Politecnico di Torino

Master of Science in Cybersecurity

Master Degree Thesis

A vulnerability model for software supply chains

Supervisors
prof. Fulvio Valenza
prof. Daniele Bringhenti
prof. Riccardo Sisto
dott. Gianmarco Bachiorrini

Candidate
Giovanni Bloise

Academic Year 2024-2025

Abstract

This thesis addresses the need to detect vulnerabilities in digital supply chains (DSCs).
Modern supply chains rely on interconnected services such as software components, cloud platforms and IoT devices. While this interconnection brings several advantages, it also introduces new security challenges: the overall level of protection is no longer determined only by strong internal defenses, but also depends on the security level of each actor in the chain. Specifically ,each link in the DSC needs a high level of security,which makes manual monitoring of the whole supply chain not scalable.

To address this issue, this work starts from an existing threat analysis framework, *TAMELESS* (Threat & Attack ModEL Smart System). TAMELESS is a tool that, given as input the components of a system, their relationships, and properties, can identify potential threats. This thesis proposes an updated version of the model, specifically optimized for software-based environments. The new version integrates *Common Vulnerabilities and Exposures* (CVEs) into the model's entities, together with new relations,rules and a patching mechanism. The goal is to extend the framework so that it can support threat analysis of digital supply chains and address the challenges arising from their complexity.

The model presented in this thesis makes it possible to analyze a software environment by considering its components, the threats that may target them, the known vulnerabilities (CVEs) they are affected by, and the relationships that connect them. In this way,it is possible to determine to which threats a specific software component is exposed, which vulnerabilities compromise it, and how a potential threat could propagate through the supply chain if that component were compromised.

Finally, the graphical user interface (GUI) of the tool has been enhanced not only to support the new version, but also to improve its usability and make it more accessible for end users.

Acknowledgements

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In an ever-evolving world, technology is one of the fastest changing dimensions. This aspect has changed not only how companies operate internally (and so their own organization as well), but also how they interact with their partners, through (sometimes complex) supply chains.
In a general supply chain, the typical workflow is design the product, then procuring materials and components, making the final product, estimating demand, deciding how you will market the final product, taking orders, thinking about the logistics ,providing customers with visibility into their orders and finally deliver the goods. However, companies started to digitalize each of these steps and so the concept of Digital Supply Chain has become more and more important. A Digital Supply Chain is a set of processes that uses advanced technologies (such as AI, Software, Cloud etc..) to let actors make better decisions about the materials they need, and all the aspects between the demand for their products and the final delivery, making more clear and transparent all the chain.
However, since the Digital Supply Chain uses advanced technologies, it can introduce new risks that can propagate through the supply chain network to (or from) the customer due to the increased connectivity this digitization requires.[6]. It becomes clear then that security is a crucial aspect to consider and this is where threat modeling plays a critical role.
Threat modeling is the set of activities that aim to improve security by identifying threats that may affect your system, application or business and then defying countermeasures to prevent those threats or to mitigate their risk.[7] More precisely, threat modeling is not only about identifying possible threats, but also trying to identify the vulnerabilities that such threats could exploit to compromise your system. In this context, the CVE program is proving to be very useful.
The Common Vulnerabilities and Exposures (CVE) is a database that contains publicly disclosed computer security flaws. The MITRE Corporation,a U.S. Government-funded research and development company, maintains the CVE list, but a security flaw that becomes a CVE entry is often submitted by organizations and members of the open source community. Now, there are different ways to evaluate the severity of a vulnerability; the most common one is the Common Vulnerability Scoring System (CVSS), a set of standards for assigning a number to a vulnerability to assess its severity. Scores range from 0 to 10: higher is the value, higher is the impact of the vulnerability (if exploited). [8]

Today, there are many threat modeling frameworks (such as STRIDE or PASTA, to name a few). While most of them are highly functional, they have some limitations. Specifically, many of these frameworks have limited automation and cannot identify all of a system's dependencies. Indeed, with the increasing complexity of Digital Supply Chains, it is not always easy or possible to fully describe the system being analyzed, especially if that system is highly interconnected, and so it is not always easy to identify many threats that could harm it, especially in multi-step attacks where advanced techniques such as lateral movement are used.
The goal of this thesis work is to analyze the TAMELESS framework,modify and optimize it to be more focused on software environments and includes vulnerabilities from the CVE database. In this way, the framework can be used as a threat modeling and analysis tool for highly interconnected systems such as Digital Supply Chains.

Chapter 2

Threat Modeling

Threat modeling is a fundamental aspect of IT security. When performed effectively, it helps identify weaknesses in the system, allowing us to anticipate or mitigate risks.
This chapter introduces the importance and the core concepts of threat modeling,highlights the main existing frameworks, the benefits and the challenges.

2.1 Fundamentals of Threat Modeling

2.1.1 Definition

Threat modeling is the practice of identifying, communicating and understanding threats and their mitigations, with the goal of protecting something of value (one or more assets). It can be applied to many domains: software and applications, networks, systems and IoT.
Specifically, a Threat Model is a structured way of representing all factors that impact the security of an application.
A well-defined one should always include:

- A clear description of the system under analysis.
- The assumptions being made, that must be revisited regularly since threats evolve as new technologies arise.
- The potential threats that may affect the system.
- The possible mitigation actions for the threats identified.
- A way for validating the model and verifying the effectiveness of the mitigations [7].

Furthermore, it is important to remember that having good security practices does not mean being invulnerable because threats become more sophisticated day by day. However, they play a key role in substantially reducing risks.

2.1.2 Common Methodologies

With numerous threat modeling methodologies available, companies must evaluate and select the most appropriate to meet their needs. It is also important to note that no single methodology fully satisfies all requirements; therefore, a methodology suitable for company A may not necessarily meet the needs of company B.
The following are some of the most commonly adopted ones. [2]

STRIDE

The STRIDE model has been developed by Microsoft and is a widely adopted threat modeling framework designed to help developers identify and classify potential security threats during the design and development phases of a system. In the acronym STRIDE,each letter corresponds to a specific class of security threat:

- **Spoofing identity**: when an attacker impersonates someone else to gain unauthorized access to data or resources.
- **Tampering with data**: the malicious modification of data at rest, in transit, or in use.
- **Repudiation**: when an attacker performs malicious actions that cannot be traced back to them.
- **Information disclosure**: the unauthorized disclosure of sensitive data.
- **Denial of Service (DoS)**: overwhelming a system with excessive requests or traffic to disrupt its normal functionality, often resulting in costly downtime.
- **Elevation of privilege**: exploiting vulnerabilities to bypass security measures and elevate access rights within a system.

Figure 2.1. STRIDE Threat Model [1]

In particular, the STRIDE process involves identifying inherent threats in the system design and implementing security measures to close these gaps. A central

principle of the model is ensuring that software preserves the CIA triad (Confidentiality, Integrity, and Availability).
Finally, the STRIDE model encourages developers to incorporate security considerations into the design and development lifecycle of their software [9].

PASTA

The Process for Attack Simulation and Threat Analysis (PASTA) is a 7-step risk-oriented threat modeling methodology. Unlike approaches that give equal weight to all potential threats, PASTA focuses on those with the highest risk, allowing organizations to allocate resources and time more effectively by prioritizing vulnerabilities with the highest impact. Moreover, PASTA gives a strong focus on the business context, which is often less addressed in other methodologies such as STRIDE.
PASTA is articulated into seven stages:

1. **Identification of assets and definition of the application architecture:** define what must be protected and the structure of the application.
2. **Definition of the application's threat environment:** analyze factors that could harm the application.
3. **Functional decomposition of the application:** decompose the application with the aim of identifying potential weaknesses that attackers may exploit.
4. **Identification of important attack scenarios.**
5. **Analysis of the identified attack scenarios:** applying the STRIDE threat framework.
6. **Identification of potential threat agents:** identify actors that could exploit the vulnerabilities.
7. **Prioritization and mitigation of the identified threats:** rank threats based on their impact on the application and implement countermeasures [2].

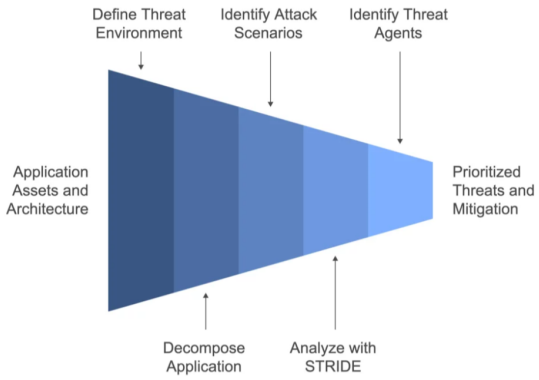

Figure 2.2. PASTA Threat Model [2]

TAMELESS

The Threat & Attack ModEL Smart System (TAMELESS) is a threat modeling framework designed to perform threat analysis on hybrid systems (where *hybrid* means that it takes into account human,physical and cyber aspects).
Specifically, TAMELESS is a prolog-based tool that:

- Represents the relations and security properties of the system's components by considering their cyber,physical and human aspects.
- Introduces a threat analysis method based on a set of derivation rules that allow to understand the overall security state.
- Adopts a static analysis approach, meaning that when the system changes, the analysis must be run again.
- Relies on a deterministic evaluation, where relations and derived facts are either true or false.

TAMELESS takes as input the system specification (i.e: components and threats), the relations among components and between components and threats, and the security assumptions. The user can then query TAMELESS to obtain a set of predefined security properties of the system components. In conclusion, TAMELESS enables the analysis of hybrid threats in smart systems, such as smart buildings, smart homes or financial transactions [10].

2.1.3 Benefits and Challenges

Benefits

There are many benefits of Threat Modeling, including:

- **Enhanced security awareness:** by participating in threat modeling activities, organizations can educate teams on the potential threats they may encounter and on the measures available to mitigate these risks.
- **Cost-effective security:** identifying vulnerabilities early in the system lifecycle significantly reduces the costs associated with security incidents and data breaches.
- **Risk mitigation:** by identifying potential threats before they are exploited, organizations can adopt proactive measures to eliminate or reduce risks, rather than relying on reactive responses after an incident has occurred.
- **Evaluation of emerging attack vectors:** new threats and attack vectors emerge every day. By continuously updating their threat models, organizations can address even the most recent risks.

- **Identify security requirements:** threat modeling helps define the security requirements necessary to protect assets, supporting a secure by design approach.
- **Mapping of assets, threat agents, and controls:** threat modeling helps organizations understand what they need to protect, from whom and how they can defend it. [11]

Challenges

The most commonly used Threat Modeling techniques (such as STRIDE or PASTA) are mostly manual, so security personnel continue to face some of the following challenges: [12][13][14]

- **Scalability:** according to the 2021 BSIMM survey, organizations lack the resources needed to assign senior personnel to multi-day threat modeling exercises for each project or application.
- **Completeness and control implementation:** manual threat modeling usually focuses on a subset of threats, so security experts are required to assist developers after the threat analysis to implement mitigations.
- **Consistency:** people have different ideas of what a threat is and how to rank it based on its impact. This affects which threats development teams choose to prioritize. Moreover, the lack of a standard leads to inconsistent results.
- **Complexity:** with applications moving to the cloud, the evolution of the IT systems and the need to introduce new features, today's systems have become highly dynamic. Under these conditions, manual threat models are no longer effective. The literature presents several automated approaches that apply mitigation strategies directly; however, they act only on known threats and do not perform threat analysis. [15] [16] [17] [18]

2.2 Threat Modeling Process and Results

2.2.1 Process

Currently, there is no single standardized process for threat modeling, as it often changes depending on the needs of each organization.
However, there are four key steps that should always be included in the process:

1. **Scope your work.**
2. **Determine threats.**
3. **Determine countermeasures and mitigations.**
4. **Assess your work.** [5]

Scope your work

The first step of the threat modeling process is to understand the system under analysis. This may include activities such as:

- Creating diagrams such as data flow diagrams (DFDs).
- Identifying entry points where potential attackers could interact with the application.
- Identifying relevant assets.
- Defining trust levels, which represent the access rights granted to external entities.
- Reading or creating a user story (sometimes extended to abuser stories, misuse cases, etc.).

Data flow diagrams are particularly useful because they illustrate the different paths within the system and highlight privilege or trust boundaries.
This phase is often called *decomposing the application*. It is a method frequently used by consultants when performing threat models or architectural reviews, producing a Threat Model report.
The final step of this first phase often involves identifying assets, which may include valuable elements to be protected, intermediate resources and targets of potential attackers. Sometimes asset identification may fall outside the scope and become a distraction, while in other cases assets may remain difficult to define until attackers themselves reveal them. [5]

Determine Threats

An important part of threat identification is the use of a threat categorization methodology. In particular, as explained in the previous section ,STRIDE is commonly applied in threat modeling, while kill chains and MITRE ATT&CK are often used for operational threat modeling.
The purpose of threat categorization is to facilitate the identification of threats. The DFDs created in the first step are useful for highlighting possible targets such as data sources, processes, data flows, and user interactions.
Once identified, these threats can be further organized through the use of threat trees, with one tree associated with each threat goal. Moreover, common threat lists with examples can make the identification process easier.
Finally, use and abuse cases can be used to show how existing protective measures could be bypassed and/or where such measures are missing. [5]

Determine Countermeasures and Mitigation

A vulnerability can be mitigated through the implementation of a countermeasure. These countermeasures are often identified using threat countermeasure mapping

lists; however, their prioritization is still a complex and open topic. Different approaches exist, and each organization must choose the method that best meets its needs. Common factors considered include the likelihood of an attack, the potential damage it could cause, and the cost or complexity of the fix.
The risk mitigation strategy usually involves evaluating threats based on their potential business impact. Once the impact is identified, different options for addressing the risk are available:

- **Accept:** decide that the impact is acceptable and document who has chosen to accept the risk.
- **Eliminate:** decide to eliminate the risk.
- **Mitigate:** introduce features or controls to reduce either the impact or the probability that the risk occurs.
- **Transfer:** transfer the risk to another party such as a costumer. [5]

Assess your work

The last phase focuses on verifying the effectiveness of the work performed. Specifically, it involves validating the model to assess whether it correctly identifies the threats to the assets and whether it is able to suggest appropriate countermeasures. In summary, this final phase is useful for determining whether the model functions properly and achieve the defined goals.

2.2.2 Example of a Threat Model Process

The following example illustrates a simple threat model for an application. [5]

Scope the work

The goal of this phase is to understand the application under analysis, gathering all the information considered useful for the entire process. Some important information may include:

- Application name.
- Application version.
- Description.
- Threat modeling document owner.
- Participants involved in the threat modeling process.
- Reviewers of the threat model.

For example:

Application name: Library App
Version: 1.0
Description: library website
Document Owner: John X
Participants: Mary Y
Reviewer: Mike Z

During this phase, it is also important to identify and document the external dependencies. In particular,the production environments and requirements should be analyzed and reported with their ID and Description. Example:

ID	Description
1	The website will run on a Linux server
2	The database server will be MySQL
3	The web server and the database will communicate through a private network
4	The web server is behind a firewall

Table 2.1. Threat model example - External Dependencies [5]

The next important aspect is the identification of entry points and assets. Entry points determine how users (and therefore potential attackers) can interact with the application i.e where data enters the system. Assets represent the information or resources that need protection.
Both entry points and assets must be carefully documented as they are a fundamental part of the threat modeling. In particular,they can be reported with an ID, a name and a trust level.
Trust levels specify the permissions that the application assigns to external entities. Example [5]:

ID	Name
1	Anonymous Web User
2	User with Valid Login Credentials
3	User with Invalid Login Credentials
4	Librarian
5	Database Server Administrator
6	Website Administrator
7	web server User Process
8	Database Read User
9	Database Read/Write User

Table 2.2. Threat model example - Trust Levels

ID	Name	Description	Trust Level
1	HTTPS port	all the pages are layered on this port	(1),(2),(3),(4)
2	Login Page	Used to log in to the website	(1),(2),(3),(4)
3	Search Entry Page	Used to enter a search query	(2),(4)

Table 2.3. Threat model example - Entry Points

ID	Name	Description	Trust Level
1	Library users and Librarian	Assets relating to students and librarians	(2),(4),(5),(6),(7),(8),(9)
2	System	Assets relating to the underlying system	(5),(6),(7),(8),(9)
3	Website	Assets relating to the library website	(2),(4),(5),(6)

Table 2.4. Threat model example - Assets

Finally, the gathered information makes it possible to model the application through the use of the Data Flow Diagram (DFD) [5].

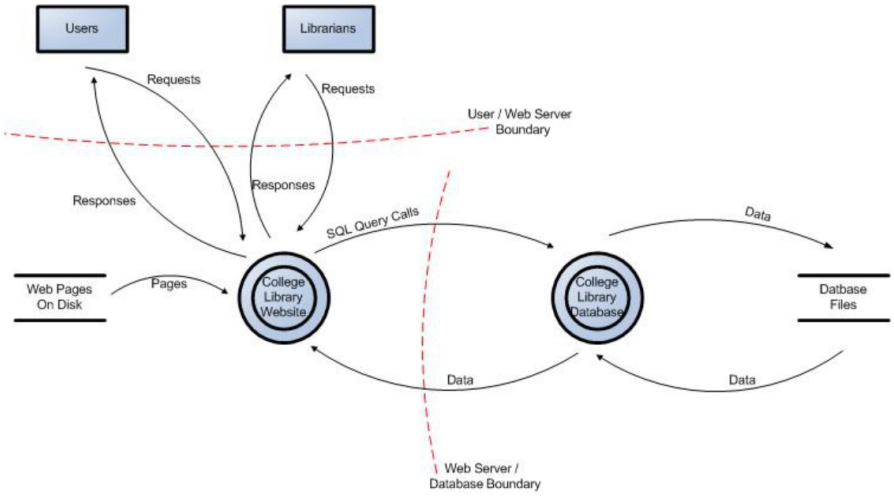

Figure 2.3. Threat model example - DFD

Determine Threats

Threat lists, such as those based on the STRIDE model, are useful for the identification of potential threats. Afterwards , a threat analysis can be performed.
Threat analysis focus on identifying possible threats to an application by examining the system. It consists of several steps: first, all threats relevant to each component are considered; second, threats are analyzed by exploring the attack paths, root causes and mitigation controls; finally, the identified threats can be ranked based on their impact. Different risk factors can be used to classify threats

as High, Medium or Low [5].
In our example, some of the main threats that have been identified[1] , with their respective ratings, are:

Threat	Rating
SQL injection	High
Spoofing	High
Privilege Escalation	High
XSS	High
DoS	Medium
Information Disclosure	Medium
Outdated libraries	Medium

Table 2.5. Threat model example - Threats

Determine Countermeasures and Mitigation

The goal of this phase is to determine whether protective measures exist for the identified threats.
In our example, some possible countermeasures that can be adopted are:

Threat	Countermeasures
SQL injection	1. Input validation 2. Parameterized queries
Spoofing	1. Appropriate authentication 2. Protect secret data 3. Don't store secrets
Privilege Escalation	1. Run with least privilege
XSS	1. Sanitize user input 2. CSP
DoS	1. Appropriate authentication 2. Appropriate authorization 3. Filtering 4. Quality of service
Information Disclosure	1. Authorization 2. Encryption 3. Don't store secrets
Outdated libraries	1. Remove unused packages 2. Regular patch cycle

Table 2.6. Threat model example - Countermeasures [5]

[1]The list of threats mentioned is not exhaustive and has been simplified for the purposes of this example.

Finally, once the threats and their countermeasures have been identified, they can be classified as follows:

- **Mitigated:** countermeasures have been applied.
- **Partially mitigated:** threats that are only partially mitigated by one or more countermeasures.
- **Not mitigated:** threats that do not have countermeasures in place. [5]

Conclusion

Threat modeling complements code review by identifying entry points and potential threats early in the Software Development Lifecycle (SDLC), allowing reviewers to focus on the most important areas. Moreover, for existing applications, it promotes a depth-first approach that prioritizes the review of components associated with higher-risk threats.

Chapter 3

KillChain and CVE

The previous chapter introduced the concept of Threat Modeling, highlighting the importance of identifying potential threats and countermeasures to protect assets. However, defending a system also requires an understanding of how an attack can be carried out.

This chapter introduces the kill chain and the Common Vulnerabilities and Exposures (CVE), two important concepts that help to understand the typical workflow of an attack. In particular, the kill chain highlights the steps that an attacker often follows when targeting a system, while CVE provides a standardized way to describe and classify vulnerabilities.

3.1 Kill Chain

3.1.1 Definition

In military context, the term *kill chain* refers to the process of identifying and destroying a specific target. In 2011, Lockheed Martin introduced the same concept in the field of cybersecurity. Specifically, the company developed a cyber kill chain to describe the steps of a cyber attack.
These steps are:

1. **Reconnaissance.**
2. **Weaponization.**
3. **Delivery.**
4. **Exploitation.**
5. **Installation.**
6. **Command and Control (C2).**
7. **Actions and Objectives.**

It is important to note that detecting a threat in the earlier stages of the chain reduces potential damage compared to detection in the later stages [19].

3.1.2 Attack Lifecycle

Reconnaissance

Reconnaissance is the first of the seven stages of the kill chain. During this initial phase, the attacker tries to gather as much information as possible about the chosen target.To achieve this, he may use different tools as well as OSINT, which allows him to collect valuable information from publicly available sources. This step is very important, since based on the information he is able to collect, the next steps may be more or less effective.
In particular, depending on his final goal, he needs to create different types of links. If he aims at destroying a system, a one way link may be established. However, if the goal is to steal sensitive data, a two way link is required [19].
A practical example of this phase is the use of *nmap* to scan the target, which provides information about the services and ports that are active and potentially exploitable. Once such ports are identified, the attacker can try to establish communication, such as through an authorized Telnet connection, which then serves as a possible entry point for later stages of the kill chain.

Weaponization

Once the adversary collects all the information he needs and identifies possible vulnerabilities, he uses them to prepare what to deliver and how. Specifically, there are two types of payloads:

- Malware that does not require communication with the adversary, such as viruses and worms.
- Malware that requires communication with the adversary, such as Remote Access Trojan (RAT) [19].

For example, if during the Reconnaissance phase the attacker discovered that the target company relies on PDF documents and emails, he can prepare a PDF that contains a malware. It may appear to be a legitimate Threat Modeling report, but once the victim opens it, the exploit is triggered.
It is important to note that in this phase the file is not yet delivered, but only prepared for the next steps of the kill chain.

Delivery

In this phase, the malware that was prepared in the previous step is delivered and it can be done in different ways. For example, the attacker can send a phishing email with the malicious PDF attached and ask the victim to download or open it (user interaction is required). Another possibility is to deliver the payload by exploiting software or hardware vulnerabilities (user interaction is not required). This second method is generally more sophisticated and therefore more difficult to detect and block [19].

Exploitation

Exploitation is the fourth stage of the kill chain. Once the malicious payload is delivered, several conditions must be satisfied in order for the payload to be installed on the target device:

- The malware must have the minimum privileges required for the installation.
- The target's OS must be compatible with the malware (i.e: malware created for Windows cannot be installed on a macOS).
- The anti-malware defenses must not detect (and therefore block) the installation.

If any of these conditions are not met, the kill chain fails. Moreover, during this phase the malware is not yet installed but rather prepared for the installation phase [19].

Installation

At this stage, the infection of the target device begins. For malware to be installed or executed, exploitable flaws must exist within the system. Many of these flaws are listed in the Common Vulnerabilities and Exposure (CVE) database (discussed later as a fundamental part of this work). After the installation, malware changes the appearance of its files to avoid detection.
Moreover, malware can be classified as follows:

- **Polymorphic malware:** the payload remains unchanged while the header is modified, so the file appears normal in memory.
- **Metamorphic malware:** it changes the pattern of the payload in memory.

This phase also ensures that the attacker can establish communication with the target device, allowing the next stage to begin [19].

Command and Control

The adversary establishes a Command and Control channel to monitor and guide the installed malware to fulfill its goal.One of the most challenging aspects of the final two stages of the kill chain is to remain undetected, while from a defensive perspective this phase is the last opportunity to block the malicious activity.
The channel created can then be used to steal data and/or to send instructions to the malware, enabling lateral movement to discover other valuable assets.

Actions on Objectives

In the final phase of the kill chain, the malware starts to carry out its functions. Some of the most common actions include:

- **Data exfiltration:** stealing data.
- **Ransomware:** encrypting the victim's data.
- **Cyber terrorism:** erasing data.

If the adversary achieves its goal, the damage to the target can be catastrophic; therefore, it is important to detect the attack as early as possible.
For this reason threat modeling plays an important role as it helps defenders identify what data attackers might look, the possible attack vectors and the vulnerabilities that may be exploited [19].

3.2 Digital Supply Chain

3.2.1 Introduction

A digital supply chain can be defined as a connected system of processes that uses advanced technologies and shared data to improve decision-making capabilities of all stakeholders across the chain. In traditional supply chains, information moves step by step, and decisions are often made only after problems occur. By contrast, digital supply chains use real-time data to give all participants better visibility and control. Sensors, monitors, and IoT gateways collect information during production, transport, and distribution. This data is then aggregated and shared among stakeholders, creating a collaborative environment in which each actor can anticipate changes in demand, identify potential disruptions, and adjust operations accordingly.

Benefits

Traditional supply chains rely mainly on people to interpret machine data and act accordingly, which can take time and lead to mistakes. In digital supply chains, much of the analysis is done automatically by systems that process the data and suggest or even trigger actions. Humans remain in control, but the technology helps them to act faster and more accurately. This reduces the need for long and detailed planning and increases the ability to deal with unexpected changes.
Another key element is the level of collaboration; in digital supply chains, suppliers, service providers and customers are connected through a shared flow of information. For example, quality data from a supplier can immediately be used by a manufacturer to adjust its production. This improves efficiency and reduces waste by ensuring that everyone has the same up to date information.
Finally, digital supply chains represent a strategic evolution in supply chain management. They use technology and data to create a more flexible, transparent, and

resilient system. Instead of a linear chain where each step depends only on the one before it, the digital supply chain is an interconnected network. This model is better suited to today's complex and global industrial environments, where speed, adaptability, and collaboration are essential.[6].

3.2.2 Risks in Digital Supply Chains

In today's interconnected world, organizations depend on a wide network of external partners to deliver products and services. Although globalization and digitalization offer significant advantages, they also reduce the control that organizations have over their supply ecosystems. As consequence, protecting only internal infrastructures is no longer enough [20].
A typical digital supply chain attack usually develops in two stages: first,the attacker compromises the supplier, and then it uses this access to target the supplier's customers [21].
In recent years, cases such as SolarWinds and the MOVEit breaches have shown that supply chain attacks can impact thousands of organizations at once. These examples demonstrate that such risks are not limited to single companies, but represent systemic threats that can disrupt entire ecosystems.

Supplier attacks

Table 3.1 outlines common techniques used to infiltrate suppliers. These range from technical exploits such as malware infections or brute force attacks, to human approaches such as phishing and social engineering.

Technique	**Example**
Social Engineering	phishing, fake applications
Brute-Force Attack	guessing a web login
Exploiting Software Vulnerability	SQL injection or buffer overflow exploit in an application
Exploiting Configuration Vulnerability	taking advantage of a configuration problem
OSINT	search online for credentials, usernames
Counterfeiting	imitation of USB with malicious purposes

Table 3.1. Attack techniques used to compromise the supply chain [3].

Suppliers manage a wide range of critical assets, from software components to sensitive customer data. Compromise these assets can provide attackers with an effective entry point into downstream organizations. Table 3.2 summarizes the most common targeted assets.

Asset	Example
Pre-existing Software	web servers, applications, databases, firmware.
Software Libraries	third party libraries
Code	software produced by the supplier
Configurations	passwords, API keys, firewall rules
Data	information about the supplier, certificates, personal data of customers
Processes	updates, backups, certificate signing
Hardware	hardware produced by the supplier, chips
People	targeted individuals with access to data, infrastructure, or other people

Table 3.2. Supplier assets targeted in supply chain attacks [3].

Customer attacks

Once a supplier is compromised, attackers often shift their focus to the supplier's customers. Customers may be deceived through social engineering or directly compromised via technical exploits. Table 3.3 presents typical techniques.
Certain attacks, such as malware infection or hardware tampering, are relevant for both suppliers and customers. However, their impact differs: in suppliers they provide a foothold for downstream attacks, while in customers they are often the final step of the intrusion chain.

Technique	Example
Trusted Relationship	trust a certificate, trust an automatic update
Drive-by Compromise	malicious scripts in a website to infect users with malware
Phishing	messages impersonating the supplier
Counterfeiting	create a fake USB, modify a motherboard

Table 3.3. Attack techniques used to compromise customers [3].

Customers also manage assets, both technical and human. By compromising financial resources, software, or employees, attackers can inflict direct damage and use the compromised environment to further propagate malicious activities. Table 3.4 lists the most commonly targeted assets.

Asset	Example
Data	payment data, documents, emails, financial data
Personal Data	customer data, employee records
Software	access to the customer product source code
Bandwidth	use bandwidth for DDoS, mass infection
Financial	hijack bank accounts, money transfers
People	individuals targeted due their position

Table 3.4. Customer assets targeted in supply chain attacks [3].

Significant Incidents

Follow some of the most famous supply chain attacks:

- In February 2021, ethical hacker Alex Birsan demonstrated the potential scale of supply chain vulnerabilities by launching a "novel supply chain attack." His technique successfully infiltrated the network systems of more than 35 major technology companies, among them Microsoft, Apple, PayPal, Shopify, Netflix, Tesla, and Uber [22].

- SolarWinds, a major software provider, was the target of one of the most significant supply chain attacks known. The intrusion began in September 2019, when attackers first gained access to the company's internal network. By October, they were experimenting with injecting malicious code into Orion, SolarWinds' flagship software platform. After several months of testing, the attackers succeeded in embedding a backdoor later named *Sunburst*. On March 26 2020, SolarWinds unknowingly distributed Orion updates that contained this malicious code. More than 18,000 customers installed the updates, thereby spreading the malware. Once inside a victim's system, the malware granted attackers remote access, enabling them to deploy additional tools and conduct extensive cyber espionage campaigns across multiple organizations [23].

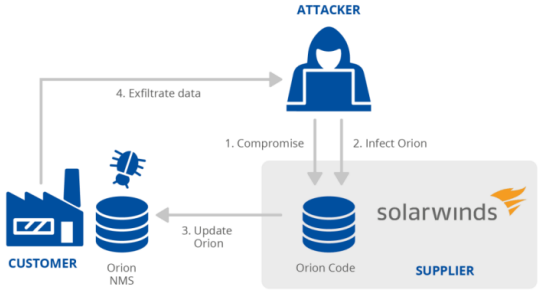

Figure 3.1. SolarWinds attack [3]

- The MOVEit attack was a cyber attack that exploited a vulnerability in the MOVEit managed file transfer service, a tool widely used by organizations to securely exchange sensitive data. The attack began on May 27 2023, and relied on a zero day vulnerability that allowed attackers to inject SQL commands and gain unauthorized access to customer databases. The attack has been attributed to the Cl0p ransomware group, which is known for extorting victims and publishing stolen data on dedicated leak sites. Among the affected organizations were the BBC, British Airways, Boots, and Aer Lingus, whose employees' personal information was potentially exposed [24].

3.2.3 Empirical Evidence and Prevention Strategies

According to Cisco's Annual Cyber Security Report in 2018, 38% of the surveyed organizations foresee a risk of cyber security attacks in their manufacturing supply chains as they combine operational technologies with information technologies [22]. Moreover, according to the very recent Ivanti's 2025 State of Cybersecurity Report [4], just 1 out of 3 companies are ready to protect themselves 3.2 and half of them still don't have identified the most vulnerable components in their software supply chain 3.3.

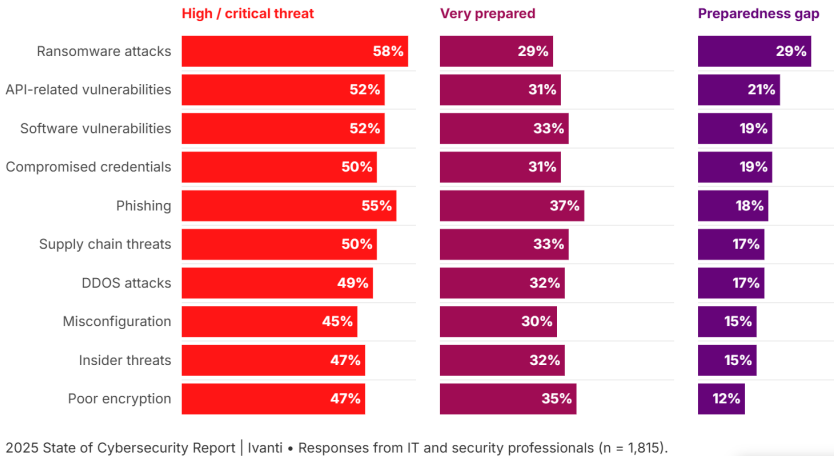

Figure 3.2. preparedness gap against cyber threats [4]

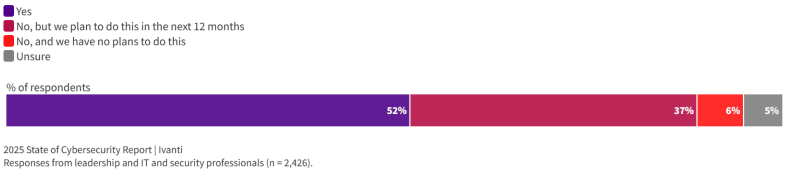

Figure 3.3. identification of the most vulnerable components in the SW supply chain [4]

However, organizations are moving to prepare themselves and to identify the areas where they may be most vulnerable. They are beginning to analyze which connections, processes, or third-party dependencies represent the most critical entry

points for potential attackers.
In this context, threat modeling helps. It provides a structured way to visualize the entire digital supply chain, identify weak spots, and understand how those weaknesses could be exploited. By modeling possible attack paths, organizations can anticipate which assets might be targeted, what kind of damage could occur if an entry point is compromised, and how attacks could propagate across interconnected systems.
This approach not only highlights where defenses need to be placed but also helps prioritize resources, ensuring that the most valuable assets and the most fragile links in the chain receive proper attention.

3.3 Common Vulnerabilities and Exposures (CVE)

3.3.1 Introduction

The Common Vulnerabilities and Exposures (CVE) system was introduced in 1999 by the MITRE Corporation. CVE provides a standardized method for reporting and tracking security flaws, and CVE identifiers (CVE IDs) make it possible to uniquely recognize vulnerabilities across systems, improving the coordination of security tools and mitigation strategies.
Although MITRE manages the CVE list, entries are often submitted by members and organizations of the open source community. In this way, CVE helps IT professionals to prioritize and address security flaws.
A flaw is assigned a CVE Id if it meets the following conditions:

- **Independence:** the vulnerability can be fixed without requiring other vulnerabilities to be resolved first.
- **Acknowledgment:** either the vendor confirms the bug and its security impact or the reporter provides sufficient evidence of its effect on system security.
- **Scope:** the flaw affects only a single codebase. If multiple products are impacted, each one receives its own CVE Id unless they share unavoidable vulnerable code. [8]

Moreover, vulnerabilities can be categorized as follows:

- **Software vulnerabilities:** flaws in operating systems and software.
- **Network vulnerabilities:** software vulnerabilities but exist in network devices or protocols.
- **Embedded/Firmware/Hardware vulnerabilities:** flaws in devices that attackers primarily exploit to steal intellectual property or to counterfeit the device [19].

3.3.2 CVE JSON Record Format

A CVE record is structured in JSON format and contains several mandatory fields. Specifically, each record must include:

- **dataType:** specifies the type of data in the JSON record ("CVE_RECORD").
- **dataVersion:** specifies the version of the CVE schema used to validate the record.
- **cveMetadata:** contains information about the CVE itself, including the CVE ID, the organization that requested or assigned it, the request date and the current state (i.e: PUBLISHED).
- **containers:** stores the vulnerability details. Each record must include one *cna* container that contains the vulnerability information provided by the CNA that created the CVE entry. However,several *adp* containers may be included allowing organizations in the CVE program to add additional details related to the vulnerability [25].

To better clarify the structure, the following example presents a basic CVE record showing how these fields are organized [26].

Example

```
{
  "dataType": "CVE_RECORD",
  "dataVersion": "5.1",
  "cveMetadata": {
    "cveId": "CVE-1900-1234",
    "assignerOrgId": "b3476cb9-2e3d-41a6-98d0-0f47421a65b6",
    "state": "PUBLISHED"
  },
  "containers": {
    "cna": {
      "providerMetadata": {
        "orgId": "b3476cb9-2e3d-41a6-98d0-0f47421a65b6"
      },
      "title": "Buffer overflow in Example Enterprise allows Privilege Escalatio
      "datePublic": "2021-09-08T16:24:00.000Z",
      "problemTypes": [
        {
          "descriptions": [
            {
              "lang": "en",
              "cweId": "CWE-78",
              "description": "CWE-78 OS Command Injection",
              "type": "CWE"
```

```
        }
      ]
    }
  ],
  "impacts": [
    {
      "capecId": "CAPEC-233",
      "descriptions": [
        {
          "lang": "en",
          "value": "CAPEC-233 Privilege Escalation"
        }
      ]
    }
  ],
  "affected": [
    {
      "vendor": "Example.org",
      "product": "Example Enterprise",
      "platforms": [
        "Windows",
        "MacOS",
        "XT-4500"
      ],
      "versions": [
        {
          "version": "1.0.0",
          "status": "affected",
          "lessThan": "1.0.6",
          "versionType": "semver"
        },
        {
          "version": "3.0.0",
          "status": "unaffected",
          "lessThan": "*",
          "versionType": "semver"
        }
      ],
      "defaultStatus": "unaffected"
    }
  ],
  "descriptions": [
    {
      "lang": "en",
      "value": "OS Command Injection vulnerability parseFilename function of
    }
  ],
  "references": [
```

```
        {
          "url": "https://example.org/ESA-22-11-CVE-1900-1234"
        }
      ]
    }
  }
}
```

Chapter 4

Thesis Objectives

The previous chapters introduced threat modeling, digital supply chains, kill chains, and CVEs showing how each of them plays a fundamental role in modern cybersecurity. In particular, Chapter 3 highlighted that the main issue with supply chains lies in the fact that they are no longer just about the movement of goods (raw materials, products, etc..) but also about an interconnected digital network composed of software, cloud services, partners, IoT devices and so on.
This leads to several consequences: even with very strong internal defenses (firewalls, segmentation, monitoring etc..) a company cannot consider itself secure as it depends on the security level of other actors in the chain. A vulnerability in a supplier or partner can become an entry point for an attacker, who exploits the digital connection to access the company's systems. As a result, visibility and control are limited since it is not possible to monitor or manage all the infrastructures of the supply chain, so there is never complete certainty of having a high security level.
In this context, threat modeling frameworks face some difficulties. They work very well when applied to the internal perimeter (to systems that are known and under the control of the organization). In digital supply chains, however, one is faced with a network of external dependencies where it is not always clear how a supplier develops its software, which libraries it uses or which security practices it follows. For this reason, applying threat modeling to the entire chain remains a complex task.

The goal of this thesis is to analyze an existing framework, TAMELESS, which already provides strong support for threat modeling and analysis in hybrid environments, and to propose an extension that integrates CVEs, optimizing its use in predominantly software-based scenarios. The aim is to make it a functional framework for threat analysis of digital supply chains in order to address the challenges linked to their complexity.

4.1 Thesis Workflow

The thesis workflow can be described in three main steps:

1. Analyze and adapt the TAMELESS model to make it more effective in software-based environments and more suitable for addressing threats of digital supply chains.

2. Integrate into the model the Common Vulnerabilities and Exposures (CVE), with a focus on the Software category introduced in Chapter 3. An easy patch management will also be included: if a software component is vulnerable to a specific CVE, the model will provide information on the availability of a fixed version (patch) that is not affected by the vulnerability.

3. Update and enhance the GUI so that it supports the new features and provides a more user friendly interface.

The final result will be an extended and optimized framework, able to support threat analysis more effectively in complex software scenarios and digital supply chains. Finally, thanks to the integration of CVE and patch management, the framework will be able to represent vulnerabilities and possible mitigations in a more realistic way.

Chapter 5

Approach and Model

This chapter introduces and analyzes the TAMELESS framework in detail.
Specifically, it describes the components, their properties and the relationships that connect them. It also explains the derivation rules and the modifications that have been applied to the original model in order to make it suitable for software environments.
A particular focus is given to the integration of the CVEs and patch management, which are essential for the security and reliability of today's systems.

5.1 Introduction to TAMELESS

As introduced in Chapter 2, the Threat & Attack ModEL Smart System (TAMELESS) is a threat analysis tool that provides users with a comprehensive view of the system's security across the human, physical, and cyber dimensions.
Specifically, TAMELESS provides a graphical view of the attack propagation,allowing users to introduce new protection and monitoring components. The tool can then be rerun with the updated information to generate the new threat model (the process can be repeated until the resulting model is satisfactory).
In short, once system components, their properties and their relationships are defined, TAMELESS automatically analyzes them and displays the resulting threats and how they were derived.
TAMELESS also assists architects in identifying countermeasures by suggesting entities that can protect, monitor, detect, restore, repair, or replicate [10].

The following sections describe the key aspects of the model and present the modifications introduced to achieve the goals defined in Chapter 4.

5.2 Components and Security Properties

This section introduces the components of the model together with their security properties.

5.2.1 Components

The original model was composed of two elements: *entity* and *threat*. However, some modifications have been introduced.
Specifically, the current model is composed of three elements:

- **Software:** a software element identified by its name and version, represented in the format *Sofware_Version*.
  An Example: `Express_4.17`.
  In some cases, the component can also be defined in a more generic way, such as `backend`.
- **Threat:** a malicious event that may affect the software.
  Example: `Denial of Service (DoS)`.
- **CVE:** the Common Vulnerabilities and Exposures Identifier (CVE ID), representing a known vulnerability.
  Example: `CVE-2025-7784`.

In this revised model, the concept of *entity* (that previously included human,cyber and physical components) has been replaced by *software*, in order to give a software oriented perspective. Moreover, a third component *cve* has been introduced to include documented vulnerabilities that may affect the considered software.

5.2.2 Security Properties

Security properties can either be assumed or derived through derivation rules. Moreover, these properties are classified into two categories: *basic* and *auxiliary*.

Basic Properties

Basic properties describe the security aspects of the components.
In particular, we define *three* basic properties:

$$\mathcal{P}_{\mathcal{B}} = \{\text{Comp, Malfun, Vul}\}$$

Specifically:

- ***Comp(S,T)***: states that software S has been compromised by threat T. [1]

[1] Comp(S) means that there exists at least one threat for which S is compromised

- ***Malfun(S)***: indicates that software S is malfunctioning (for example, a Firewall misconfiguration).
- ***Vul(S,T)***: states that software S is vulnerable to threat T.

These properties are inherited directly from the original model [10] without modification.

Auxiliary Properties

Auxiliary properties are used to describe either the state of a software component after it has been compromised, or the status of specific vulnerabilities.
In particular, *four* auxiliary properties are defined:

$$\mathcal{P}_{\mathcal{A}} = \{\text{Det, Rest, Fix, Avail}\}$$

Specifically:

- ***Det(S,T)***: indicates that the compromise of software S by threat T has been detected.
- ***Rest(S)***: states that control over software S has been restored after a compromise.
- ***Fix(S)***: states that malfunction of software S has been resolved.
- ***Avail(C)***: indicates that a patch has been released for cve C.

In the original model , the set of auxiliary properties included just three elements. The new model introduces the fourth property *Avail* that specifies whether a patch has been released for a specific CVE.

Assumed and Derived Properties

Assumed properties are those that are explicitly stated as a part of the security assumptions.
The original set of assumed properties has been extended with the property αAvail and is defined as follows:

$$\mathcal{P}_{\alpha} = \{\alpha\text{Comp}, \alpha\text{Malfun}, \alpha\text{Vul}, \alpha\text{Avail}\}$$

Derived properties, on the other hand, represent those properties that may become true through the application of derivation rules:

$$\mathcal{P}_{\kappa} = \{\kappa\text{Comp}, \kappa\text{Malfun}, \kappa\text{Vul}, \kappa\text{Det}, \kappa\text{Rest}, \kappa\text{Fix}\}$$

5.3 Relations and High-Level Properties

This section describes the relationships between components and introduces their high level properties.

5.3.1 Relations

Relations are a fundamental concept in TAMELESS, as they provide a way to describe interconnected system and, in particular, the main aspects of digital supply chains,where different elements depend on each other and potential risks may propagate through these dependencies.
Relations can be established:

- between software
- between software and CVEs
- between CVEs and threats
- between threats and software

This work extends the model by introducing relations related to CVEs, while the other ones remain as defined in the original model [10].

Software–Software Relations

The set of relations between software components is derived from the original model [10] and is defined as follows:

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

Specifically:

- ***Contain(S1,S2):*** indicates that software S1 contains software S2.
- ***Control(S1,S2):*** indicates that software S1 controls software S2.
- ***Connect(S1,S2,S3):*** indicates that software S1 connects software S2 with software S3.[2]
- ***Depend(S1,S2):*** indicates that the functionality of software S1 depends on software S2.
- ***Check(S1,S2):*** indicates that software S1 verifies the functionality of software S2.
- ***Replicate(S1,S2):*** indicates that software S1 replicates software S2.

[2]The connect relation is unidirectional

Software–Threat Relations

These relations make it possible to identify whether software components can protect each other or spread a specific threat, which is important for the derivation rules.
Similarly to the relations between software, the set of relations between software and threat is taken from the original model [10] and is defined as follows:

$$\mathcal{R} = \{\text{Protect, Monitor, Spread}\}$$

Specifically:

- ***Protect(S1,S2,T):*** states that software S1 protects software S2 against threat T.
- ***Monitor(S1,S2,T):*** states that software S1 monitors software S2 for threat T.
- ***Spread(S,T):*** states that software S spreads threat T.

Software–CVE Relations

This model introduces a new set of relations between software and CVEs. In particular, this set contains only one element, defined as follow:

$$\mathcal{R} = \{\text{affectedBy}\}$$

Specifically:

- ***affectedBy(S,C):*** states that software S meets the conditions that make it vulnerable to cve C.

Such conditions may include the software version, operating system (OS), and privileges. This relation is fundamental because it allows mapping a software component to a specific CVE.

Threat–CVE Relations

The final set of relations describes the connection between CVEs and threats.
As in the previous case, this set is newly introduced in the model and is defined as follows:

$$\mathcal{R} = \{\text{Exploit}\}$$

Specifically:

- ***Exploit(C,T):*** indicates that cve C can be exploited by threat T.

5.3.2 High-Level Properties

High level properties make it easier to represent the overall state of a software system including its components, dependencies and connections.
This set of properties is derived from the original model [10] and is defined as follows:

$$\mathcal{P}_{\mathcal{H}} = \{\text{Valid, Defended, Safe, Monitored, Replicated, Checked}\}$$

Specifically:

- ***Valid(S):*** a software S is considered valid if it has not been compromised and is not subject to malfunction.
- ***Defended(S1,T):*** a software S1 is considered defended against a threat T if there exists another software S2 that protects S1 against T and, S2 is valid.
- ***Safe(S,T):*** a software S is considered safe from a threat T if it is either defended against T or not vulnerable to T.
- ***Monitored(S1,T):*** a software S1 is considered monitored with respect to a threat T if there exists another software S2 that monitors S1 for T and S2 is valid.
- ***Replicated(S1):*** a software S1 is considered replicated if there exists another software S2 that replicates S1 and S2 is valid.
- ***Checked(S1):*** a software S1 is considered checked if there exists another software S2 that verifies the functionality of S1 and S2 is valid.

To make definitions clearer, the table 5.1 reports each property together with its logical formulation.

Valid(S)	:=	$\neg Comp(S) \land \neg Malfun(S)$
Defended(S1,T)	:=	$\exists S2.\ Protect(S2, S1, T) \land Valid(S2)$
Safe(S,T)	:=	$\neg Vul(S, T) \lor Defended(S, T)$
Monitored(S1,T)	:=	$\exists S2.\ Monitor(S2, S1, T) \land Valid(S2)$
Checked(S1)	:=	$\exists S2.\ Check(S2, S1) \land Valid(S2)$
Replicated(S1)	:=	$\exists S2.\ Replicate(S2, S1) \land Valid(S2)$

Table 5.1. High level properties.

5.4 Derivation Rules

This section describes the derivation rules,which represent a fundamental concept in TAMELESS. They allow us to determine whether a software component can be compromised, malfunctioning, vulnerable, restorable, fixed or patched.

5.4.1 Rules for Vulnerabilities

If there exists a cve C that a threat T can exploit, and a software S satisfies the conditions specified by C, then S can be vulnerable to T:

$$\textbf{Exploit(C,T)} \land \textbf{affectedBy(S,C)} \rightarrow \kappa\textbf{Vul(S,T)}$$

Table 5.2. Derivation rule for vulnerabilities.

This rule allows us to determine whether a software S can be vulnerable to a specific threat T. In this model, the rule differs from the original formulation as it has been adapted to include the CVE system.
Finally, the rule potentiallyVul which was part of the original model has been removed.

5.4.2 Rules for Compromise

In this model, *four* derivation rules are defined for the compromised state:

1. *A software S can be compromised by a threat T if S is vulnerable to T and is not defended against T.*

2. *If there exists a software S2 that controls a software S1, S2 can spread a specific threat T, S1 is not defended against T and S2 can be compromised (by any threat, regardless of which one), then software S1 can be compromised by threat T.*

3. *If a software S1 contains (or is contained in) a software S2, S2 can spread a specific threat T, S1 is not defended against T and S2 can be compromised (by any threat, regardless of which one), then software S1 can be compromised by threat T.*

4. *If there exists a software S3 that connects a software S2 to a software S1, S2 can spread a specific threat T, S1 is not defended against T, S2 can be compromised (by any threat, regardless of which one) and either S3 can be compromised (by any threat) or S3 is not defended against T, then S1 can be compromised by threat T.*

Table 5.3 reports the rules in logical expressions:

$$\kappa\textbf{Vul(S,T)} \land \neg\textbf{Def(S,T)} \rightarrow \kappa\textbf{Comp(S,T)}$$

$$\textbf{Control(S2,S1)} \land \kappa\textbf{Comp(S2)} \land \neg\textbf{Defended(S1,T)} \land \textbf{Spread(S2,T)} \rightarrow \kappa\textbf{Comp(S1,T)}$$

$$(\textbf{Contain(S1,S2)} \lor \textbf{Contain(S2,S1)}) \land \kappa\textbf{Comp(S2)} \land \neg\textbf{Defended(S1,T)} \land \textbf{Spread(S2,T)} \rightarrow \kappa\textbf{Comp(S1,T)}$$

$$\textbf{Connect(S3,S2,S1)} \land \kappa\textbf{Comp(S2)} \land \textbf{Spread(S2,T)} \land \neg\textbf{Defended(S1,T)} \land (\kappa\textbf{Comp(S3)} \lor \neg\textbf{Defended(S3,T)}) \rightarrow \kappa\textbf{Comp(S1,T)}$$

Table 5.3. Derivation rules for compromised.

These rules allow us to determine and analyze how threats can propagate across systems and how software components can compromise each other.
In particular, the rules make it possible to better identify the weaknesses in the digital supply chain.
To ensure compatibility with the CVE system, some modifications have been introduced compared to the original model :

- The first rule has been introduced to define whether a software can be compromised independently of its relationships with other software. In particular, it states that the software must not only be vulnerable to a specific threat but also lack any external protection against it.
- In all rules, the condition safe has been replaced by the condition defended in order to avoid inconsistencies in dependency relations. For example, if a software A has full control over a software B, and A is compromised by a threat T, then T directly compromises B as well, regardless of whether exists a vulnerability that can be exploited in B.

5.4.3 Rules for Malfunctions

There are two derivation rules for malfunctioned state:

1. *A compromised software S can malfunction.*
2. *If a software S1 depends on a software S2 and S2 malfunctions, then S1 can also malfunction.*

Table 5.4 reports the rules in logical expressions:

$$\kappa\textbf{Comp(S,T)} \rightarrow \kappa\textbf{Malfun(S)}$$

$$\textbf{Depend(S1,S2)} \wedge \kappa\textbf{Malfun(S2)} \rightarrow \kappa\textbf{Malfun(S1)}$$

Table 5.4. Derivation rules for malfunctioned.

These rules are inherited from the original model [10] without modifications.

5.4.4 Rules for Threat Detection

When a software S can be compromised by a threat T and S is monitored for threat T by some uncompromised software, then T can be detected for S:

$$\kappa\textbf{Comp(S,T)} \wedge \textbf{Monitored(S,T)} \rightarrow \kappa\textbf{Det(S)}$$

Table 5.5. Derivation rule for detecting threats.

This rule is taken from the original model [10] without modifications.

5.4.5 Rules for Software Restoration

When a threat T can be detected for the software S and S has been replicated, then S can be restored:

$$\kappa\textbf{Det(S,T)} \wedge \textbf{Replicated(S)} \rightarrow \kappa\textbf{Rest(S)}$$

Table 5.6. Derivation rule for Restoring software.

This rule is inherited from the original model [10] without modifications.

5.4.6 Rules for Fixes

When a software S malfunctions and this malfunction can be detected (i.e: S is checked), then S can be fixed:

$$\kappa\textbf{Malfun(S)} \wedge \textbf{Checked(S)} \rightarrow \kappa\textbf{Fix(S)}$$

Table 5.7. Derivation rule for Fixing software.

This rule is adopted from the original model [10] without modifications.

5.4.7 Rules for Patching

If a software S is affected by the cve C , and a patch for that cve has been released, then software S can be patched (i.e: updated to a more recent version not affected by C):

$$\textbf{affectedBy(S,C)} \wedge \textbf{Avail(C)} \rightarrow \kappa\textbf{Patch(S,C)}$$

Table 5.8. Derivation rule for Patched.

This rule has been introduced with this work.

Chapter 6

Implementation and Validation

The previous chapter discussed the TAMELESS model in detail, together with the modifications introduced to achieve part of the objectives outlined in Chapter 4. This chapter focuses on the implementation and describes the changes made both in Prolog and in the graphical user interface (GUI).
Finally, two use cases are presented, followed by the validation of the model on these scenarios, in order to provide a practical demonstration in contexts that resemble real-world situations.

6.1 Implementation

In this section, the attention moves from the theoretical aspects of the model to its actual implementation.
The discussion is divided into two main parts:

1. The first part concerns the modifications made to the Prolog side.
2. The second part focuses on the GUI, where the goal was to implement the new features and to improve the usability.

6.1.1 Prolog Implementation

For the implementation of the model in Prolog, the knowledge base[1] has been divided into two separate files:

1. The first contains the high level properties and the derivation rules.
2. The second defines the specific relations and properties of the system under analysis.

The first file, named `rule.p`, implements the general rules and is the main focus of this subsection.

[1] In Prolog, the knowledge base consists of facts and rules

High-Level Properties

At the beginning of the file, we find the high level properties.

In the original implementation of the model, the high level properties were defined as follows:

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

%%Protected
protected(A,T):-
                protect(B,A,T), usable(B).

%%Safe
notSafe(A,T):-
                e(A), t(T),  (\+protected(A,T),   canbeVul(A,T)).

%%Monitored
monitored(A,T):-
                monitor(B,A,T), usable(B).

%%Replicate
replicated(A):-
                replica(B,A), usable(B).

%%Checked
checked(A):-
                check(B,A), usable(B).
```

Listing 6.1. Original high-Level properties

However, this was not fully consistent with the original model , where the properties where defined with different names.

The new implementation address this issue and redefines the high level properties as follows:

```
%%Valid
valid(S):-
           software(S),(\+canbeComp(S,_T), \+canbeMalfun(S)).

%%Defended
defended(S1,T):-
           protect(S2,S1,T), valid(S2),
           write("\n -> ("),
```

```
            write(S2), write(") protect ("), write(S1),
            write(") against threat ("), write(T),
            write(")\n").

%%Safe
safe(S,T):-
            software(S), threat(T),
           ( defended(S,T) ; \+canbeVul(S,T) ).

%%Monitored
monitored(S1,T):-
            monitor(S2,S1,T), valid(S2),
            write("\n -> ("),
            write(S2), write(") monitor ("), write(S1),
            write(") for threat ("), write(T),
            write(")\n").

%%Replicated
replicated(S1):-
            replicate(S2,S1), valid(S2).

%%Checked
checked(S1):-
            check(S2,S1), valid(S2).
```

Listing 6.2. New high-Level properties

The main modifications are listed below:

- The property names have been aligned with the terminology used in the Chapter 5.
- The *monitored* property now includes print statements.
- The property *safe* replaces the previous *notSafe*.

Derivation Rules

Immediately after the high level properties we find the derivation rules.

In the original implementation, *canbeVul* was defined as follows:

```
canbeVul(A,T):- potentiallyVul(A,T), canbeMalfun4Dep(A).
```

Listing 6.3. Original canbeVul

With the introduction of CVEs, the rule has been completely revised. Its new implementation is the following:

```
canbeVul(S,T):-
    exploit(CVE,T),
    affectedBy(S,CVE),
    write('\n-> ('),
    write(S),
    write(") meet conditions for ("),
    write(CVE),
    write(") exploitable by ("),
    write(T),
    write(")\n"),
    (
        avail(CVE) ->
        write("\n -> ("),
        write(CVE),
        write(") patch ("),
        write("is available"),
        write(")\n")
        ;
        write("\n -> ("),
        write(CVE),
        write(") patch ("),
        write("is NOT available"),
        write(")\n")
    ).
```

Listing 6.4. New canbeVul

Specifically,the changes introduced are:

- Removal of *potentiallyVul* and *canbeMalfun4Dep*.
- Introduction of *exploit* and *affectedBy* to integrate the CVE system.
- Addition of print statements providing information about the CVE affecting a given software component S, and whether a patch for that CVE has been released. This not only improves the readability of the output, but also provides useful input for the graph generated by the GUI.

A new rule has been added concerning the *spread*. This addition has just an implementative purpose, aimed at producing well structured output. The logical semantics remain unchanged, since the only condition is the spread itself:

```
canSpread(S,T):-
        spread(S,T),
        write('\n-> ('), write(S),
        write(') can spread ('), write(T),
        write(')\n').
```

Listing 6.5. canSpread

The set of derivation rules concerning compromise has also been revised. The original implementation was as follows:

```
canbeComp(A,T2):-
                control(B,A),
                canbeComp(B,_T1),
                spread(B,T2),
                notSafe(A,T2),
                write('\n-> ('),
                write(A),
                write(') can be compromises through ('),
                write(B),
                write(') by ('),
                write(T2),
                write(')').


canbeComp(A,T2):-
                connect(C,B,A),
                canbeComp(B,_T1),
                spread(B,T2),
                notSafe(A,T2),
                (canbeComp(C,_T3); \+protected(C,T2)),
                write('\n-> ('),
                write(A),
                write(') can be compromises through ('),
                write(B),
                write(') by ('),
                write(T2),
                write(')').

canbeComp(A,T2):-
                (contain(A,B); isContained(A,B)),
                canbeComp(B,_T1),
                spread(B,T2),
                notSafe(A,T2),
                write('\n-> ('),
                write(B),
                write(') compromises ('),
                write(A),
                write(') by ('),
                write(T2),
                write(')').
```

Listing 6.6. Original canbeComp

The implementation in the new model is:

```
canbeComp(S1,T):-
                (   contain(S1,S2); isContained(S1,S2) ),
                canSpread(S2,T),
                \+defended(S1,T),
                canbeComp(S2,_T2),
                write('\n-> ('),
                write(S2),
                write(') compromises ('),
                write(S1),
                write(') by ('),
                write(T),
                write(')\n').

canbeComp(S1,T):-
                control(S2,S1),
                canSpread(S2,T),
                \+defended(S1,T),
                canbeComp(S2,_T),
                write('\n-> ('),
                write(S1),
                write(') can be compromised through ('),
                write(S2),
                write(') by ('),
                write(T),
                write(')\n').

canbeComp(S1,T1):-
                connect(S3,S2,S1),
                canSpread(S2,T1),
                \+defended(S1,T1),
                canbeComp(S2,_T2),
                (canbeComp(S3,_T3); \+defended(S3,T1)),
                write('\n-> ('),
                write(S1),
                write(') can be compromised through ('),
                write(S2),
                write(') by ('),
                write(T1),
                write(')').

canbeComp(S,T):-
               \+defended(S,T),
               canbeVul(S,T),
               write('\n-> ('),
               write(S),
               write(") can be compromised by ("),
               write(T),
```

```
                write(")\n").
```

Listing 6.7. New canbeComp

The main modifications introduced are the following:

- Introduction of a fourth rule, which allow us to verify whether a software component S can be compromised by a threat T without considering its relationships with other software.
- Replacement of *notSafe* with *notDefended*, as discussed in Chapter 5.
- Replacement of *spread* with the rule *canSpread* , which handles the generation of the corresponding output messages.

A new derivation rule has been introduced in order to determine whether a software component S is affected by a specific vulnerability CVE and whether an update is available. If both conditions hold, the software can be patched and the output message indicates this possibility. The rule is defined as follows:

```
canbePatch(S,CVE):-
                affectedBy(S,CVE), avail(CVE),
                write('\n-> ('),
                write(S),
                write(') can be patched for vulnerability ('),
                write(CVE),
                write(')\n').
```

Listing 6.8. New canbePatch

The last modification concerns the *canbeDet* derivation rule:

```
canbeDet(A,T):- canbeComp(A,T), monitored(A,T)
```

Listing 6.9. Original canbeDet

In the revised model, print statements have been added in order to display when a detection of a given threat is possible:

```
canbeDet(S,T):-
          canbeComp(S,T), monitored(S,T),
```

```
            write('\n-> ('),
            write(S),
            write(') can be detected if compromised by threat ('),
            write(T),
            write(')\n').
```

Listing 6.10. New canbeDet

6.1.2 Graphical User Interface (GUI) Implementation

This section describes the most relevant changes made to the Tameless GUI in order to integrate the new version of the model and to make the interface itself easier to use.

Backend

The backend is responsible for handling communication with the database and exposing the APIs required by the frontend. Its structure is organized into several directories, each containing files dedicated to specific functionalities. In particular, the backend of TAMELESS is implemented in *Node.js*, while *MongoDB* is used as the database.

The first important directory is *controllers* as each file within it defines how requests are processed and how responses are generated.
In particular, a new file `cve.controller.js` containing four controllers has been added:

- **createCveHandler:** controller for inserting a new CVE.
- **getAllCveHandler:** controller for retrieving all CVEs.
- **updateCveHandler:** controller for updating a CVE.
- **deleteCveHandler:** controller for deleting a CVE.

Moreover, the file `entities.controller.js` and all the controllers it contains have been renamed, since the model no longer includes a generic entity, but instead uses the concept of software. Therefore, the file has been renamed to `software.controller.js` and the controllers have been renamed accordingly to *createSoftwareHandler*, *getAllSoftwareHandler*, *updateSoftwareHandler* and *deleteSoftwareHandler*.

The directory *db/models* contains the files that define the schemas for the MongoDB database.
A new file `Cve.js` has been added to define the schema for the CVEs:

```
import mongoose from "mongoose";

const CveSchema = mongoose.Schema(
    {
        name: {
            type: String,
            require: true,
            maxLength: 50
        },
        description: {
            type: String,
            require: false,
            maxLength: 1000
        },
    },
    {
        toJSON: {
          virtuals: true,
          transform: (doc, ret) => {
            ret.id = ret._id; //Map _id to id
            delete ret._id; //Remove original _id field
            delete ret.__v; //Remove Mongoose's versioning key
          },
        },
    }
);

export default mongoose.model("Cve", CveSchema, 'cve');
```

Listing 6.11. Cve.js

Moreover, the file `Entity.js` has been renamed to `Software.js` and the *type* field of the entity schema has been removed. This field was previously used to specify whether an entity was physical, human or cyber. However, since the new model focuses on a software-based environment, this information is no longer needed. Finally, *EntitySchema* has been renamed to *SoftwareSchema*.

The directory *json_schemas* contains the JSON schemas used to validate the structure of the data and to ensure that the information stored in or retrieved from the database respects the expected format. The file `cve_schema.json` has been added, while the other files have been modified to align with the decision to remove the entity type and to rename the entity to software.

The directory *routes* contains the files that define the API routes. Each route

specifies an endpoint and associates it with the corresponding controller. A new file `cve.route.js` has been added to handle the routes related to CVEs:

```
import express  from 'express';
import { validate, cveSchema } from '../utils/validator.js';
import { createCveHandler, getAllCveHandler, updateCveHandler,
   deleteCveHandler } from '../controllers/cve.controller.js';

const router = express.Router();

/**Add a new cve */
router.post('/cve', validate({ body: cveSchema }),
   createCveHandler);

/**Get all present cve */
router.get('/cve', getAllCveHandler);

/**Update the selected cve */
router.put('/cve/:cveId', validate({ body: cveSchema }),
   updateCveHandler);

/**Route to delete an cve */
router.delete('/cve/:cveId', deleteCveHandler);

export default router; //To import and use
```

Listing 6.12. cve.route.js

Moreover, the `entities.route.js` file and its routes have been renamed to align with the new software-based model.

The directory *service* contains the files responsible for managing services. Services perform the operations of saving, updating, deleting or retrieving data from the database. A new file `cve.service.js` has been added to define services related to CVEs:

- **createCve:** to save a new CVE.
- **getAllCve:** to retrieve all CVEs.
- **updateCve:** to update a CVE.
- **deleteCve:** to delete a CVE.

Similarly to the previous directories, the file `entity.service.js` has been renamed to `software.service.js` and the services it contains have been updated to *createSoftware* , *getAllSoftware*, *updateSoftware* and *deleteSoftware*.

The last directory *utils* contains two files:

- **validator.js:** defines the JSON validator middleware.
- **prolog_function.js:** provides functions to run Prolog scripts and prepare the JSON responses.

The file `validator.js` has been updated to include the CVE validators and to rename the *entitySchema* as *softwareSchema*.

The file `prolog_functions.js` has also been modified.
Originally,it contained functions to execute Prolog queries and process their results, generating a separate graph for each response statement. However, in large-scale environments this approach is not efficient since a single query may return many results, producing 100-200 (or even more) separate graphs, often with duplicates.
In the new implementation, only one graph is generated for each query. This graph aggregates all possible results without duplicates, providing a complete view from the beginning.
To achieve this, the original *prepareResponse* function has been refactored and replaced by the new *prepareResponseGlobal* function.

Finally, the last important file is *index.js*. In the original implementation, this file was responsible for starting the server and immediately establishing the database connection. The connection to MongoDB was initiated within the *app.listen* callback, meaning that a preliminary script had to be executed to load the DB before launching the server and the client:

```
// activate the server
app.listen(configuration.SRV_PORT, async () => {
    await connectMongoDB();
    console.log(Server listening at ...);
});
```

Listing 6.13. Old index.js

In the revised implementation, this mechanism has been modified. The DB connection is no longer initiated at server startup; instead, it is triggered through a specific API. This endpoint executes a shell script that optionally receives the name of a use case, allowing the user either to load a predefined scenario or to start with an empty database. Only after the script has finished, the system establishes the connection to MongoDB:

```
app.post("/api/db", (req, res) => {
  const { caseName } = req.body; //case or empty
```

```
    console.log(caseName);
    const scriptPath = process.cwd() + "/../db/dump/runInsert.sh";

    const cmd = caseName ?
    bash $scriptPath $caseName :
    bash $scriptPath;

    exec(cmd, async (error, stdout, stderr) => {
      console.log("STDOUT:", stdout);
      console.log("STDERR:", stderr)
      if (error) {
        console.error("Errore script:", error);
        return res.status(500).json(
        { error: "Errore esecuzione script" }
        );
      }

      try {
        await connectMongoDB();
        res.json({ ok: true });
      } catch (err) {
        res.status(500).json(
        { error: "Connessione al DB fallita" }
        );
      }
    });
  });
```

Listing 6.14. New index.js

This design change has two important implications:

1. The database can now be initialized directly from the graphical interface, giving the user the control over whether and what data should be loaded.
2. It enables switching between different use cases without restarting the server, which makes navigation more flexible and easier.

Frontend

The client of TAMELESS has been developed using React. As in the backend, the frontend is organized into directories:

- **APIs:** contains the files for handling API calls.
- **components:** contains the React components.
- **contexts:** contains the files for managing contexts.

- **models:** contains the files that define the different models (Cve, Threat etc..).
- **routes:** contains the files for handling routing.

Similarly to the backend, all files, functions and variables related to entities have been renamed in order to align with the updated model, which no longer relies on the concept of entity but on that of software. Moreover, the necessary changes have been applied to remove the *type* field from the software model.
The first modification concerns the home page. The previous interface was as follows:

Figure 6.1. Original home page

In the new interface,a *Try it now* button has been added to guide the user on how to test the GUI. This button redirects to the use case selection page. Finally, the *page title* has also been redesigned, with a blue color and improved style:

Figure 6.2. New home page

The updated GUI also introduces a dedicated page that allows the user to select a use case and upload its data, simplifying the overall workflow.
This functionality is implemented in the `TryPage.jsx` file located in the */src/components/Mng* directory.
Once the desired use case is selected, the client triggers a server-side API that executes a script to create the necessary collections and populate them, establishes the database connection and then the client sets dbReady state to True and fetches the data.
Finally a green button *Home* has been added, allowing user to return to the home page and thus improving navigation.
Figures 6.3 and 6.4 provide a visual representation of this page.

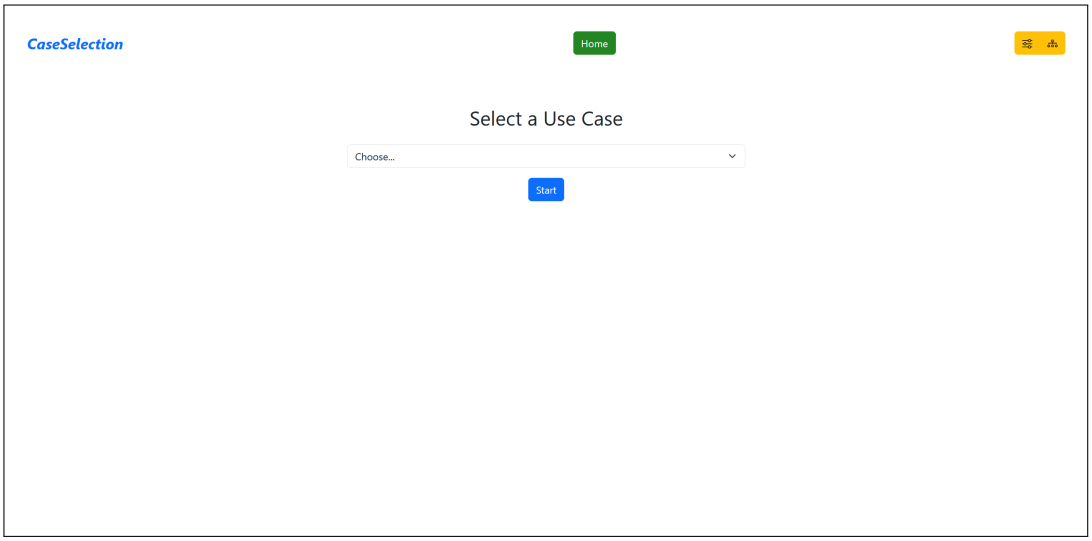

Figure 6.3. Try page

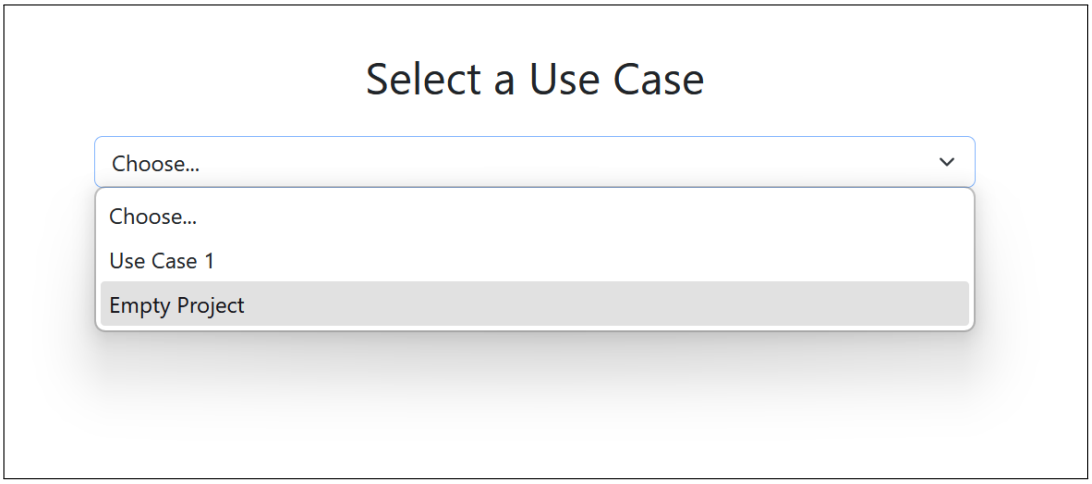

Figure 6.4. Use case selection

The configuration page has also been revised to improve both usability and visual consistency. In addition to the green Home button, an orange *CaseSelection* button has been introduced. This button allows the user to return to the use case case selection page (TryPage) and load a different scenario. The implementation of these buttons is contained in the `MyButtons.jsx` file, located in *src/components/-General*.
Regarding the software management section, as anticipated in the description of server-side modifications, the *Type* field has been removed.
Moreover, a styling issue affecting the lower part of the interface has been fixed: when scrolling down, the DeleteSelected, Cancel and Add buttons previously displayed a background that was partially black and partially white due to inconsistent page styiling (see Figure 6.5).
The problem has been solved by enforcing a uniform background color for the entire application through the following rule added in *src/style/App.css*:

```
    body, html, #root {
    background-color: white !important;
}
```

Finally, the Cve section is provided by the `Cve.jsx` component in the *src/components/Mng* directory.
To better highlight the modifications, Figure 6.5 illustrates the previous interface, while Figures 6.6 and 6.7 show the updated implementation.

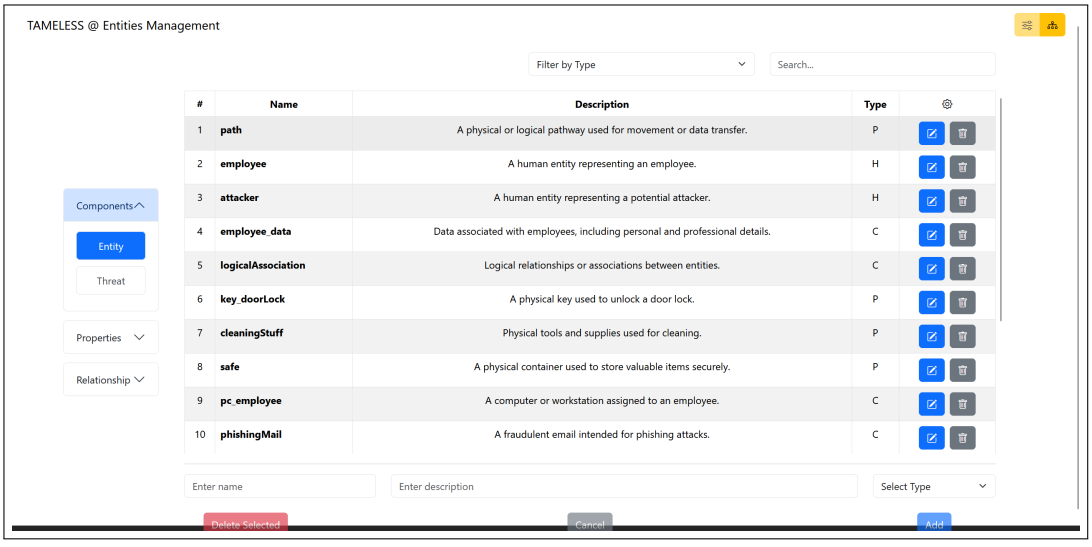

Figure 6.5. Original entities section

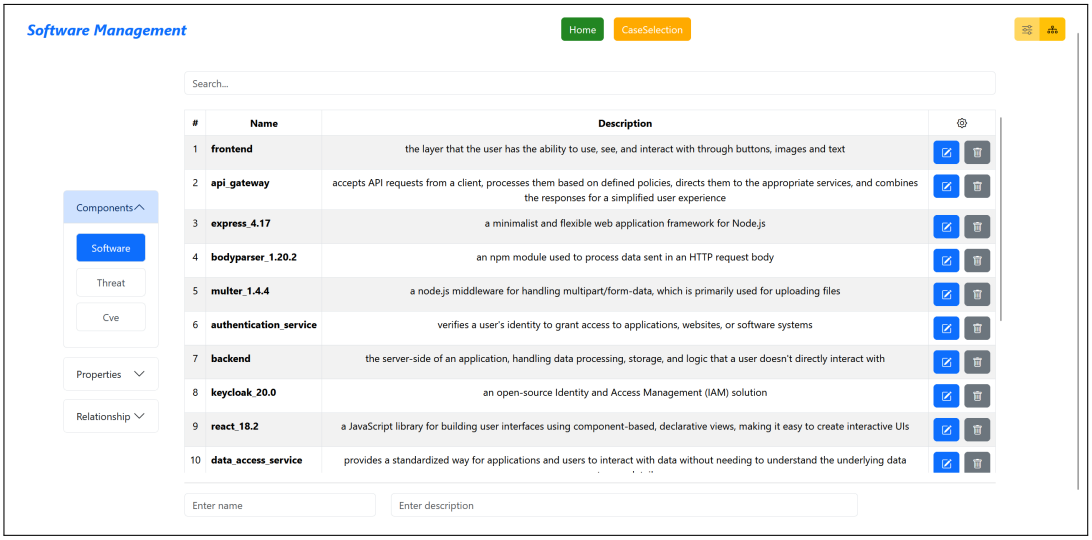

Figure 6.6. Software section

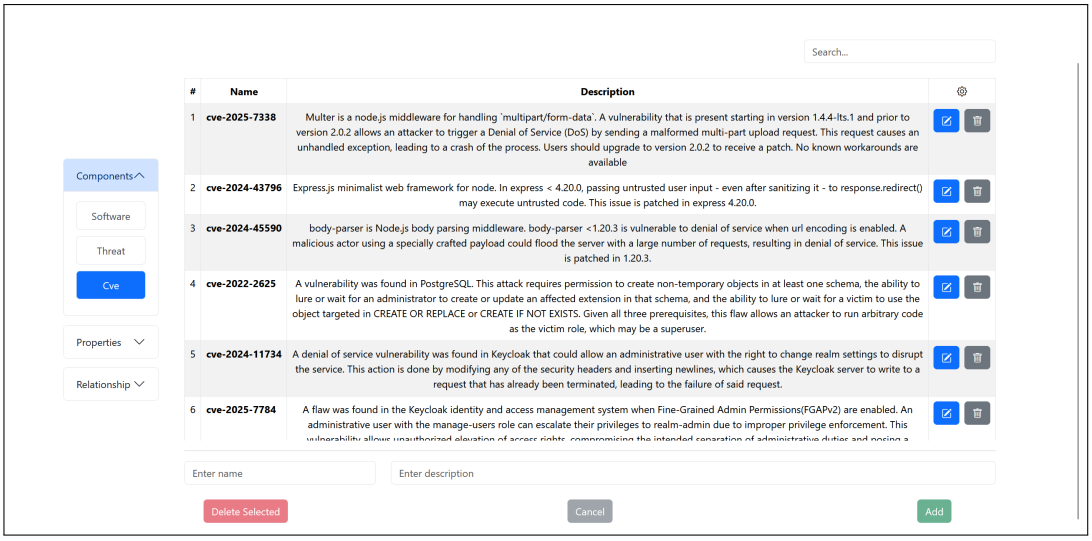

Figure 6.7. CVE section

The last relevant modification concerns the graph component.
As described in the section related to the server-side changes, the previous implementation generated a separate graph for each row of the query results. This approach could produce 100-200 graphs (or even more in large environments), with many duplicates. The updated implementation provides a single graph that immediately offers a comprehensive view of all results returned by the query.
However, there was an issue regarding edge labels: if n relationship existed between two nodes, the system would render n distinct edges. To address this issue, a custom edge component has been developed, implemented in the `EdgeCustom.jsx` file

located in *src/components/Result/Graph*. With this solution, even if multiple relationships with different labels exist between two nodes, the visualization displays a single edge containing all the labels, separated.
An example of a single graph with multiple relationships is shown in Figure 6.8.

Figure 6.8. Graph example

6.2 Validation

This section describes two use cases, followed by the validation of the model on them.

6.2.1 First Use Case

Description

The first use case models a web application composed of several interconnected components: a **frontend**, a **gateway**, a **backend**, an **authentication service**, a **data access service**, and a **database**.
Some of these components relies on specific software packages. In particular:

- the **frontend** is built with `React 18.2`.
- the **backend** includes `Express 4.17` together with the libraries `Multer 1.4.4` and `Bodyparser 1.20.2`.
- the **authentication service** is based on `Keycloak 20.0`.
- the **database** relies on `Postgres 13.8`.
- the **data access service** uses an `ORM 6.0` to interact with the database.

- protection and monitoring mechanisms are provided by `Snort 3.1`, `Prometheus 2.37`, and a **firewall**.

The model incorporates several relevant threats, including *DoS*, *SQL Injection*, *Privilege Escalation*, *Cross-Site Scripting (XSS)*, and *Unauthenticated Access*. It also considers known vulnerabilities (CVEs) affecting specific components and highlights the possibility of threat propagation across the system. This illustrates how the exploitation of a single vulnerability may trigger cascading effects on the entire application.
Figure 6.9 shows the complete structure of the system.

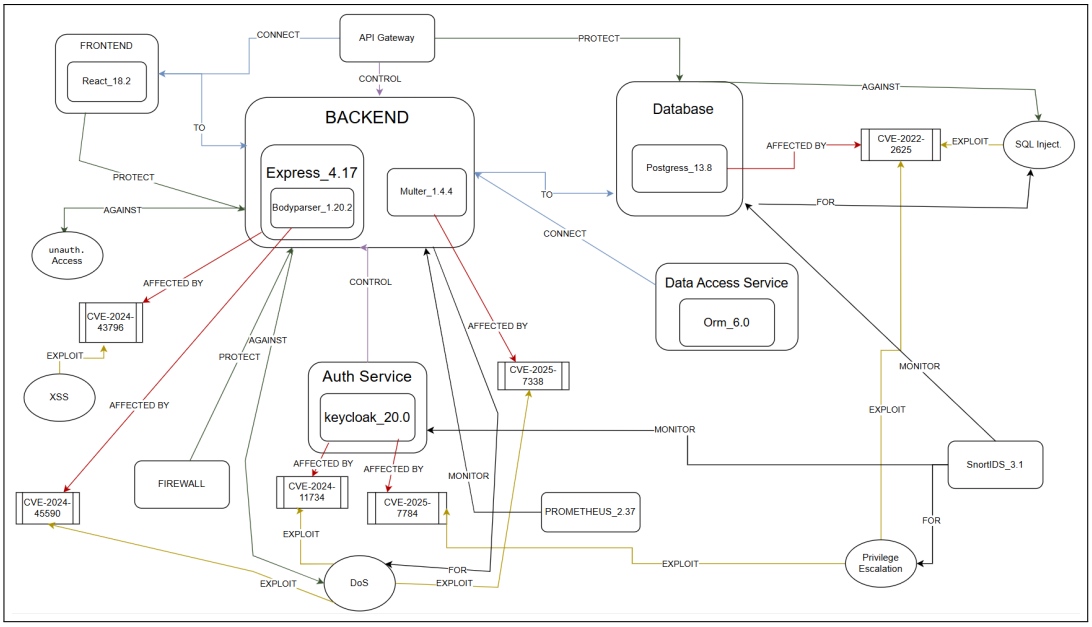

Figure 6.9. First use case schema

Finally, the following extract represents the core part of the Prolog file that implements the use case:

```
affectedBy("multer_1.4.4","cve-2025-7338").
affectedBy("bodyparser_1.20.2","cve-2024-45590").
affectedBy("keycloak_20.0","cve-2024-11734").
affectedBy("postgres_13.8","cve-2022-2625").
affectedBy("express_4.17","cve-2024-43796").
affectedBy("keycloak_20.0","cve-2025-7784")

contain("frontend","react_18.2").
contain("backend","express_4.17").
contain("backend","multer_1.4.4").
contain("express_4.17","bodyparser_1.20.2").
contain("authentication_service","keycloak_20.0").
contain("database","postgres_13.8").
contain("data_access_service","orm_6.0").
```

```
control("api_gateway", "backend").
control("authentication_service", "backend").

connect("api_gateway","frontend","backend").
connect("data_access_service","backend","database").

monitor("snort_3.1","authentication_service",
"privilegeEscalation").
monitor("snort_3.1","database","sqlInjection").
monitor("prometheus_2.37","backend","denialOfService").

protect("firewall","backend","denialOfService").
protect("api_gateway","backend","unauthenticatedAccess").
protect("api_gateway","database","sqlInjection").

exploit("cve-2025-7338","denialOfService").
exploit("cve-2024-45590","denialOfService").
exploit("cve-2024-11734","denialOfService").
exploit("cve-2022-2625","sqlInjection").
exploit("cve-2022-2625","privilegeEscalation").
exploit("cve-2024-43796","xss").
exploit("cve-2025-7784","privilegeEscalation").

spread("backend", "denialOfService").
spread("backend", "sqlInjection").
spread("database", "privilegeEscalation").
spread("frontend", "xss").
spread("express_4.17","denialOfService").
spread("bodyparser_1.20.2","denialOfService").
spread("multer_1.4.4","denialOfService").
spread("keycloak_20.0","denialOfService").
spread("authentication_service","denialOfService").
spread("postgres_13.8","privilegeEscalation").
spread("postgres_13.8","sqlInjection").
spread("keycloak_20.0","privilegeEscalation").
spread("authentication_service","privilegeEscalation").
assMalfun(firewall).
```

Listing 6.15. Prolog useCase1

Validation

The first query executed is ***canbeComp(backend,denialOfService)***,in order to determine whether the backend can be compromised by a DoS. The Prolog output is the following:

```
-> (express_4.17) can spread (denialOfService)
-> (bodyparser_1.20.2) can spread (denialOfService)
```

```
-> (bodyparser_1.20.2) meet conditions for (cve-2024-45590)
   exploitable by (denialOfService)
 -> (cve-2024-45590) patch (is NOT available)
-> (bodyparser_1.20.2) can be compromised by (denialOfService)
-> (bodyparser_1.20.2) compromises (express_4.17) by
   (denialOfService)
-> (express_4.17) compromises (backend) by (denialOfService)
true;

-> (express_4.17) meet conditions for (cve-2024-43796)
   exploitable by (xss)
 -> (cve-2024-43796) patch (is NOT available)
-> (express_4.17) can be compromised by (xss)
-> (express_4.17) compromises (backend) by (denialOfService)
true ;

-> (multer_1.4.4) can spread (denialOfService)
-> (multer_1.4.4) meet conditions for (cve-2025-7338)
   exploitable by (denialOfService)
 -> (cve-2025-7338) patch (is NOT available)
-> (multer_1.4.4) can be compromised by (denialOfService)
-> (multer_1.4.4) compromises (backend) by (denialOfService)
true ;

-> (authentication_service) can spread (denialOfService)
-> (keycloak_20.0) can spread (denialOfService)
-> (keycloak_20.0) meet conditions for (cve-2024-11734)
   exploitable by (denialOfService)
 -> (cve-2024-11734) patch (is NOT available)
-> (keycloak_20.0) can be compromised by (denialOfService)
-> (keycloak_20.0) compromises (authentication_service) by
   (denialOfService)
-> (backend) can be compromises through (authentication_service)
   by (denialOfService)
true ;

-> (keycloak_20.0) meet conditions for (cve-2025-7784)
   exploitable by (privilegeEscalation)
 -> (cve-2025-7784) patch (is NOT available)
-> (keycloak_20.0) can be compromised by (privilegeEscalation)
-> (keycloak_20.0) compromises (authentication_service) by
   (denialOfService)
-> (backend) can be compromises through (authentication_service)
   by (denialOfService)
true ;

-> (keycloak_20.0) can spread (privilegeEscalation)
```

```
   -> (keycloak_20.0) meet conditions for (cve-2024-11734)
      exploitable by (denialOfService)
    -> (cve-2024-11734) patch (is NOT available)
   -> (keycloak_20.0) can be compromised by (denialOfService)
   -> (keycloak_20.0) compromises (authentication_service) by
      (privilegeEscalation)
   -> (backend) can be compromises through (authentication_service)
      by (denialOfService)
   true ;

   -> (keycloak_20.0) meet conditions for (cve-2025-7784)
      exploitable by (privilegeEscalation)
    -> (cve-2025-7784) patch (is NOT available)
   -> (keycloak_20.0) can be compromised by (privilegeEscalation)
   -> (keycloak_20.0) compromises (authentication_service) by
      (privilegeEscalation)
   -> (backend) can be compromises through (authentication_service)
      by (denialOfService)
   true.
```

Listing 6.16. Prolog canbeComp(backend,denialOfService)

This result matches the expected behavior, since there is the firewall that protects backend from DoS, but it has been modeled as malfunctioning. Although `express_4.17` (and the `authentication_service`) are not directly subject to DoS, they are able to spread the threat, so if they are compromised they can cause the backend to be compromised according to the model's derivation rules. Figure 6.10 shows the graph corresponding to the output of the query just executed and reflects the same conclusions.

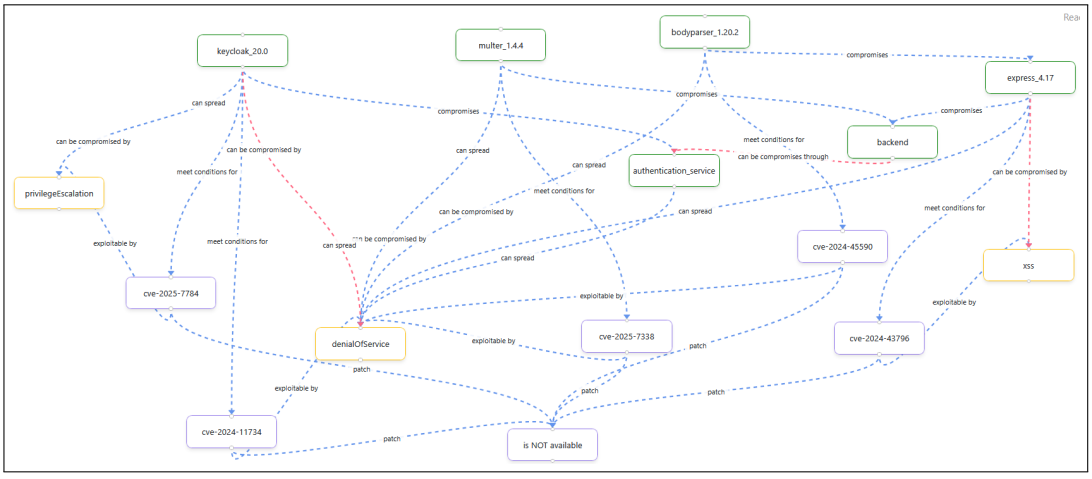

Figure 6.10. canbeComp(backend,denialOfService) - no protection

Now, if we assume that the `authentication_service` can't spread DoS, the Prolog output becomes (Figure 6.11 displays the corresponding graph):

```
   -> (express_4.17) can spread (denialOfService)
```

```
-> (bodyparser_1.20.2) can spread (denialOfService)
-> (bodyparser_1.20.2) meet conditions for (cve-2024-45590)
   exploitable by (denialOfService)
 -> (cve-2024-45590) patch (is NOT available)
-> (bodyparser_1.20.2) can be compromised by (denialOfService)
-> (bodyparser_1.20.2) compromises (express_4.17) by
   (denialOfService)
-> (express_4.17) compromises (backend) by (denialOfService)
true;
-> (express_4.17) meet conditions for (cve-2024-43796)
   exploitable by (xss)
 -> (cve-2024-43796) patch (is NOT available)
-> (express_4.17) can be compromised by (xss)
-> (express_4.17) compromises (backend) by (denialOfService)
true ;

-> (multer_1.4.4) can spread (denialOfService)
-> (multer_1.4.4) meet conditions for (cve-2025-7338)
   exploitable by (denialOfService)
 -> (cve-2025-7338) patch (is NOT available)
-> (multer_1.4.4) can be compromised by (denialOfService)
-> (multer_1.4.4) compromises (backend) by (denialOfService)
true ;
```

Listing 6.17. Prolog canbeComp(backend,denialOfService) - no auth. Service

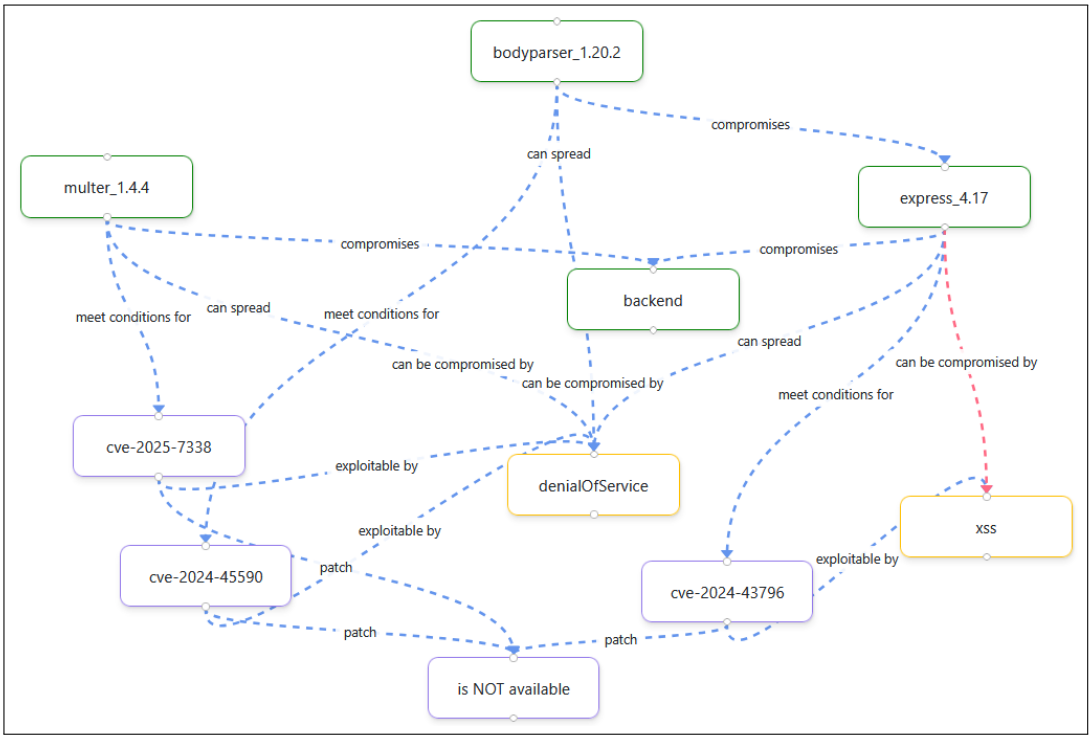

Figure 6.11. canbeComp(backend,denialOfService) - no auth. Service

Finally, if the firewall is modeled as functioning, the `backend` is protected from DoS (Figure 6.12):

```
-> (express_4.17) can spread (denialOfService)
 -> (firewall) protect (backend) against threat (denialOfService)
-> (multer_1.4.4) can spread (denialOfService)
 -> (firewall) protect (backend) against threat (denialOfService)
-> (authentication_service) can spread (denialOfService)
 -> (firewall) protect (backend) against threat (denialOfService)
 -> (firewall) protect (backend) against threat (denialOfService)
false.
```

Listing 6.18. Prolog canbeComp(backend,denialOfService) - protected

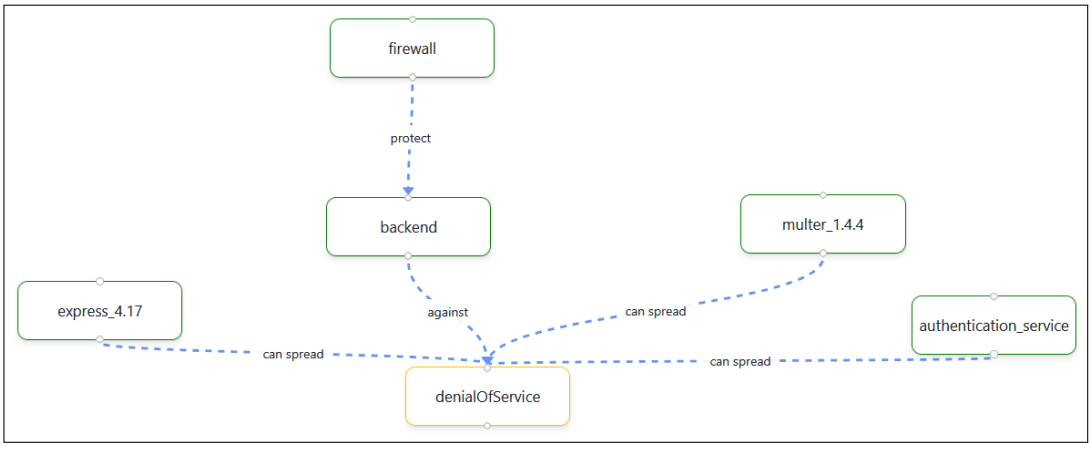

Figure 6.12. canbeComp(backend,denialOfService) - protected

Assume now that a patch for CVE-2022-2625 has been released and execute the query ***canbeComp(database,privilegeEscalation)***.
The output is:

```
-> (postgres_13.8) can spread (privilegeEscalation)
-> (postgres_13.8) meet conditions for (cve-2022-2625)
   exploitable by (sqlInjection)
 -> (cve-2022-2625) patch (is available)
-> (postgres_13.8) can be compromised by (sqlInjection)
-> (postgres_13.8) compromises (database) by
   (privilegeEscalation)
true ;

-> (postgres_13.8) meet conditions for (cve-2022-2625)
   exploitable by (privilegeEscalation)
 -> (cve-2022-2625) patch (is available)
```

```
-> (postgres_13.8) can be compromised by (privilegeEscalation)
-> (postgres_13.8) compromises (database) by
   (privilegeEscalation)
true
```

Listing 6.19. Prolog canbeComp(database,privilegeEscalation)

The graphical representation of the output is shown in Figure 6.13.

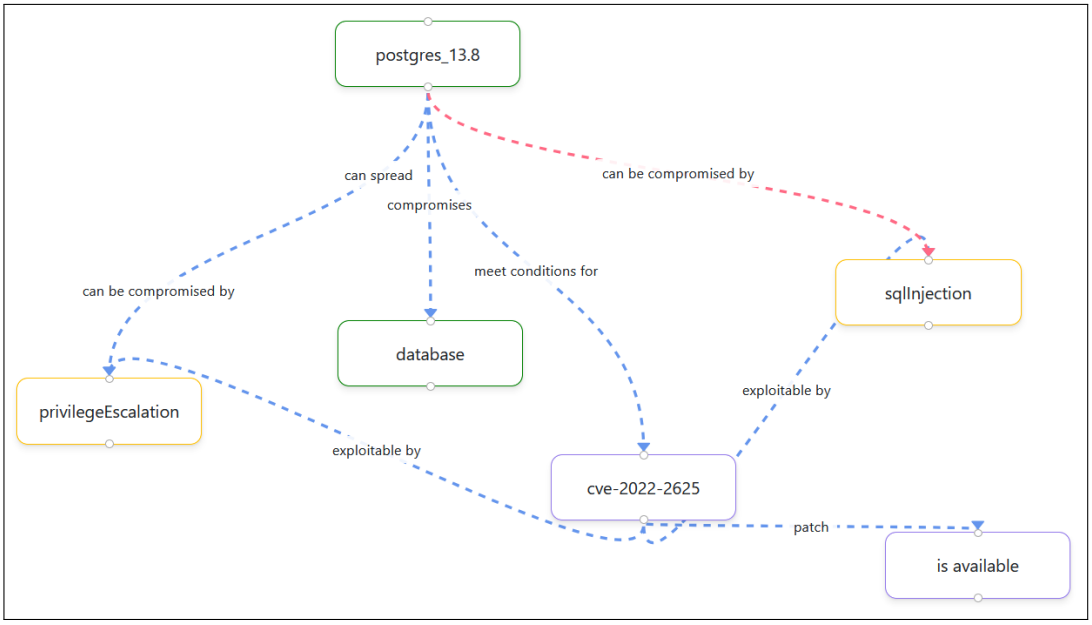

Figure 6.13. canbeComp(database,privilegeEscalation)

6.2.2 Second Use Case

The second use case focuses on an e-commerce system. These kinds of applications are among the most attractive targets for attackers because they process sensitive data and must guarantee service availability and continuity.
The system considered in this use case is composed of several interconnected software components, with their own dependencies. In particular, the architecture includes:

- an **application** built with `NextJs 13.4.4`,`NodeJs 18.16.0` and `pg 3.10.1`.
- a **gateway** based on `Nginx 1.24.0`.
- an **authentication service**, implemented with `Spring 5.3.16` and `Tomcat 9.0.56`, responsible for identity management and access control.
- a **payment service**, depending on the external provider `Stripe 18.5.0`.
- a **database** `PostgreSQL 16.0`, supporting data persistence and transaction management.

- **protection and monitoring mechanisms**, such as the `WAF ModSecurity 3.0.12` and the intrusion detection system `Suricata 7.0.5` used to mitigate or detect known threats.

Figure 6.14 presents a diagram of the architecture, showing the main components and their interconnections.

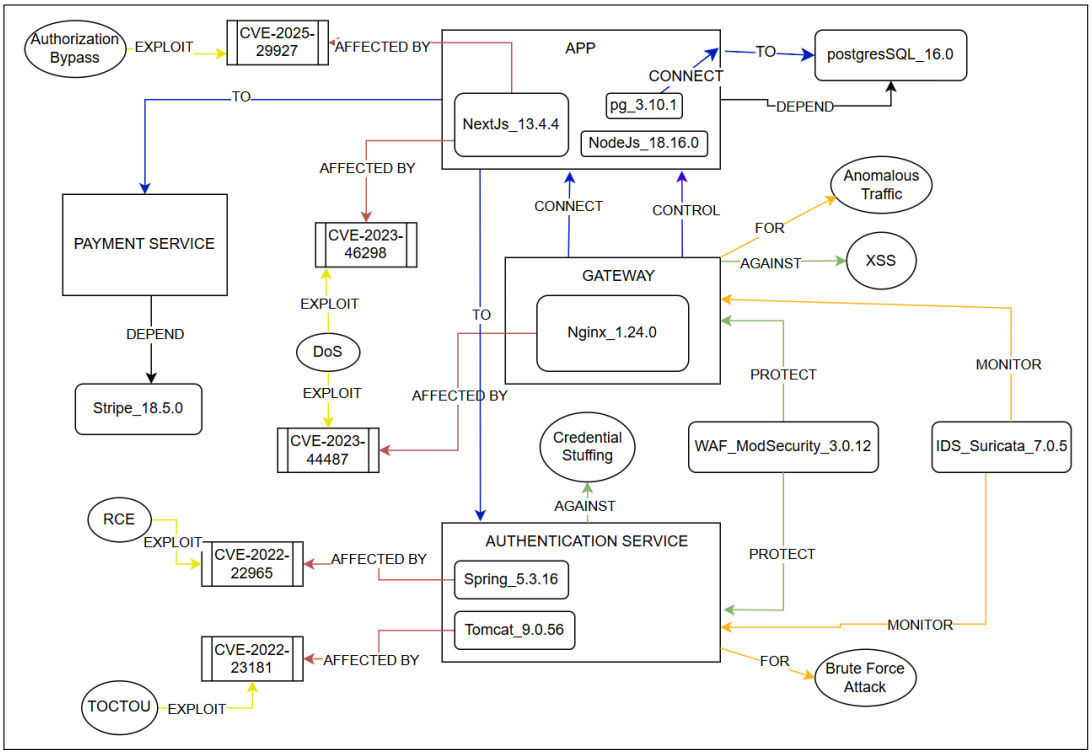

Figure 6.14. Second use case schema

The architecture is then formalized using Prolog, and the following extract represents the core part of the file:

```
contain("app","nextJs_13.4.4").
contain("app","nodeJs_18.16.0").
contain("app","pg_3.10.1").
contain("gateway","nginx_1.24.0").
contain("authenticationService","spring_5.3.16").
contain("authenticationService","tomcat_9.0.56").

connect("gateway","app","authenticationService").
connect("gateway","app","paymentService").
connect("pg_3.10.1","app","postgresSQL_16.0").

control("gateway","app").
```

```
depend("paymentService","stripe_18.5.0").
depend("app","postgreSQL_16.0").

monitor("IDSsuricata_7.0.5","gateway","AnomalousTraffic").
monitor("IDSsuricata_7.0.5","authenticationService",
"BruteForceAttack").
protect("WAFModSecurity_3.0.12","gateway","xss").
protect("WAFModSecurity_3.0.12","authenticationService",
"credentialStuffing").

spread("gateway","xss").
spread("gateway","denialOfService").
spread("nginx_1.24.0","xss").
spread("nginx_1.24.0","denialOfService").
spread("nginx_1.24.0","AnomalousTraffic").
spread("nextJs_13.4.4","denialOfService").
spread("app","denialOfService").
spread("spring_5.3.16","remoteCodeExecution").
spread("tomcat_9.0.56","toctou").
spread("tomcat_9.0.56","credentialStuffing").
spread("tomcat_9.0.56","BruteForceAttack").
spread("nextJs_13.4.4","AuthorizationBypass").
spread("authenticationService","AuthorizationBypass").
spread("app","sqlInjection").

affectedBy("nginx_1.24.0","cve-2023-44487").
affectedBy("nextJs_13.4.4","cve-2023-46298").
affectedBy("nextJs_13.4.4","cve-2025-29927").
affectedBy("spring_5.3.16","cve-2022-22965").
affectedBy("tomcat_9.0.56","cve-2022-23181").
exploit("cve-2023-44487","denialOfService").
exploit("cve-2023-46298","denialOfService").
exploit("cve-2025-29927","AuthorizationBypass").
exploit("cve-2022-22965","remoteCodeExecution").
exploit("cve-2022-23181","toctou").
```

Listing 6.20. Prolog useCase2

Validation

To validate the model, we first execute the query ***canbeDet(gateway, AnomalousTraffic)***. The expected result is that the threat can be detected, since the IDS monitors the gateway for anomalous traffic.
The prolog output is reported below, while the corresponding graph is shown in Figure 6.15:

```
-> (nginx_1.24.0) can spread (AnomalousTraffic)
```

```
-> (nginx_1.24.0) meet conditions for (cve-2023-44487)
   exploitable by (denialOfService)
 -> (cve-2023-44487) patch (is available)
-> (nginx_1.24.0) can be compromised by (denialOfService)
-> (nginx_1.24.0) compromises (gateway) by (AnomalousTraffic)
 -> (IDSsuricata_7.0.5) monitor (gateway) for threat
   (AnomalousTraffic)
-> (gateway) can be detected if compromised by threat
   (AnomalousTraffic)
true.
```

Listing 6.21. Prolog canbeDet(gateway, AnomalousTraffic)

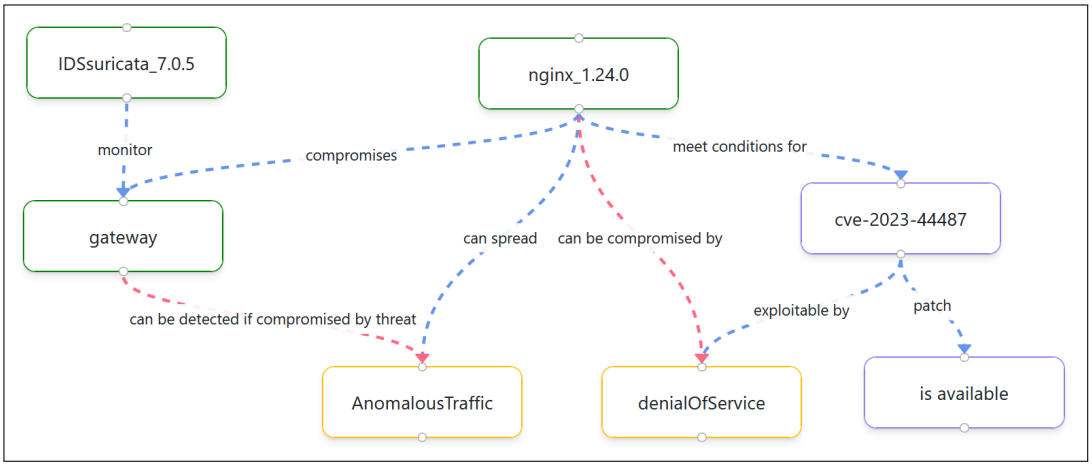

Figure 6.15. canbeDet(gateway,AnomalousTraffic)

To reduce complexity and avoid irrelevant paths,thus simplifying the analysis, we make now the following assumptions:

- `nextJs` cannot spread `AuthorizationBypass`,
- `nginx` cannot spread `AnomalousTraffic`,
- the `gateway` (and its components) cannot spread `xss`.

Under these assumptions, we run the query ***canbeComp(postgresSQL_16.0, sqlInjection)***.
The relevant part of the Prolog output is the following (Figure 6.16):

```
-> (app) can spread (sqlInjection)
-> (nextJs_13.4.4) can spread (denialOfService)
-> (nextJs_13.4.4) meet conditions for (cve-2023-46298)
   exploitable by (denialOfService)
 -> (cve-2023-46298) patch (is NOT available)
-> (nextJs_13.4.4) can be compromised by (denialOfService)
-> (nextJs_13.4.4) compromises (app) by (denialOfService)
-> (postgresSQL_16.0) can be compromises through (app) by
   (sqlInjection)
```

```
true ;

-> (nextJs_13.4.4) meet conditions for (cve-2025-29927)
   exploitable by (AuthorizationBypass)
 -> (cve-2025-29927) patch (is NOT available)
-> (nextJs_13.4.4) can be compromised by (AuthorizationBypass)
-> (nextJs_13.4.4) compromises (app) by (denialOfService)
-> (postgresSQL_16.0) can be compromises through (app) by
   (sqlInjection)
true ;

-> (gateway) can spread (denialOfService)
-> (nginx_1.24.0) can spread (denialOfService)
-> (nginx_1.24.0) meet conditions for (cve-2023-44487)
   exploitable by (denialOfService)
 -> (cve-2023-44487) patch (is available)
-> (nginx_1.24.0) can be compromised by (denialOfService)
-> (nginx_1.24.0) compromises (gateway) by (denialOfService)
-> (app) can be compromises through (gateway) by
   (denialOfService)
-> (postgresSQL_16.0) can be compromises through (app) by
   (sqlInjection)
true ;
```

Listing 6.22. Prolog canbeComp(postgresSQL,sqlInjection)

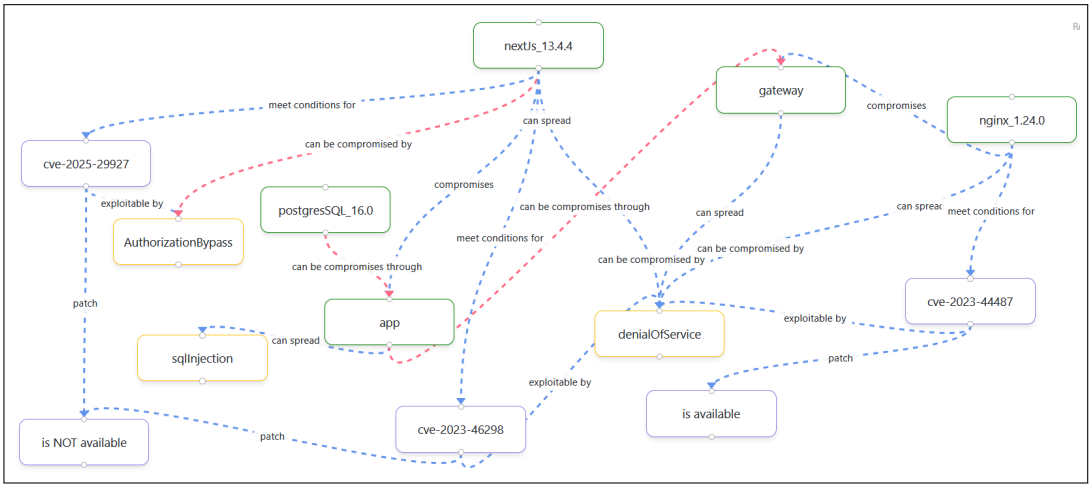

Figure 6.16. canbeComp(postgresSQL, sqlInjection)

The third query executed is ***canbeComp(authenticationService,toctou)***. The Prolog output is reported below (see Figure 6.17 for the corresponding graph):

```
-> (tomcat_9.0.56) can spread (toctou)
-> (tomcat_9.0.56) meet conditions for (cve-2022-23181)
   exploitable by (toctou)
```

```
 -> (cve-2022-23181) patch (is available)
-> (tomcat_9.0.56) can be compromised by (toctou)
-> (tomcat_9.0.56) compromises (authenticationService) by
   (toctou)
true ;
```

Listing 6.23. Prolog canbeComp(authenticationService,toctou)

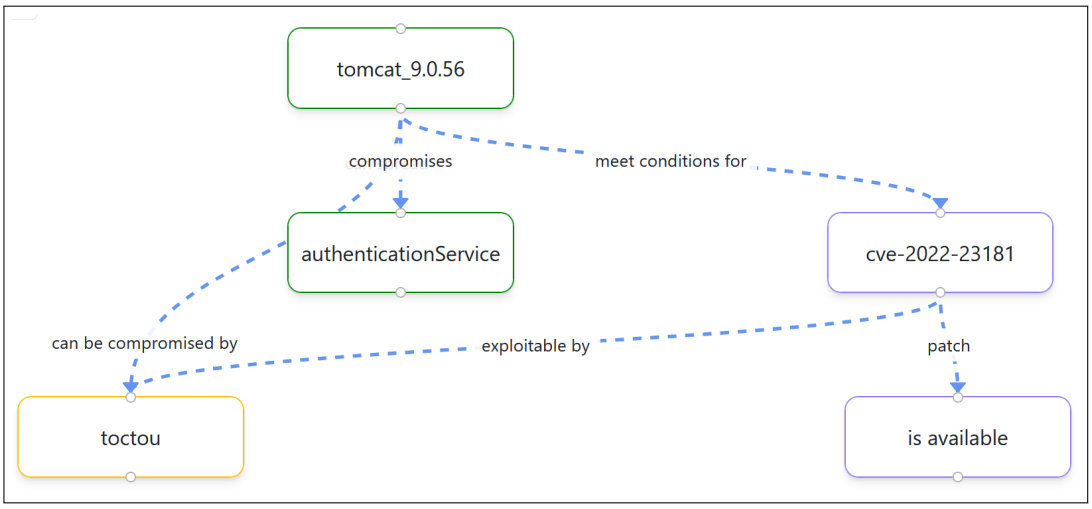

Figure 6.17. canbeComp(authenticationService,toctou)

Finally, the WAF is tested by running the query ***canbeComp(authenticationService, credentialStuffing)***. As expected, the attack is blocked (Figure 6.18):

```
-> (tomcat_9.0.56) can spread (credentialStuffing)
 -> (WAFModSecurity_3.0.12) protect (authenticationService)
   against threat (credentialStuffing)
false.
```

Listing 6.24. Prolog canbeComp(authenticationService,credentialStuffing)

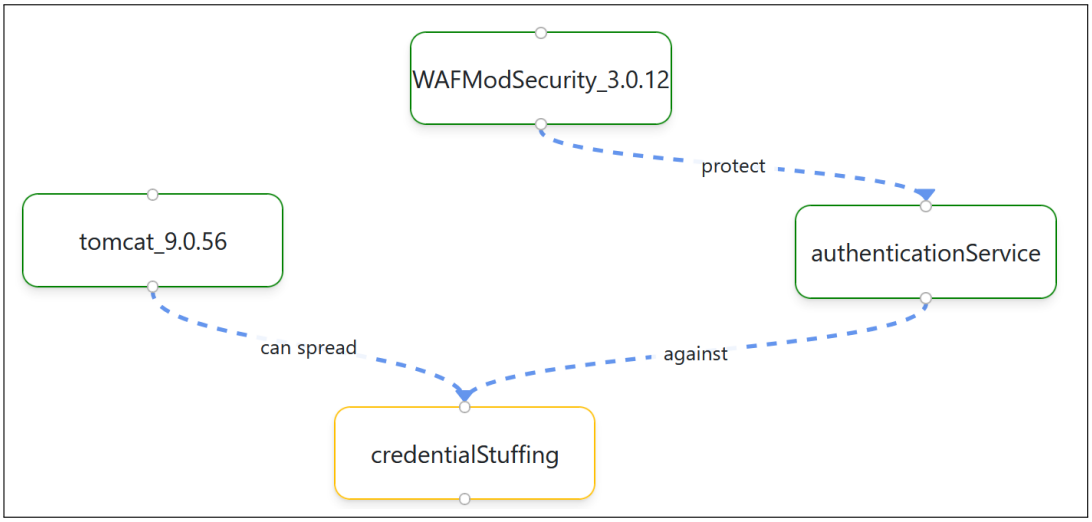

Figure 6.18. canbeComp(authenticationService,credentialStuffing)

Chapter 7

Conclusions

In this thesis, several improvements have been introduced to the threat analysis framework TAMELESS, with the goal of making it more effective and versatile.The work started with the optimization of the framework for software environments and was then extended with the inclusion of CVE and a patch management mechanism. Thanks to these developments, it is now possible to describe a system, map the CVEs associated with it and perform threat analysis that not only identifies vulnerabilities but also studies the propagation of threats across the entire system.This is particularly useful in the context of digital supply chains, where third-party products and software (the weakest links) often represent potential entry points for attackers. The ability to detect vulnerabilities in advance helps prevent consequences that could have a significant impact. Another important improvement concerns the GUI of the framework. Not only the new version of the model has been integrated, but several features, including navigation, have been enhanced. The application is now easier and more intuitive to use. The graph creation and visualization system have also been redesigned: instead of producing multiple and often duplicated graphs, the framework now generates a single graph for each executed query. This provides an immediate, complete, and consistent representation, free of redundant edges, making the results easier to interpret. To evaluate the effectiveness of the new features, the framework has been tested on two use cases. The results and graphical representations obtained through the GUI have been reported, showing how TAMELESS can highlight the propagation of threats within a system, indicating which components compromise others, which CVEs are exploited, and whether patches have been released for them.

7.1 Limitations and Future Works

Despite the improvements, the model still has some limitations. In particular, the use of the framework requires a manual description of the analyzed system, including software components, possible threats, and known vulnerabilities. This process may lead to errors, especially in complex and dynamic systems. Moreover, the integration of CVEs is not yet automated, as a continuous update mechanism

is still missing.
Future works could focus on:

- introducing automation mechanisms for system description, also with the help of AI.
- integrating automatic updates of CVEs, so that the framework always stays aligned with the most recent known vulnerabilities.
- exploring predictive models, able to estimate the probability that new vulnerabilities could be exploited.
- improving the scalability of the framework by optimizing graph management. While a single graph provides a complete view, it may become difficult to interpret in very large environments. Therefore, an alternative approach could be the use of multiple graphs, where each graph represents a result (instead of just a part of a result, as in the original model).
- further improving the GUI, not only by showing the graph but also by including the Prolog output and making the application more interactive.

In conclusion, the improvements introduced in this thesis have made TAMELESS a more complete and practical tool for threat analysis in complex environments such as digital supply chains, providing a solid foundation for the development of additional future features.

Bibliography

[1] C. Klein, "Stride threat model: A complete guide," https://www.jit.io/resources/app-security/stride-threat-model-a-complete-guide, published: 28/04/2025, Accessed: 11/09/2025.

[2] M. Thevarmannil, "10 types of threat modeling methodology to use in 2025," https://www.practical-devsecops.com/types-of-threat-modeling-methodology/?srsltid=AfmBOoos0vQU9x5KSVBkcNlSSDeGNxFE658nu9wnvFOe9HDK0Uim24_W, published: 06/02/2023, Accessed: 11/09/2025.

[3] E. U. A. for Cybersecurity, I. Lella, M. Theocharidou, E. Tsekmezoglou, and A. Malatras, *ENISA threat landscape 2021 – April 2020 to mid-July 2021*, I. Lella, M. Theocharidou, E. Tsekmezoglou, and A. Malatras, Eds., 2021.

[4] Ivanti, "2025 state of cybersecurity report: Paradigm shift," https://www.ivanti.com/resources/research-reports/state-of-cybersecurity-report, published: 18/03/2025, Accessed: 11/09/2025.

[5] L. Conklin, "Threat modeling process," https://owasp.org/www-community/Threat_Modeling_Process, accessed: 11/09/2025.

[6] A. Jenkins, "Digital supply chain explained," https://www.netsuite.com/portal/resource/articles/erp/digital-supply-chain.shtml, 2022, published: 24/08/2022, Accessed: 11/09/2025.

[7] V. Drake, "Threat modeling," https://owasp.org/www-community/Threat_Modeling, accessed: 11/09/2025.

[8] "What is a cve?" https://www.redhat.com/en/topics/security/what-is-cve#overview, published: 04/09/2024, Accessed: 11/09/2025.

[9] M. Thevarmannil, "What is the stride threat model? beginner's guide – 2025," https://www.practical-devsecops.com/what-is-stride-threat-model/, published: 07/12/2022, Accessed: 11/09/2025.

[10] F. Valenza, E. Karafili, R. V. Steiner, and E. C. Lupu, "A hybrid threat model for smart systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 5, pp. 4403–4417, 2023.

[11] C. Gonzalez, "Threat modeling: 5 steps, 7 techniques, and tips for success," https://www.exabeam.com/blog/infosec-trends/top-8-threat-modeling-methodologies-and-techniques/, published: 06/06/2023, Accessed: 11/09/2025.

[12] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[13] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–37, 2023.

[14] D. Bringhenti, L. Seno, and F. Valenza, "An optimized approach for assisted firewall anomaly resolution," *IEEE Access*, vol. 11, pp. 119 693–119 710, 2023.

[15] "Why traditional threat modeling fails and how to get it right," https://www.securitycompass.com/whitepapers/why-traditional-threat-modeling-fails-and-how-to-get-it-right/, accessed: 11/09/2025.

[16] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, 2022.

[17] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "A two-fold traffic flow model for network security management," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3740–3758, 2024.

[18] D. Bringhenti and F. Valenza, "Greenshield: Optimizing firewall configuration for sustainable networks," *IEEE Transactions on Network and Service Management*, 2024.

[19] G. Francia, L. Ertaul, L. H. Encinas, and E. El-Sheikh, *Computer and network security essentials*. Springer, 2017.

[20] B. Hammi, S. Zeadally, and J. Nebhen, "Security threats, countermeasures, and challenges of digital supply chains," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–40, 2023.

[21] A. R. Nygård, A. Sharma, and S. Katsikas, "Reverse engineering for thwarting digital supply chain attacks in critical infrastructures: Ethical considerations," 2022.

[22] S. Boyson, T. M. Corsi, and J.-P. Paraskevas, "Defending digital supply chains: Evidence from a decade-long research program," *Technovation*, vol. 118, p. 102380, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166497221001619

[23] "Solar winds cyber attack," https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack, accessed: 11/09/2025.

[24] P. Robinson, "The moveit attack explained," https://www.lepide.com/blog/the-moveit-attack-explained/, published: 07/01/2025, Accessed: 11/09/2025.

[25] "Cve json record format," https://cveproject.github.io/cve-schema/schema/docs/#collapseDescription_root, accessed: 11/09/2025.

[26] "Full-record basic example (cve schema)," https://github.com/cveproject/cve-schema/blob/main/schema/docs/full-record-basic-example.json, accessed: 11/09/2025.