation of an engineer within the company.

7. Finally, through the compromised engineer's workstation, the attacker obtain access to confidential document of the company. [2]

In conclusion, multi-step attacks represent an enormous challenge in cybersecurity due to their advanced and sophisticated nature. Characterized by a series of organized steps, these attacks exploit multiple vulnerabilities across different systems, progressing gradually and stealthily to elude traditional security measures. Their complexity is further amplified within digital service chains. Understanding the complex nature of multi-step attacks is crucial for recognizing the significant threat they represent.

2.3 Common Vulnerability Exposure (CVE)

The Common Vulnerabilities and Exposure (CVE) is a database of public disclosed vulnerabilities and exposures developed to find a unique catalogue of information. The difference between vulnerabilities and exposures is that the former affects computer software, firmware, and hardware, while the latter are errors not related to software or other components, but rather a misconfiguration, such as open ports or weak credentials.
The CVE database is managed by National Cybersecurity FFRDC (Federally Funded Research and Development Center) and operated MITRE Corporation, a US no-profit organization.[4]

In Figure 2.3 we can see the steps through how CVEs are discovered, recorded, submitted and finally published. In particular it follows the following steps:

1. Vulnerabilities and exposures are first discovered by someone, such as a user or a researcher.

2. After a vulnerability is discovered, it is reported to a CVE Program participant.

3. The CVE Program participant requests a CVE identifier (CVE ID).

4. A CVE ID is reserved by a CNA (CVE Numbering Authority), which assigns the ID and creates a Recrod.

5. At this point, the vulnerability is not yet made public; first an investigation is conduced, and the CVE Program participant submits details about the CVE.

6. Finally, the CVE is published and made available for download.[4]

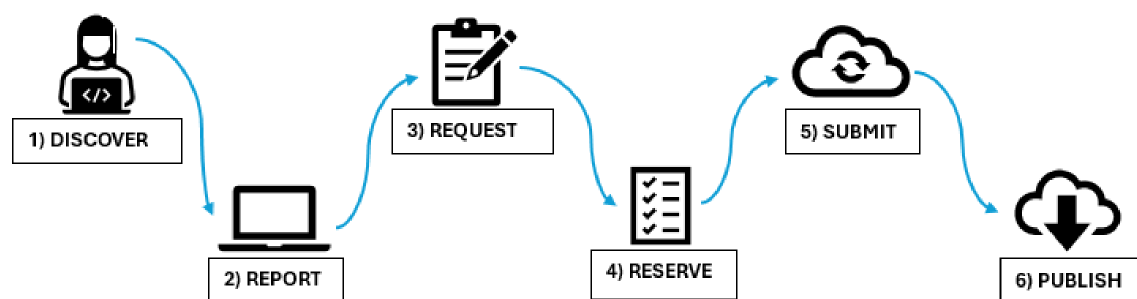

Figure 2.3: The CVE Lifecycle

The CVE system plays a fundamental role in the cybersecurity landscape, providing a consistent and reliable naming convention for vulnerabilities that is widely used by organizations.
Here are the main reason of the importance of CVE:

- **Standardization**: provides a standardized method for identifying vulnerabilities, permitting organizations to share and compare information across different platforms and tools. Cybersecurity experts can better coordinate their responses and communicate about threats when they have a standard vocabulary for vulnerabilities.

- **Coordination**: by using CVE identifiers it possible for diverse security tools and services to integrate more effectively, enhancing coordination in vulnerability management and facilitating system interoperability.

- **Awareness**: CVE entries raise awareness of vulnerabilities, helping organizations to prioritize their security efforts according to recognized problems. By organizing and publicizing these issues, CVE system assists organizations in staying informed about emerging dangers by organizing and disseminating information about these topics.
- **Mitigation**: by identifying and understanding CVEs, organizations can apply appropriate patches and mitigation strategies to protect their systems from attacks.

Contents of CVE Record

The CVE system provides a standard naming scheme for vulnerabilities, which facilitates communication among cybersecurity services. It is also helpful as a starting point for evaluating the effectiveness of security tools, helping users determine which ones are more appropriate for the organization's requirements.

Element	**Description**
CVE Identifier	CVE-2021-44228
Description	Apache Log4j2 JNDI features do not protect against attacker controlled LDAP and other JNDI related endpoints
CWE	CWE-400 Uncontrolled Resource Consumption CWE-502 Deserialization of Untrusted Data CWE-20 Improper Input Validation
CVSS	Critical
Affected Versions	Apache Log4j 2.x versions from 2.0-beta9 to 2.14.1

Table 2.1: CVE basic structure

Each entry in CVE system has a unique identifier number, such as CVE-2024-44228, which is used as a numeric marker for a particular vulnerability. With this identifier, each entry provides a brief description of the vulnerability, detailing its characteristic and potential impact. Additionally, important references are included to provide a comprehensive understanding of the issue, such as vulnerability reports, advisories, and other relevant documentation, such as CWE (Common Weakness Enumeration) which represents weaknesses.[5]

A description is also provided, including the vulnerability's type, root cause and the impact it could have. Another important element is the severity, which is measured using the Common Vulnerability Scoring System (CVSS).
The CVE record also includes exploitability metrics, which represent the ways through which the vulnerability can be exploited. There is also the scope, which refers to the possibility that one vulnerability could affect another piece of software or component. The record can also include impact metrics, which measure the potential effects of a successful exploitation of the vulnerability.
Finally there are references related to the vulnerability and affected versions of software or components, helping users verify if they are using vulnerable version. [4] We can see an example of CVE structure in Table 2.1 .

2.4 Attack graph

An attack graph is a model-based method for analyzing network security. It maps out potential attack paths and their interdependencies within a network, helping to identify vulnerabilities and understand how they might be exploited.
Each node in an attack graph represents a system state or condition, such as a vulnerability or an attack step, while edges represent the transitions between these states caused by exploits. This graphical representation is crucial for understanding the sequences of exploits an attacker could use to compromise a system, enabling more effective defense strategies.[6] A basic example of an attack graph can be see in Figure 2.4.

The process of generating attack graphs involves several key steps. Initially, it requires the input of information about the network, including its topology, configurations, and any known vulnerabilities. This information forms the basis for the subsequent steps.
Once input data are collected, algorithms are used to simulate potential attack scenarios. This simulation involves iteratively applying known exploits to the network configuration to explore all possible ways an attacker could reach through the system. The result of this process is a graph that visually and analytically represents the various paths an attacker might take. This graph helps in identifying critical points where security measures can be most effective. This graph is then analysed to gain several information such as number of attack paths, minimum length of attack paths and essential exploit for the attacker.[6]
The generation of attack graphs can be approached through various methodologies, each

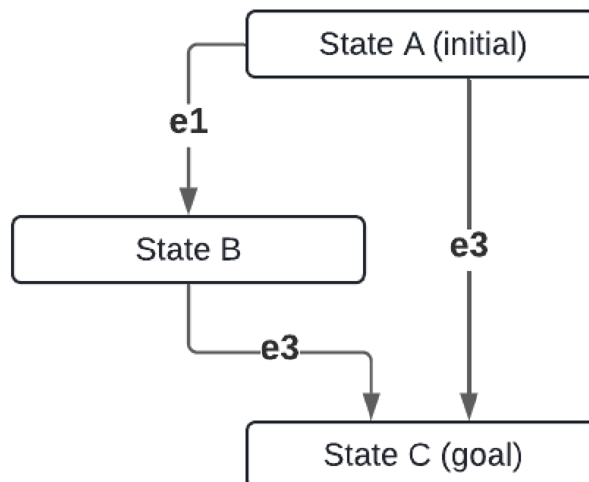

Figure 2.4: Attack Graph

with its own set of techniques and advantages.One of this methodology is the model-driven approach, which begins with system descriptions and applies transformation rules to convert these descriptions into attack models. This approach rely on predefined transformation rules to create attack graphs from system models.
Another methodology is the analysis-driven approach, which starts with a system description and focuses on performing security analyses such as model checking or vulnerability scanning. The outcomes of these analyses are then used to generate the attack graphs.
The third approach is vulnerability-driven, which begins with collecting data on vulnerabilities, often sourced from vulnerability databases and use data to create attack graphs by matching known vulnerability patterns to the system.[7]
Each of these methodologies offers a distinct way of approaching the creation of attack graphs but they are not disjoint as we can see in Figure 2.5.
Attack graphs are used for various purposes, including:

- Risk Analysis: identify and prioritize vulnerabilities based on their roles in potential attack path.
- Network Hardening: develop strategies to mitigate risks by reinforcing critical points in the network.
- Incident Response: analyze the steps taken by an attacker to improve defenses.

In conclusion, attack graphs are a valid method for cybersecurity experts to understand and

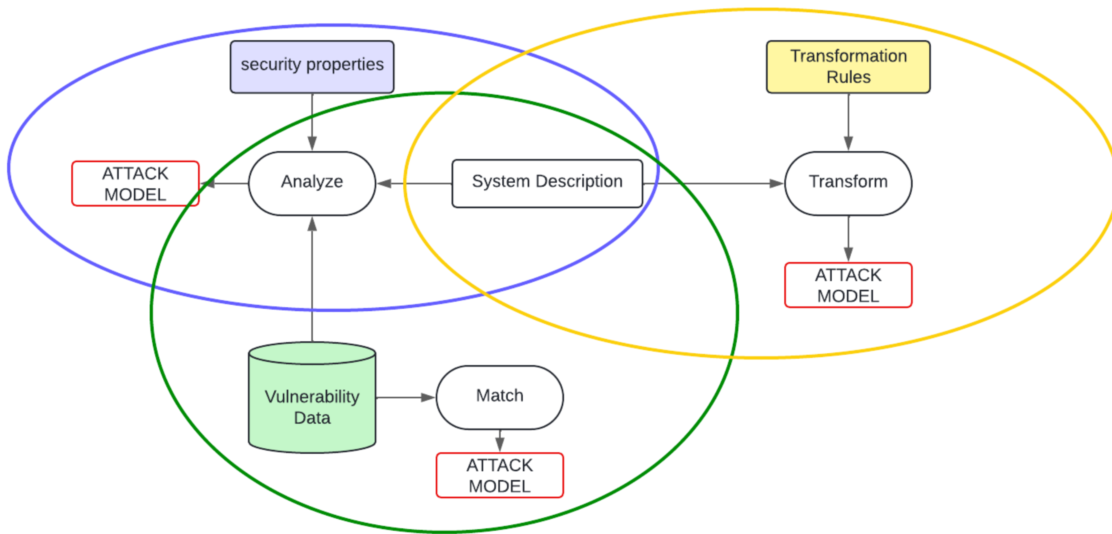

Figure 2.5: Generation Methodologies

mitigate the risks posed by potential attackers, by analyzing network and exploitable attack paths.

2.5 REST API

REST APIs (Representational State Transfer Application Programming Interfaces) represent a standard way to connect components and applications, providing an high level of scalability, flexibility and efficiency for developers.
They uses HTTP request to perform standard operations like creating, reading, updating or deleting resources. An API enables one application or service (the client) to access a resource within another application or service (the server).[8]
REST APIs follow six fundamental principles:

- Uniform interface: APIs requires a standardized interface for communications between client and server, ensuring that the same data belong to the same URI (Uniform Resource Identifier).
- Client-Server Separation: client and server should be independent of each other. The client only need to know the URI of the requested resources because otherwise it can not properly interact with the server.
- Stateless condition: to process a request it must include all the information needed.

Server-side sessions are not required and server is not allow to store information about client request.

- Caching capability: client or server should caches information about resources. This is done to improve performance on client and increase scalability on server.
- Layered system architecture: REST APIs architectures can be composed by many layers so there can be many intermediaries in the communication between client and server. The design of REST APIs need to ensure that client and server can't know if they are communicating with the end application or not.
- On-demand code: optionally, REST supports the extension of client functionality to contain executable code [8].

REST APIs use HTTP methods to communicate and to define possible operations on resources [8]. The most important methods are in Table 2.2.

Method	**Description**	**Example**
GET	Retrieves a record without modifying it	'GET /entities' reports all the entities
POST	Creates a new record on the server	'POST / entities' add a new entity
PUT	Updates an existing record 2	'PUT / entities/1' updates the entity with ID 1
DELETE	Deletes a resource from the server	'DELETE /entities/1' deletes the entity with ID 1

Table 2.2: REST APIs' methods

An **API** is a set of rules and protocols that allows software applications to interact with each other. It acts as an intermediary between the consumer (the application requesting the information) and the provider (the application offering the information), defining how requests and responses should be formatted.

REST is an architectural style that defines a set of constraints for designing applications. It is not a protocol or a standard, it represents a way of designing software based on HTTP protocol. [9]

Chapter 3

Threat Modelling

In this chapter, we present a threat analysis to understand the complexity and impacts of multi-step attacks on Digital Service Chains (DSCs). Our analysis begins with the development of concrete examples of multi-step attacks in Section 3.1. These examples illustrate the sequential nature of such attacks, highlighting the exploitation of multiple vulnerabilities across interconnected systems.
Following the examples, Section 3.2 analyzes the Common Vulnerabilities and Exposures (CVEs) involved in these multi-step attacks. We analyze each CVE to understand how these vulnerabilities can be exploited by attackers at different stages of an attack, providing a clear view of the potential entry points and escalation paths within the DSCs.
Finally, in Section 3.3, we propose a threat model for the structured description of CVEs. This model aims to represent the conditions, strategies, and targets of various threats, enabling a machine-oriented understanding of vulnerabilities and attack sequences.

3.1 Example of multi-step attack

Before developing a threat description, it was essential to model examples of multi-step attacks to understand the relationships between the various steps and the different CVEs involved.
To achieve this, we have developed three detailed examples by thinking about potential attack scenarios in DSCs. These examples are helpfull to illustrate how attackers could exploit vulnerabilities in interconnected services to achieve their malicious objectives.

Example 1

The first example illustrates a multi-step attack consisting of six steps, showing how an attacker can exploit a series of vulnerabilities to gain unauthorized access, escalate privileges, and maintain persistent control over a system:

1. The attacker examine exposed services to identify technologies used and known vulnerabilities. Using Nmap an attacker can perform a scan to identify exposed services and their versions. Suppose the scan finds a vulnerable MySQL server (CVE-2012-2122 MySQL Authentication Bypass).

2. Once some vulnerabilities have been found, the attacker exploits them to gain unauthorized access to the server. For example, if he finds a SQL injection vulnerability, he can insert malicious code to access the database and thus obtain users or critical services credentials. The code depends on the context and on a specific vulnerability, the most common example that can be used is the string "' OR $1 = 1$" and it is used to manipulate the string in order that it is always true. It is possible to use the tool sqlmap to identify the structure of the database to write a more specific injection code (CVE-2019-18609 OpenStack Horizon SQL Injection).

3. Using the stolen credentials, the attacker accesses the CMS (Content Management System) and can load and install plugins containing malware or backdoors to gain remote and persistent access. For example, an attacker can log to the CMS OpenStack Horizon using the stolen credentials and put a plugin containing a backdoor to execute arbitrary commands (CVE-2019-14433 OpenStack Horizon Code Injection Vulnerability).

4. Using a malicious plugin, the attacker executes commands to identify any kernel vulnerabilities or insecure configurations and attempt to obtain higher services on the server. The attacker can use a Python script or a bash shell into the malicious plugin to execute commands on the server and identify system information and potential vulnerabilities (CVE-2021-3156 Sudo Buffer Overflow).

5. Once the attacker has obtained elevated privileges, he installs a RAT (Remote Access Trojan), like Cobal Strike Beacon, on the compromised server and configures it to maintain unauthorized access and control in the long term. Through the RAT the attacker can perform further reconnaissance, collect sensitive information, or perform

malicious actions. First, the attacker loads the RAT payload using the shell and then configures the RAT for persistence connection (CVE-2020-0796 SMBGhost Vulnerability).

6. After the attacker has obtained access to sensitive systems, he executes a targeted ransomware attack against the organization's servers. It can distribute ransomware, like Ryuk, on systems via RAT.

Example 2

The second example involves a similar multi-step approach of the first one, but this time using social engineering techniques and internal exploits to compromise the target system. This attack evolves through the following steps:

1. The attacker sends a targeted phishing email to employees of an organization. This email contains fraudulent links or malicious attachments designed to trick recipients into providing their login credentials.

2. Using the stolen credentials, the attacker logs into a system, such as a server or web application like Tomcat.

3. Once inside the system, the attacker exploits known vulnerabilities to gain administrator privileges. An example is exploiting a vulnerability in sudo services to execute commands as root (CVE-2019-14287 Sudo Security Bypass Vulnerability).

4. With administrator privileges, the attacker installs a backdoor to maintain unauthorized access even after the session is closed. The backdoor communicates with a remote server controlled by the attacker and can execute commands, collect data, or provide remote access (CVE-2018-10933 libssh Authentication Bypass).

5. The malware allows the attacker to move laterally across the internal network, exploiting previously stolen credentials or other security weaknesses. An example is taking control of the Domain Controller via the Netlogon protocol (CVE-2020-1472 Zerologon Vulnerability).

Example 3

The third example provides an attack chain that combines elements of phishing, remote shell access, container escape, privilege escalation, and internal reconnaissance. This scenario highlights the complexity and interconnected nature of attacks in Digital Service Chains (DSC). The attack develops through the following steps:

1. The attacker sends a targeted phishing email to obtain user credentials or exploit a malicious URL.
2. Using the stolen credentials or a malicious URL, the attacker opens a remote shell by exploiting a vulnerability in Apache mod_cgi (CVE-2014-6271 Shellshock).
3. The attacker escapes the Docker container using vulnerabilities such as runC overwrite or Docker daemon privilege escalation (CVE-2019-5736) or Docker cgroups container escape (CVE-2022-0492).
4. The attacker can perform, if necessary, further privilege escalation to obtain root access.
5. The attacker locates critical servers like MySQL or SSH within the network, using tools like nmap for scanning.
6. The attacker uses brute force to access the MySQL server (as described in Example 1) or bypasses authentication on the SSH server (as described in Example 2).
7. Using the compromised MySQL/SSH server, the attacker targets additional resources within the network.

3.2 Analisys of involved CVEs

In this section, we will analyze the Common Vulnerabilities and Exposures (CVEs) involved in the multi-step attack examples. Each CVE represents a specific vulnerability that attackers can exploit at various stages of an attack. The descriptions of the vulnerabilites are taken from CVE Website [10].
We can start with the CVEs in the first example:

- **CVE-2012-2122: MySQL Authentication Bypass**: this vulnerability allows remote attackers to bypass authentication by repeatedly try to authenticate with the same incorrect password. Eventually due to an improper token check, the system accepts the incorrect password as if it is correct and permits the access.

- **CVE-2019-18609: OpenStack Horizon SQL Injection**: this vulnerability causes an integer overflow that leads to heap memory corruption. A malicious server could send a bad frame header that makes the target_size value smaller than it should be. This affects memcpy function which will copy too much data into a heap buffer, causing memory corruption.

- **CVE-2019-14433: OpenStack Horizon Code Injection Vulnerability**: if an API request made by an authenticated user causes an error due to an external exception, the response might leak details about envioremnts and expose sensitve configuration or other data.

- **CVE-2021-3156: Sudo Buffer Overflow**: the vulnerability found in some Sudo versions allows privilege escalation to root.

- **CVE-2020-0796: SMBGhost Vulnerability**: this vulnerability occurs when SBMv3 (Microsoft Server Message Block) handles certain requests improperly, allowing an attacker to execute code remotely on affected system.

Let's now analise the CVEs in the second example:

- **CVE-2014-6271: Shellshock (Bash Remote Code Execution)**: this vulnerability in the Bash shell allows attackers to execute arbitrary using a crafted environment.

- **CVE-2019-5736: Docker runC Overwrite or Docker Daemon Privilege Escalation**: this vulnerability allows attackers to overwrite the host runC binary by exploiting the ability to execute a command as root within a new container with an attacker-controlled image. It is also possible to use and existing container, to which they previously had write access, and attach it with docker exec.

- **CVE-2022-0492: Docker cgroups Container Escape**: this vulnerability in the Linux kernel can, under certain conditions, allow privilege escalation and bypass namespace isolation using a particular feature.

Finally, we can consider the CVEs of the third example:

- **CVE-2019-14287: Sudo Security Bypass Vulnerability**: in Sudo versions before 1.8.28, an attacker with a Runas ALL sudoer account can bypass policy blacklists and session PAM modules, and cause incorrect logging by using a crafted user ID.
- **CVE-2018-10933: libssh Authentication Bypass**: this vulnerability in libssh allows attackers to create a channels without first performing authentication, so it is an unauthorized access.
- **CVE-2020-1472: Netlogon Vulnerability**: this vulnerability allows attackers to gain domain administrator access by exploiting a weak connection to a domain controller.

This analysis highlights the critical role of CVEs in multi-step attacks, illustrating how attackers exploit these vulnerabilities to achieve their purpose.

3.3 Threat description

After developing the examples and analyzing the CVEs involved, we have developed a threat description framework. This framework is designed to enable automated systems to evaluate if a node is vulnerable to a multi-step attack or not. By incorporating detailed conditions, strategies and targets, our threat description provides a structured and systematic approach to identify and mitigate potential security risks.

3.3.1 Structure

The Threat description is divided into 3 parts, Conditions, Strategy and Target. Each component provides a specific purpose in outlining the vulnerabilities and the potential impact of an attack. This detailed representation is helpfull for a machine-oriented understanding of vulnerabilities and attack sequences, facilitating detection, prevention, and response mechanisms.
Conditions describe the environment configurations that make the attack feasible and includes:

"cve": CVE identifier of the vulnerability

"type": Type of vulnerability

"cwe": CWE identifier of the weakness type

"system": Affected systems

- **"software":** Software affected by the vulnerability
- **"component":** Specific part of the software affected by the vulnerability
- **"version":** Range of component versions vulnerable to the threat
- **"os":** Operating system affected by the vulnerability

"privileges": Level of privilege required to exploit the vulnerability

"methodology": Methodology used in the attack (enumeration / exploitation / post-exploitation / pivoting)

"vector": Attack vector (network / physical / social / local / remote)

"payload": Payload used in the attack exploiting the RCE vulnerability

"resource": Resource needed to start the attack (shell/backdoor/network/...)

Strategy describes operations that the attacker must perform to made the attack. Specifically, the fields included in Strategy are:

"steps": steps to perform during the attack

- **"number":** step number
- **"description":** description of the step
- **"tool":** tool or technique used for that step

The last section, Target, describes the final impact of the attack and includes the following fileds:

"impact": ": impact of the attack (confidentiality / integrity / availability / privilege escalation / remote code execution / denial of service)

"description": description of the result of the attack

"privileges": ": level of privileges gained (none/user/root)

"access": type of access obtained (local/remote/physical)

"resources": resources that can be accessed (shell/backdoor/..)

3.3.2 Implementation

After generating the threat description framework, we proceeded by modellomg the CVE information based on this schema. This modeling process allows for a precise and machine-oriented representation of CVEs, which is crucial for automated systems to effectively understand and analyze vulnerabilities.

Below there are an example of a CVE (already descripted in 3.2) modeled using this schema:

```
1  {
2      "condition": {
3              "cve": "CVE-2014-6271",
4              "type": "Remote Code Execution",
5              "cwe": "CWE-78",
6              "system": {
7                  "software": "GNU Bash",
8                  "component": "Bash Shell",
9                  "version": "1.14 - 4.3",
10                 "os": [
11                     "Linux",
12                     "Unix",
13                     "Mac OS X"
14                 ]
15             },
16             "privileges": "none",
17             "methodology": "exploitation",
```

```
18            "vector": "remote",
19            "payload": "arbitrary shell commands"
20        },
21        "strategy": {
22            "steps": [
23                {
24                    "number": 1,
25                    "description": "Craft malicious environment
   variable",
26                    "tool": "script"
27                },
28                {
29                    "number": 2,
30                    "description": "Inject environment variable
   via CGI request",
31                    "tool": "curl"
32                },
33                {
34                    "number": 3,
35                    "description": "Exploit Bash vulnerability
   to execute commands",
36                    "tool": "Bash shell"
37                }
38            ]
39        },
40        "target": {
41            "impact": [
42                "remote code execution",
43                "confidentiality",
44                "integrity"
45            ],
46            "description": "Execution of arbitrary commands,
   potential full system compromise",
47            "privileges": "root",
```

```
48                 "access": "remote",
49                 "resources": "shell"
50             }
51 }
```

Chapter 4

TAMELESS

TAMELESS (Threat & Attack ModEL Smart System) is a threat analysis model that allows deriving the current security state of a system and its components, using input information regarding the system components and their security properties.

As illustrated in figure 4.1, TAMELESS takes as input the description of the system to be analyzed, which includes system components, threats, relationships between components and threats, and user queries.

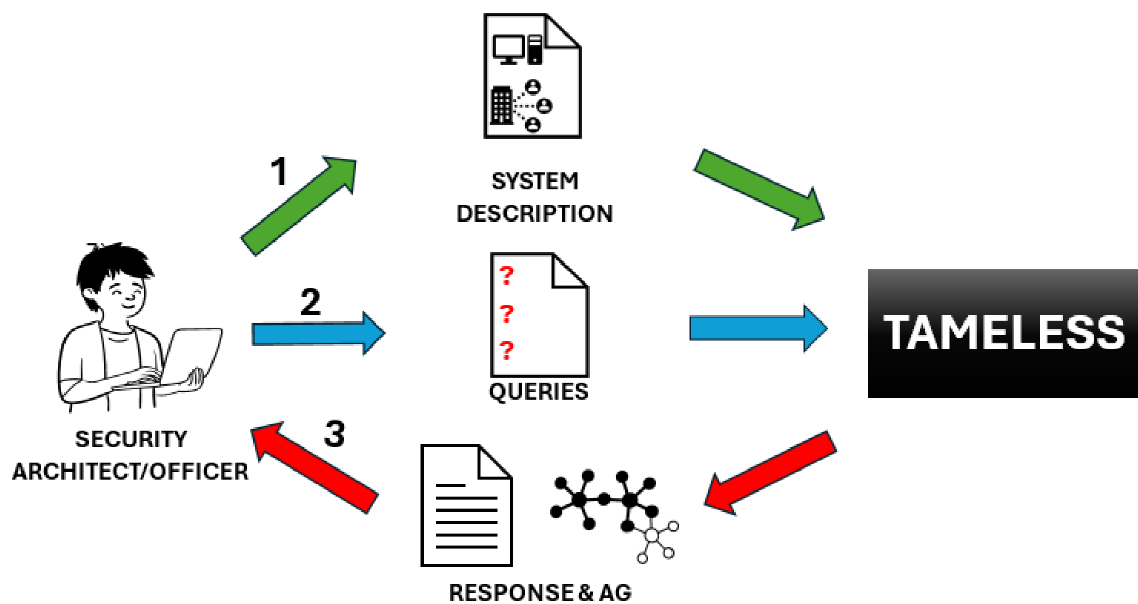

Figure 4.1: TAMELESS workflow

Users conducting a threat analysis can then obtain the output, which includes the default security properties of the system components and a graphical representation of the attack

propagation.[11]
TAMELESS can be used to generate attack graphs by posing specific queries and analyzing the responses. By asking questions about the relationships and vulnerabilities within the system, users can visualize potential attack paths and understand how different threats can propagate through the network.

4.1 Hybrid Threat Model

TAMELESS is based on a hybrid threat model, which combines cyber, physical, and human aspects to represent the relationships and security properties of the system components.
The hybrid threat model consists of two key components: entities (E) and threats (T).
Entities can be categorized as cyber, physical, or human, and threats are actions that can alter the security status of these entities.
The properties of an entity are divided into *Basic Properties* and *Auxiliary Properties.*
The set of all ***Basic Properties*** are

$$P_B = \{\mathrm{Comp}, \mathrm{Malfun}, \mathrm{Vul}\}$$

where

> *Comp(A, T)* means that entity A has been compromised by threat T.
> *Malfun(A)* means that entity is malfunctioning, so some of its functionalities are not working properly.
> *Vul(A, T)* means that A is vulnerable to threat T.

The set of all ***Auxiliary Properties*** are

$$P_A = \{\mathrm{Det}, \mathrm{Rest}, \mathrm{Fix}\}$$

where

> *Det(A, T)* means that it has been detected that entity A has been compromised by threat T.
> *Restored(A)* means that control over A is restored.
> *Fix(A)* means that the functionality of A is repaired.

The model includes several relationships between entities and between entities and threats. ***Relations Between Entities*** are

$$\{\text{Contain}, \text{Control}, \text{Connect}, \text{Depend}, \text{Check}, \text{Replicate}\} \in R$$

where

Contain(A,B) means that A contains B.

Control(A,B) means that A controls B.

Connect(A,B,C) means that B and C are connect trough A.

Depend(A,B) means that the functionality of A depends on that of B.

Check(A,B) means that A checks if B is functioning.

Replicate(A,B) means that A is a replica of B.

Relations Between Entities and Threats are

$$\{\text{Protect}, \text{Monitor}, \text{Spread}, \text{PotentiallyVul}\} \in R$$

where

Protect(A,B,T) means that A protect B from threat T.

Monitor(A,B, T) means that A monitor B from threat T.

Spread(A,T) if A can propagate threat T.

PotentiallyVul(A,T) if A can be vulnebarble to T.

Finally Hybrid Threat Model has some ***High-Level Properties*** that help users to understand the level of system security. They are

$$P_H = \{\text{Val}, \text{Def}, \text{Safe}, \text{Mon}, \text{Che}, \text{Rep}\}$$

where

Val(A) means A is valid.

Def(A,T) means that A is defended from threat T by an entity B.

Safe(A,T) means that A is safe from T.

Mon(A,T) means that A is monitored from threat T.

Che(A) means A is checked.

Rep(A) means that A is replicated. [11]

4.2 Code

The code for TAMELESS is available in the GitHub repository FulvioValenza/TAMELESS. This repository contains the essential components and definitions that drive the TAMELESS framework. The two files of interest for this analysis are system.p and rule.p, which are critical for defining the system components and the rules for threat propagation within TAMELESS.

4.2.1 Rule.p

The rule.p file defines the logical rules and conditions under which threats can propagate and impact the system components. These rules specify how vulnerabilities can be exploited, how attacks can spread, and the countermeasures that can be applied. They govern the interaction between components and threats.In this subsection we will focused on the rules that we have modified during our analysis.

- The function **canbeComp(A, T)** determines if an entity A can be compromised by threat T. It uses relations between entities to establish the compromise. In particular, checks if 'assComp(A, T)' is defined in our system, which indicates that entity A can be compromised by threat T.

```
canbeComp(A,T):-
    assComp(A,T),
    write('\n-> ('),
    write(A),
    write(') is Assumed Compromised by ('),
    write(T),
    write(')').
```

- The function **canbeVul(A, T)** checks if an entity A is vulnerable to a threat T. This function is focused on relations, specifically 'assVul(A, T)', and checks if it is defined in the system, indicating that entity A can be vulnerable to threat T.

```
canbeVul(A,T):-
    assVul(A,T),
```

```
3     write('\n-> ('),
4     write(A),
5     write(') is Assumed Vulnerable to ('),
6     write(T),
7     write(')').
```

- The **potentiallyVul(A, T)** function was originally used to identify entities that are potentially vulnerable.

```
1 potentiallyVul(a7,b7).
```

- The **canbeFix(A)** function determines if an entity A can be repaired.

```
1 canbeFix(A):- canbeMalfun(A), checked(A).
```

During the development of the thesis we have modified these functions and some others related to them in order to achieve different goals, such as including CVEs analysis.

4.2.2 System.p

The system.p file is used to design the structure of the system being analyzed. It contains definitions of all system components, including properties and relationships between them. Each component is detailed with properties such as vulnerability and relationships with other components, represented by spread or connect functions, described in Section 4.1. These definitions are very important for system's security model. Here are some examples of basic component definitions and their relationships:

```
1 e(network).                 %entity
2 t(scanning).                %threat
3 assVul(network, scanning)   %network is vulnerable to scanning
```

According to the rules defined in the file rule.p , 'canbeVul(network, scanning)' will return a positive result, indicating that the network is vulnerable to the scanning threat.

Starting from these files, TAMELESS can be used to generate attack graphs. By querying the system, users can receive responses indicating whether an entity is vulnerable or not. This allows the creation of attack graphs, providing a visual and analytical representation of potential attack paths and improving the understanding of how different threats can propagate through the network. This capability supports dynamic and interactive threat modeling, offering deeper insights into the system's security posture.

4.3 RESTful Web Service in TAMELESS

TAMELESS includes a RESTful web service that provides a standardized interface to interact with components of the system. The APIs are designed to manage entities, threats, relations and properties in a cyber-physical system.
Resources supported by APIs are:

- **Entities**: components of the system, which can be human, cyber or physical.
- **Threats**: potential action that may compromise the security of the system.
- **Relations**: interaction between entities and threats. There can be many type of relation, like relations between two or three entities.
- **Properties**: assumed and derived properties associated with entities and threats.

The RESTful web service in TAMELESS was developed using Jersey framework (version 2.25) on a Tomcat v8.5 server. The development environment was set up in Eclipse Neon 2, using Swagger API.
The API uses standard HTTP methods to define operations to perform operations on resources, like GET, POST, PUT and DELETE, and follows REST principle. The Table 2.2 summarizes these methods.
By using these standard HTTP methods, the TAMELESS API permits consistency and interoperability, in fact, design is flexible and so suitable for many applications. [12]

4.3.1 SystemService.java

In the already mentioned GitHub repository FulvioValenza/TAMELESS, the file 'SystemService.java' implements some RESTfull APIs for managing components in the system.
The file contains methods that handle only the back-end logic for operations, but does not contain the JAX-RS annotations needed to directly expose these methods as RESTful API endpoints accessible via HTTP.
We can divided the operations in 4 sections based on componets they managed.

Entity

The 'SystemService.java' file includes methods to manage entities that can represent human, cyber or physical components. The methods are:

- **createEntity(Entity e):** creates a new entity in the system.
- **getEntities(String name, BigInteger page):** retrieves a list of entities.
- **updateEntity(BigInteger id, Entity toUpdate):** updates an existing entities identified by its ID.
- **deleteEntity(BigInteger id):** deletes an entity by its ID.

Threat

The file also provides methods to manage threats. These methods are:

- **createThreat(Threat t):** creates a new threat in the system.
- **getThreats(String name, BigInteger page):** gets a list of threats.
- **update Threat(BigInteger id, Threat toUpdate):** updates an existing threat identified by its ID.
- **deleteThreat(BigInteger id):** deletes a threat based on it ID.

Relation

The file supports relations between entities and threats, in particular aids creating, retrieving and managing different types of relationships.
Supported methods are:

- **createRelation2Entities(Relation2Entities r2e):** cretaes relationship between two enitites.
- **getRelations2Entities(BigInteger page):** retrieves all relations between two entities.
- **createRelation3Entities(Relation3Entities r3e):** creates relations that involved three entities.
- **getRelations3Entities(BigInteger page):** lists all relationships between three entities.
- **createRelation2Entities1Threat(Relation2Entities1Threat r2e1t):** creates relationships between two entities and one threat.
- **getRelations2Entities1Threat(BigInteger page):** retireves all relations between two entities and one threat.
- **createRelation1Entity1Threat(Relation1Entity1Threat r1e1t):** creates a relationship between one entity and one threat.
- **getRelations1Entity1Threat(BigInteger page):** retrieves all relations between one entity and one threat.

Property

Properties can be both assumed and derived and they are associated with entity and also threat. The file includes the following methods:

- **createProperty1Entity(Property1Entity p1e):** creates a property associated with an entity.
- **getProperties1Entity(BigInteger page):** lists properties associated with one entities.

- **createProperty1Entity1Threat(Property1Entity1Threat p1e1t):** creates a property associated with one entity and one threat.
- **getProperties1Entity1Threat(BigInteger page):** retrieves properties associated with one entity and one threat.

Chapter 5

Thesis overwiev

5.1 Thesis objectives

The aim of this thesis is to enhanche the TAMELESS framework in order to make it more precise in the cyber domain and more efficient in handling Common Vulnerabilities and Exposures (CVEs) and related threats. The enhancements are targeted to improve the framework's capabilities to detect and analyze threats in more complex digital environments. The main goals are:

- **Modification to focus on Cyber Domain:** The first objective is to modify TAMELESSS framework, by modifying, adding or removing functions, to focus only on cyber aspect. This need a previous analysis of the code of the framework to determine necessary changes.
- **Inclusion of CVEs:** The second objective is to include CVEs into TAMELESS's analysis of entities and threats. This can include a future work, based on already developed threat model, which deals with automate the analysis of CVEs to enable framework to automatically identify and manage vulnerabilities.
- **Implementation of Patch Management:** The third objective is to incorporate patch management strategies in the TAMELESS framework. This is done by adding new functions and modifying some others to ensure that ,one a vulnerability is patched, it is no longer considered dangerous for the system.

These objective aim to create a more precise and efficient tool for threat analysis to address

the new challenges in increasingly complex digital scenario, like in Digital Service Chain. This work sets the foundation for future research and development in automating and optimizing threat detection.

5.2 Phases of Work

Several phases of work were carried out to achieve the set objectives. Each phase represent a critical step in enhancement of the TAMELESS framework, starting from the analysis of the framework and ending with testing software on examples.
The phases are the following:

1. **Analysis of CVEs and development of threat model:** this initial phase is focused on analyzing Common Vulnerabilities and Exposures that are relevant in Digital Service Chains (DSCs). The goal was to understand how these vulnerabilities can be exploited in interconnected environments. Then a threat model was created to automate the analysis of CVEs in the TAMELESS framework.

2. **Creation of multi-step attack scenarios:** in this phase, three multi-step attack scenarios were created to simulate realist attacks in interconnected systems. These scenarios were usefull as examples to test the effectiveness of the TAMELESS framework after modification.

3. **Analysis of the TAMELESS framework:** this phase involved analysis of existing TAMELESS framework to identify which functions required modification or needed to be deleted. This analysis highlight the aspects that needed to be modified to our purpose.

4. **Implementation of code modifications:** based on the findings from the previous phases, this phase consists in modifying the code to improve the TAMELESS framework's functionalities. The code was updated to incorporate CVE analysis, patch management and to focus only on cyber aspects.

5. **Validation and testing:** the finale phase focuses on validating and testing the modified TAMELESS framework. The multi-step attack scenarios developed in the previous phase were used to evaluate the framework, in particular its ability to detect, analyze and mitigate threats. Test were made to ensure the functionality of new

components and modified ones. The obtained result were analyzed to confirm that framework work as expected.

Chapter 6

Optimization of TAMELESS

This chapter delves into optimization of the TAMELESS framework after the analysis of original rules discussed in Section 5.2.
The primary goal of these optimizations was to refine the framework to focuse exclusively on cyber threats, as detailed in Section 6.1, in order to have create a more precise and targeted tool.
The second goal deals into integrate Common Vulnerability Exposure (CVE) in TAMELESS's analysis, as described in Section 6.2. We have considered CVEs as a link between entities and threats.
Finally, the chapter discusses the implementation of effective patch management strategies, detailed in Section 6.3, by adding new functions that enabled TAMELESS to consider the application of patches to threats.
All these modifications aim to create a more precise and efficient tool for threat modelling and analysis.

6.1 Modification to Focus on Cyber Domain

After studying all rules in file rule.p, to simplify TAMELESS framework and ensure its relevance exclusively to cyber threats, several functions that were not strictly related to the cyber sphere need to be removed. These modifications are necessary to focus the framework on digital security issues.
In particular we have removed three functions and, as consequence, we have modified some other:

- **canbeMalfun(A)**: This function checks if an entity can be malfunctioning, involving potential mechanical or hardware issues, which is in the physical domain. Removing this function requires updating all operations that involve it, such as 'usable(A)'.
  The removed function is:

```
canbeMalfun(A):-
    assMalfun(A),
    write('\n-> ('),
    write(A),
    write(') is Assumed Malfunction').
```

  The function 'usable(A)' before:

```
usable(A):-
    e(A),(\+canbeComp(A,_T), \+canbeMalfun(A)).
```

  and after removing the function:

```
usable(A):-
    e(A),(\+canbeComp(A,_T)).
```

- **canbeMalfun4Comp(A)**: This function determines if an entity can be malfunctioning because it is compromised. This function has been removed because canindirectly involves non-cyber elements.
  The function sintax is:

```
canbeMalfun4Comp(A):-
    canbeComp(A,_T),
    write('\n-> ('),
    write(A),
    write(') can be Malfunction because it can be
  Compromised').
```

- **canbeMalfun4Dep(A)**: This function checks if an entity can malfunction due to dependency on another malfunctioning entity. Removing this function requires updating all related operations, such as 'canbeVul(A, T)'. The removed function is:

```
1    canbeMalfun4Dep(A):-
2        depend(A,B),
3        (canbeComp(B,_T); assMalfun(B); canbeMalfun4Dep(B)),
4        write('\n-> ('),
5        write(A),
6        write(') can be Malfunction because depend on ('),
7        write(B),
8        write(')').
```

The function 'canbeVul(A, T)' before:

```
1    canbeVul(A,T):-
2        assVul(A,T),
3        canbeMalfun4Dep(A),
4        write('\n-> ('),
5        write(A),
6        write(') is Assumed Vulnerable to ('),
7        write(T),
8        write(')').
```

and after removing the function:

```
1    canbeVul(A,T):-
2        assVul(A,T),
3        write('\n-> ('),
4        write(A),
5        write(') is Assumed Vulnerable to ('),
6        write(T),
7        write(')').
```

By removing these functions, the focus of TAMELESS shifts exclusively towards cyber threats, ensuring that all components and interactions are relevant to the digital security context. This modification not only simplify the model but also improves its applicability and precision in cyber threat analysis.
In conclusion, the removal of non-cyber functions represent an important step in creating a specialized and efficient tool for cyber threat analysis.

6.2 Inclusion of CVEs

The next step of this thesis consists in creating new functions and modifying existing ones in order to incorporate Common Vulnerability and Exposures (CVEs) into the analysis of the vulnerability of an entity A to a threat T. This integration improves TAMELESS framework's ability to analyze vulnerabilities by exploiting standardize informations.
To permit the inclusion of CVEs, we have added two new functions:

- **cve(a1, _)**: This function checks if an entity 'a1' is linked with a cve_id. This connection is important for associating an entity with a specific CVE.

```
1     cve(a1, _)
```

- **'cve_thread(_, b1)'**: This function checks if the cve_id is connected to a threat 'b1'. By identifying this connection, we can link the threat and the entity through the CVE.

```
1     cve_threat(_, b1)
```

Using these two functions together, we can determine if an entity is vulnerable to a threat by connecting them through the same CVE_ID.
Then we have modified the following functions, the original version of which is described in Section 4.2:

- **canbeVul(A,T)**: this updated version of the rule check if an entity A is vulnerable to a threat T, as before the modifications, but additionally verifies if there is a CVE_ID that connects the entity to the threat. The new version of the function is:

```
1    canbeVul(A,T):-
2        (assVul(A, T) ;
3        cve(A, CVE_ID),
4        cve_threat(CVE_ID, T)
5        ),
6        write('\n-> ('),
7        write(A),
8        write(') is Assumed Vulnerable to ('),
9        write(T),
10       write(')').
```

- **potentiallyVul(A,T)**: This updated version of the rule, like the one previously described, determines if an entity A is vulnerable to a threat T and additionally checks if there is a CVE_ID that links the entity to the threat. The new function is:

```
1    potentiallyVul(A,T):-
2        (assVul(A, T) ;
3        cve(A, CVE_ID),
4        cve_threat(CVE_ID, T)
5        ),
6        write('\n-> ('),
7        write(A),
8        write(') is Potentially Vulnerable to ('),
9        write(T),
10       write(')').
```

The modifications and additions made to TAMELESS framework significantly improve its capability to analyze vulnerabilities more accurately by considering CVEs. This represent a significant step in the analysis and optimization of vulnerability assessment in multi-step attacks, contributing to more secure and resilient systems.

6.3 Patch Managment

The final step of the thesis is based on improving the TAMELESS framework by adding patch managment strategies, adding and modifying functions of rule.p file.
This step is important for ensuring that once a threat has been patched, the system recognizes the enity is no longer vulnerable.
To implement patch management, we have introduced two new functions that identify if an entity's threat has been patched. These functions are designed to verify the application of patches, guaranteeing that vulnerabilities are mitigated.
The two functions are:

- **patched(T)**: The function is designed to check if a specific threat, 'T', has been patched. It is helpfull to determine if a vulnerability associated with a threat has been addressed.

```
    patched(T) :-
        applied_patch(T),
    write('\n-> ('),
    write(T),
    write(') can be fixed').
```

- **applied_patch(b1)**: It declares that a patch has been applied for threat 'b1'. This declaration is important for the previous function to determine the status of the threat.

```
    applied_patch(b1).
```

After adding new functions, we have also modified existing ones to integrate the patch management. The updated function is **'canbeFix(A)'**, described in Section 4.2. This modification ensure that the function check if the threat has been patched.
The new version of the function is:

```
canbeFix(A) :-
    canbeVul(A, T),
```

```
3     patched(T),
4     checked(A).
```

By adding and modifying these functions, TAMELESS framework can now manage patches for vulnerabilities, being a more accurate tool. The ability to recognize patches and determine that an entity is no longer vulnerable to a threat, not only improves accuracy of framework but also increase its utility in real world scenario, contributing to more secure systems.

Chapter 7

Validation

In this chapter, we will verify the TAMELESS framework changes that were made in the previous chapter. The goal is to guarantee that the improved set of rules and their implementations mitigate cyber threats within Digital Service Chains (DSCs). In order to achieve this, we will offer several examples that show how the upgraded framework may be used in real-world scenarios.
These use cases will be derived from the attack chain examples presented in Section 3.1, where we detailed multi-step cyber attack scenarios. By applying these realistic usecases, we can better evaluate the effectiveness of the TAMELESS framework in detecting and mitigating vulnerabilities within interconnected digital services.
The validation process aims to ensure that the updated functions accurately detect possible risks and want to illustrate the practical applicability of the framework by simulating realistic cyber attack scenarios and their mitigation. The effectiveness of the modified rules in identifying vulnerabilities, preventing attacks, and maintaining the integrity and security of the DSCs will be analyzed through these use cases.

Validation approach

For each of three changes made to the TAMELESS framework we followed the same validation steps. We started for attack examples developed in Section 3.1. For each example, we modified the file system.p to create a specific environment, including entities, threats and connections.
We made a series of tests to simulate cyber attack scenarios on TAMELESS framework to

evaluate functions we have created or changed. The validation consisted in checking the output ensuring that new outputs correctly represent what expected with the addition or modification of rules.
In addition, to test the robustness of the framework, we implemented some protective system like firewall and intrusion detection system to verify that the framework identified these protection and prevented the spread of threats.

7.1 Validation of Cyber-focused Rule Modifications

In this section, we will verify the effectiveness of the modifications made to the TAMELESS framework, shown in section 6.2, to focus exclusively on cyber threats.
For each example in Section 3.1, we have modified the file system.p to create a specific system for each use case, including entities, threats, connections, and spread rules to test the attack chains on TAMELESS.

Example 1

- Step 1: Examine exposed services

```
1        e(network).
2        t(scanning).
3        assComp(network, scanning).
4        spread(network, scanning).
```

- Step 2: Exploit vulnerabilities to gain unauthorized access to the server (example SQL injection)

```
1        e(sql).
2        t(injection).
3        connect(injection, netowrk, mysql).
4        assComp(mysql, injection).
5        spread(network, injection).
```

- Step 3: Using the stolen credentials, the attacker can access the CMS (Content Management System)

```
1        e(resource).
2        t(compromised_credential).
3        connect(compromised_credential, netowrk, resource).
4        assVul(resource, compromised_credential).
5        spread(resource, compromised_credentials).
```

- Step 4: Use malicious plugin to identify kernel vulnerabilities

```
1        e(kernel).
2        t(malicioud_plugin).
3        connect(malicious_plugin, resource, kernel).
4        assComp(kernel, malicious_plugin).
```

- Step 5: Using a RAT to collect information

```
1        e(server).
2        t(rat).
3        assComp(server, rat).
```

- Step 6: Execute a targeted ransomware against the organization's servers

```
1        e(server).
2        t(ransomware).
3        assVul(server, ransomware).
```

After updating the system.p file with the additions listed above, the terminal output results are:

```
| ?- canbeComp(network, scanning).
-> (network) is Assumed Compromised by (scanning)
yes

| ?- canbeComp(mysql, injection).
-> (mysql) is Assumed Compromised by (injection)
yes

| ?- canbeVul(resource, compromised_credentials).
-> (resource) is Assumed Vulnerable to (compromised_credentials)
yes

| ?- canbeComp(kernel, malicious_plugin).
-> (kernel) is Assumed Compromised by (malicious_plugin)
yes

| ?- canbeComp(server, rat).
-> (server) is Assumed Compromised by (rat)
yes

| ?- canbeVul(server, ransomware).
-> (server) is Assumed Vulnerable to (ransomware)
yes
```

These outputs validate that the modified rules are working correctly, focusing exclusively on cyber threats and effectively detecting and managing vulnerabilities and compromised states in the system.

To further validate the framework, we have added protection mechanisms to verify if they work. These mechanisms were designed to protect the system at various points in the attack chain, ensuring that the framework could accurately detect and handle these protections.

Below are the specific modifications made to the files rule.p and system.p, the resulting terminal output, and the generated attack graph.

The rule in file rule.p that we have modified to check if an entity is protected from a threat is 'canbeComp(A, T)':

```
canbeComp(A,T):-
    (assComp(A,T),\+ protected(A, T)),
    write('\n-> ('),
    write(A),
    write(') is Assumed Compromised by ('),
    write(T),
    write(')').
```

In this function we have added the check if the entity A is not protected from threat T. The same addition can be done also in the rule 'canbeVul(A, T)':

```
canbeVul(A,T):-
    (assVul(A,T),\+ protected(A, T)),
    write('\n-> ('),
    write(A),
    write(') is Assumed Vulnerable to ('),
    write(T),
    write(')').
```

New element in file system.p are:

- Firewall in Step 1:

```
    e(firewall).
    usable(firewall).
    protect(firewall, network, scanning).
```

- Web Application Firewall in Step 2:

```
    e(waf).
    usable(waf).
    protect(waf, mysql, injection).
```

- Security Module in Step 4

```
1     e(security_module).
2     usable(security_module).
3     protect(security_module, kernel, malicious_plugin).
```

- Intrusion Detection System in Step 5

```
1     e(intrusion_detection_system).
2     usable(intrusion_detection_system).
3     protect(intrusion_detection_system, server, rat).
```

The possible terminal output results that we can obtain if all protections are implemented are:

```
| ?- canbeComp(network, scanning).
-> firewall protects network from scanning
no

| ?- protected(network, scanning).
-> firewall protects network from scanning
yes

| ?- canbeComp(mysql, injection).
-> waf protects mysql from injection
-> firewall protects network from scanning
no

| ?- canbeVul(resource, compromised_credentials).
-> (resource) is Assumed Vulnerable to (compromised_credentials)
```

```
yes

| ?- canbeComp(kernel, malicious_plugin).
-> security_module protects kernel from malicious_plugin
-> firewall protects network from scanning
-> intrusion_detection_system protects server from rat
no

| ?- canbeComp(server, rat).
-> intrusion_detection_system protects server from rat
no

| ?- canbeVul(server, ransomware).
-> (server) is Assumed Vulnerable to (ransomware)
yes
```

By running the TAMELESS framework with the updated system.p and rule.p files, we can generate an attack graph that visualizes the attack chain and the impact of the protection mechanisms.
As seen in Figure 7.1, the attack graph will show the initial vulnerabilities and compromise steps but will highlight the protections' role in breaking the attack chain, hence preventing the spread of the attack.

7.2 Validation of CVEs-focused Rules

In this section, we will analyze the impact of the new rules implemented in TAMELESS framework in Section 6.2, which are designed to incorporate CVEs into vulnerability analysis.
Starting from example 2 in Section 3.1 we have defined, in file system.p, entities, threats and other rules for each step of the attack.

Example 2

- Step 1: Email to user

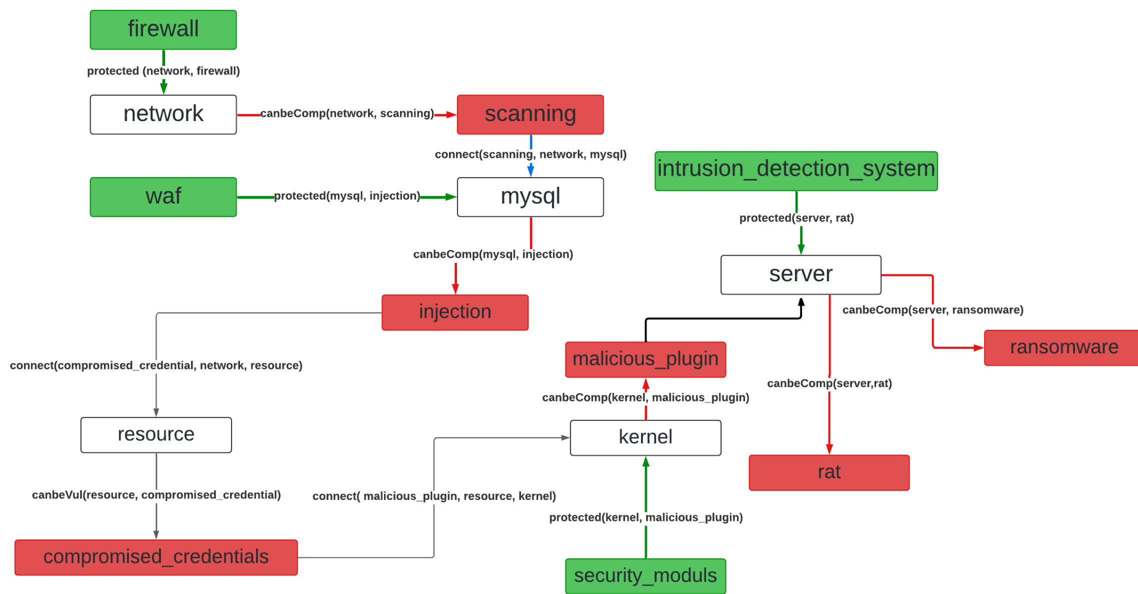

Figure 7.1: Attack Graph Example 1

```
1          e(user).
2          t(phishing).
3          connect(phising, user, system).
4          assVul(user, phising).
5          spread(user, phishing).
```

- Step 2: Using the stolen credentials to logs into the system

```
1          e(system).
2          t(stolen_credential).
3          connect(stolen_credential, user, system).
4          assComp(system, stolen_credential).
```

- Step 3: Privilege escalation by executing know exploit

```
1          e(system).
2          t(privilege_escalation).
3          cve(system, 201914287).
```

```
4           cve_threat(201914287, privilege_escalation).
5           spread(system, privilege_escalation).
```

- Step 4: Maintain unauthorized access using malware

```
1           e(system).
2           t(authentication_bypass).
3           cve(system, 201810933).
4           cve_threat(201810933, authentication_bypass).
5           spread(system, authentication_bypass).
```

- Step 5: Exploit vulnerabilities to move laterally across internal network

```
1           e(network).
2           t(lateral_movement).
3           cve(network, 20201472).
4           cve_threat(20201472,lateral_movement).
5           spread(network, lateral_movement).
```

The terminal output results after changes in rule.p and system.p files are:

```
| ?- canbeVul(user, phishing).
-> (user) is Assumed Vulnerable to (phishing)
yes

| ?- canbeComp(system, stolen_credential).
-> (system) is Assumed Compromised by (stolen_credential)
yes

| ?- canbeVul(system, privilege_escalation).
-> (system) is Assumed Vulnerable to (privilege_escalation)
```

```
yes

| ?- potentiallyVul(system, authentication_bypass).
-> (system) is Potentially Vulnerableto to (authentication_bypass)
yes

| ?- canbeVul(network, lateral_movement).
-> (network) is Assumed Vulnerable to (lateral_movement)
yes
```

Then we have added some protection mechanisms to further validate the model, similar to what we did in Section 7.1. We have changed rules 'canbeVul(A, T)' and 'potentiallyVul(A, T)' to ensure they also check if the entity is not protected from threat.
Here is the modified potentiallyVul(A,T) function:

```
1  potentiallyVul(A,T):-
2      (assVul(A, T) ;
3      cve(A, CVE_ID),
4      cve_threat(CVE_ID, T),
5      \+protected(A, T)
6      ),
7      write('\n-> ('),
8      write(A),
9      write(') is Potentially Vulnerable to ('),
10     write(T),
11     write(')').
```

Then we added the following entities and rules in the file system.p to create protection mechanism against threat:

- Intrusion Detection System in Step 3:

```
1          e(ids).
2          usable(ids).
3          protect(ids, system, privilege_escalation).
```

- Software Anti-Malware in Step 4:

```
1         e(anti_malware).
2         usable(anti_malware).
3         protect(anti_malware, system,
    authentication_bypass).
```

- Firewall in Step 5

```
1         e(firewall).
2         usable(firewall).
3         protect(firewall, network, lateral_movement).
```

The output result that we will obtain if we implement all of these protection are:

```
| ?- canbeVul(user, phishing).
-> (user) is Assumed Vulnerable to (phishing)
yes

| ?- canbeComp(system, stolen_credential).
-> (system) is Assumed Compromised by (stolen_credential)
yes

| ?- canbeVul(system, privilege_escalation).
-> ids protects system from privilege_escalation
no

| ?- potentiallyVul(system, authentication_bypass).
-> anti_malware protects system from authentication_bypass
no
```

```
| ?- canbeVul(network, lateral_movement).
-> firewall protects network from lateral_movement
no
```

We can also create an attack graph that help with the graphical visualization of the attack chain and protection mechanisms, we can see it in Figure 7.2.

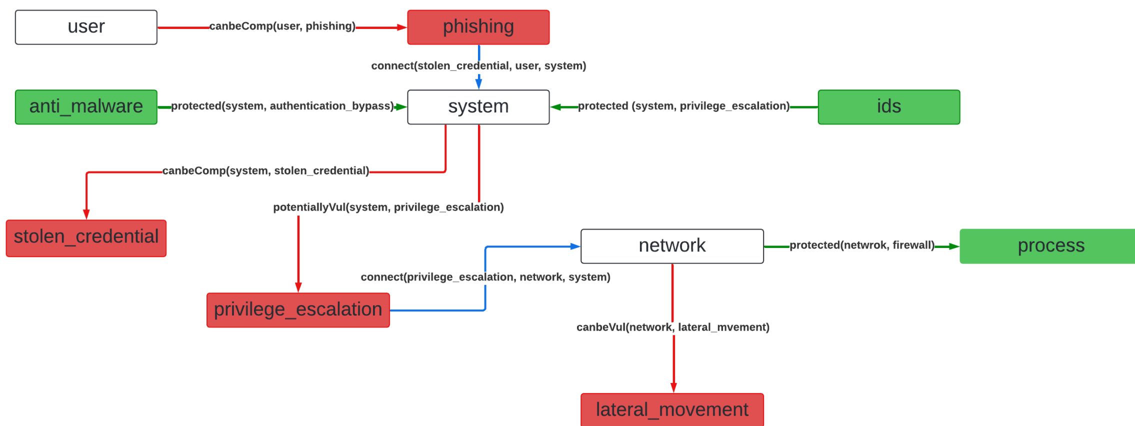

Figure 7.2: Attack Graph Example 2

7.3 Validation of Rules focused on Patch

In this section we will validate the function added and modified, in Section 6.3, for the patching of a threat.

We have started from example 3 in Section 3.1, defining in file system.p all the elements like entities and threats.

Example 3

- Step 1: Email to user

```
1    e(user).
2    t(phishing).
3    assVul(user, phishing).
4    connect(phising, user, apache).
5    spread(user, phising).
```

- Step 2: Log in with stolen credentials or exploit malicious url

```
e(apache).
t(shellshock).
cve(apache, 20146271).
cve_threat(20146271, shellshock).
connect(shellshock, user, docker).
```

- Step 3: Escape the Docker container

```
e(docker).
t(escape).
cve(docker, 20220492).
cve_threat(20220492, escape).
connect(escape,docker,system).
spread(docker, escape).
```

- Step 4: Privilege escalation

```
e(system).
t(privilege_escalation).
cve(system, 201914287).
cve_threat(201914287, privilege_escalation).
spread(system, privilege_escalation).
```

- Step 5: Locate MySQL or SSH

```
e(network).
t(scanning).
assComp(network, scanning).
```

```
4     spread(network, scanning).
```

- Step 6: Access MySQL server or bypass SSH server authentication

```
1     e(mysql_server).
2     t(authentication_bypass).
3     cve(mysql, CVE-2012-2122).
4     cve_threat(CVE-2012-2122, authentication_bypass).
5     spread(mysql, authentication_bypass).
6     connect(mysql_server, authentication_bypass, resource).
```

- Step 7: Using stolen credentials to attack another resource

```
1     e(resource).
2     t(compromised_credentials).
3     assComp(resource, compromised_credentials).
4     spread(resource, compromised_credentials).
```

The terminal output result before adding the patches is:

```
| ?- canbeVul(user, phishing).
-> (user) is Assumed Vulnerable to (phishing)
yes

| ?- canbeVul(apache, shellshock).
-> (apache) is Assumed Vulnerable to (shellshock)
yes

| ?- potentiallyVul(docker, escape).
-> (docker) is Potentially Vulnerableto to (escape)
yes
```

```
| ?- canbeVul(system, privilege_escalation).
-> (system) is Assumed Vulnerable to (privilege_escalation)
yes

| ?- canbeComp(network, scanning).
-> (network) is Assumed Compromised by (scanning)
yes

| ?- potentiallyVul(mysql, authentication_bypass).
-> (mysql) is Potentially Vulnerableto to (authentication_bypass)
yes

| ?- canbeComp(resource, compromised_credentials).
-> (resource) is Assumed Compromised by (compromised_credentials)
yes
```

We have also tried the function 'canbeFix(A)' before adding a patch:

```
| ?- canbeFix(apache).
no

| ?- canbeFix(docker).
no

| ?- canbeFix(mysql_server).
no
```

We have added three patches in steps 2, 3 and 6 to validate the functions created and modified:

```
1 checked(apache).
2 applied_patch(shellshock).
```

```
1 checked(docker).
2 applied_patch(escape).
```

```
1 checked(mysql_server).
2 applied_patch(authentication_bypass).
```

We first have tested the function 'canbeFix(A)' and the output results are:

```
| ?- canbeFix(apache).
yes

| ?- canbeFix(docker).
yes

| ?- canbeFix(mysql_server).
yes
```

Then, we have tested if, adding the patched in steps 2, 3 and 6, the output results of these steps would change. The results are the following:

```
| ?- canbeVul(apache, shellshock).
-> (apache) is Assumed Vulnerable to (shellshock)
no

| ?- potentiallyVul(docker, escape).
-> (docker) is Potentially Vulnerableto to (escape)
no

| ?- potentiallyVul(mysql, authentication_bypass).
-> (mysql) is Potentially Vulnerableto to (authentication_bypass)
no
```

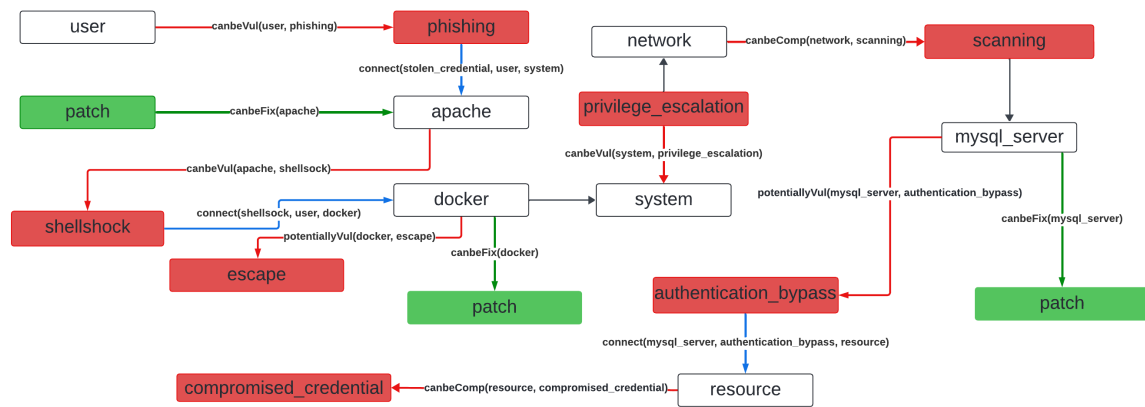

Figure 7.3: Attack Graph Example 3

We have also created an attack graph that is helpful in graphical representation of patches and attack chain, we can see it in Figure 7.3

Chapter 8

Conclusion

This thesis presents significant enhancements to the TAMELESS framework, focusing on improving its capability to detect and mitigate cyber threats by integrating Common Vulnerabilities and Exposures (CVEs).
The modification introduced, including focusing only on cyber aspects and the addition of CVE-based rules, have optimized the framework for managing threats in complex scenarios, like multi-step attacks. We have also introduced patch management to make the framework able to update the security properties of entities. This capability ensures that TAMELESS not only identifies and mitigates threats but also adapts to new vulnerabilities if patches are applied.
A key component of this work is the development of a threat analysis, which allow for better understanding of attack vectors and vulnerabilities in Digital Service Chains (DSCs). By modelling examples of multi-step attacks and analyzing the associate CVEs, we have demonstrated how attackers can exploit vulnerabilities in interconnected digital environments. We have also created a threat model to automate threat analysis, which is essential for improve cybersecurity management.
The examples and results obtained demonstrate the effectiveness of the framework in multi-step attacks, also providing a graphical representation through providing attack graphs. The enhanced framework facilitates more efficient threat detection and response, which helps in managing cybersecurity of a system in interconnected environments like Digital Service Chains (DSCs).

8.1 Future works

An area for possible future work involves the development of TAMELESS framework to improve its evolution in cyber threats. We have started a detailed study of the TAMELESS code, focusing on its API management capabilities. Our idea is to design and implement new APIs that can dynamically integrate threat models into the framework.
The hypothetical new API will accept threat models, already developed, as input and automatically generate corresponding rules in the TAMELESS environment. This automated rule generation aims to improve the framework's ability to analyze and detect threats, allowing it to adapt to new vulnerabilities when they are discovered. The development of these API will enable TAMELESS to operate in a more interconnected environment.
By expanding these capabilities, the FRAMEWORK will be an adaptive solution for cybersecurity, able not only to detect and mitigate current threat but also in be prepared for future ones.

Bibliography

[1] Agron Bajraktari Florian Skopik Markus Wurzenberger, Max Landauer. Automatic attack pattern mining for generating actionable cti applying alert aggregation. *Cybersecurity of Digital Service Chains*, 2021.

[2] David Jaeger. Enabling big data security analytics for advanced network attack detection. *Enabling Big Data security analytics for advanced network attack detection*, 2018.

[3] Matteo Repetto. Chaining digital services: Challenges to investigate cyber-attacks at run-time. *IEEE Communications MagazineComputer Networks*, pages 88–94, 2024.

[4] Sangfor Technologies. What is cve? – understanding common vulnerabilities and exposures, 2022.

[5] MITRE Corporation. Common vulnerabilities and exposures - cve, the standard for information security vulnerability names. https://cve.mitre.org/docs/cve-intro-handout.pdf, 2016.

[6] Kengo Zenitani. Attack graph analysis: An explanatory guide. *Computers and Security 126*, 2023.

[7] Beatrice Spiga Nicola Dragoni Alyzia-Maria Konsta, Alberto Lluch Lafuente. Survey: Automatic generation of attack trees and attack graphs. *computers and security 137*, 2024.

[8] IBM. What is a rest api?

[9] RedHat. What is a rest api?

[10] MITRE Corporation. Cve website.

[11] Rodrigo Vieira Steiner Fulvio Valenza, Erisa Karafili and Emil C. Lupu. A hybrid threat model for smart systems. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 20(5):4403–4417, 2023.

[12] Jaloliddin Yusupov Riccardo Sisto Andrea Ferreri, Fulvio Valenza. Restful interface for tameless, 2019-2020.