UCLouvain

epl

École polytechnique de Louvain

Building Cybersecurity Scenarios: Forging a Methodology Through Iterative Creation and Implementation

Author: **Noah Fraiture**
Supervisors: **Ramin Sadre, Benoît Duhoux**
Reader: **Lionel Metongnon**
Academic year 2024–2025
Master [120] in Computer Science

Building Cybersecurity Scenarios: Forging a Methodology Through Iterative Creation and Implementation

Noah Fraiture
UCLouvain, Louvain-La-Neuve, Belgique

***Abstract*—Existing cybersecurity literature on scenario development within cyberranges often prioritizes commercial interests, focusing on promoting cyberrange platforms rather than systematic scenario design methods. This master's thesis addresses this gap by proposing the "Helical Build Framework", an agile methodology for designing cybersecurity scenarios that integrates technical design with narrative coating. The methodology was developed and refined through the iterative creation of four distinct scenarios: Satellite Siege, Hotel Daemon, Patch War and Zheng Hijack. Each scenario was fully developed from design, using MITRE ATT&CK tactics, techniques and procedures (TTPs) for the attack-defense flow, to implementation with NixOS, a declarative Linux distribution. This approach integrates technical design with narrative elements to enhance training realism.**

***Index Terms*—Cybersecurity, Scenario, Methodology, NixOS**

Acknowledgements

I express my deepest gratitude to my supervisors, Professor Ramin Sadre and Benoît Duhoux, for their invaluable guidance throughout this master's thesis. I am particularly grateful to Benoît for dedicating significant time to mentoring me, especially during the writing process. Your commitment to excellence was evident and greatly appreciated.

I also extend my thanks to Zlatan Kovacevic, who provided invaluable assistance with CITEF and its challenging documentation.

I am grateful to the directors of engineering at xAI for developing Grok, which significantly aided in crafting a grammatically correct and engaging master's thesis.

My appreciation goes to the NixOS community for creating the remarkable NixOS Linux distribution, which I use daily and which proved to be a significant asset during the development of this thesis. Special thanks to Will Gordon, u/nixgang, and the members of the r/NixOS Reddit community for their support and feedback on my work.

Finally, I heartfeltly thank my loved ones for their unwavering support throughout my academic journey.

Table of Contents

1. Introduction

1.1. Context

The growing number of cybersecurity incidents highlights the need for skilled cybersecurity professionals. Cybersecurity is a complex area of computer science that requires deep knowledge and hands-on experience. Recent studies show a sharp rise in the frequency and impact of cyber incidents, resulting in global economic losses of hundreds of billions annually[1], [2]. For instance, the average cost of a data breach often reaches millions per incident, and indirect costs, like damage to reputation and disruptions to operations, are often overlooked[3].

1.2. Problem

Cybersecurity incidents have varied causes, requiring different strategies to address them. A key step is training cybersecurity experts who can take charge and improve security practices. Unlike software development, real-world cybersecurity training is limited by legal and ethical issues. This makes safe, simulated environments vital for building skilled professionals. These environments and scenarios are essential for effective practice, but their development needs a clear methodology.

A review of existing research reveals limited work on this topic. For example, the white paper[1] *"White Paper: Cyber Exercise Scenario Development"*[2] suggests a methodology for creating scenarios but focuses on high-level, theoretical exercises, such as tabletop activities, and emphasizes organizational factors like participant roles in a company. By contrast, the methodology in this thesis targets practical, technical scenarios that are modular and usable by individuals, independent of specific organizational settings. While the paper covers broader strategic exercises, the approach in this thesis supports detailed, technical scenario design and implementation, meeting a distinct need in cybersecurity training.

1.3. Motivation

One effective way to train cybersecurity professionals is to provide virtualized environments for learning and practice. Several online platforms, such as TryHackMe[2], Root-Me[3], and HackTheBox[4], offer such environments. However, these platforms have clear limitations:

[1] A white paper is a report or a guide that present an issue with the solution of the author[4]
[2] https://tryhackme.com
[3] https://www.root-me.org
[4] https://www.hackthebox.com

- **Proprietary nature**: Their use may be restricted and tied to third-party providers, reducing flexibility in the usage of the scenarios, without the possibility to edit them.
- **Cost**: Access to detailed exercises often requires payment, creating financial challenges for organizations over time. It should be noted that maintaining a dedicated infrastructure also incurs costs that are not necessarily lower.
- **Limited exercise scope**: The number of available exercises is limited. Creating scenarios for every plausible situation is difficult and requires significant effort. Moreover, realistic and complex scenarios often need advanced infrastructure, such as multiple machines and complex network setups, which are expensive to build and maintain. Online platforms may not prioritize offering such scenarios.
- **Limited control and feedback**: Without control over exercise design, users cannot access detailed scenario data. This makes it hard to provide specific feedback, adjust exercises to meet particular needs, or identify and address trainee weaknesses, which lowers training effectiveness.

These limitations highlight the need for a clear, platform-independent approach to designing cybersecurity scenarios. Unlike proprietary systems, this approach allows trainers to create scenarios freely, with full control to customize content. This thesis explores the use of a cyberrange, a virtual environment simulating realistic infrastructure, to support these scenarios. By avoiding reliance on commercial platforms, costs can be reduced, although the chosen infrastructure may still involve some expenses. The modular design enables the efficient creation of diverse and complex scenarios with reusable components, overcoming the restrictions of fixed exercise libraries. Additionally, the **Helical Build Framework** supports detailed scenario creation, providing precise feedback and allowing adjustments to meet specific training objectives, thus enhancing control and effectiveness.

1.4. Contributions

This thesis provides three key contributions to the development of cybersecurity training scenarios.

First, this master's thesis introduces the **Helical Build Framework**, a new methodology for designing cybersecurity scenarios that are modular, reliable, and repeatable. The methodology tackles the lack of clear scenario design methods in current research by using an agile and modular approach. It supports the creation of scenarios with reusable components, which can be applied to narrative elements and attack-defense sequences. These components build a library that helps develop future scenarios, ensuring flexibility and consistency. The methodology was improved through the creation of four distinct scenarios, showing its ability to produce effective training environments that simulate real-world cyber threats for educational purposes.

The **Helical Build Framework** is named for its iterative, spiral-like process, similar to a helix that grows upward while revisiting its core structure through repeated cycles. Each cycle enhances and expands scenario components, starting from a single idea that forms the scenario's foundation. Gradually, components are added and improved to create a complete scenario with a clear start and end, ensuring a structured yet flexible design process.

The methodology separates scenario design, which includes creating narratives, attack-defense flow, and configurations, from implementation, which focuses on deploying scenarios using specific technologies. This separation increases flexibility, allowing the same components to be used in different scenarios, whether they involve a new storyline, attack-defense flow, or configuration, to meet various training needs. For instance, the first implemented scenario *Satellite Siege*, which focuses on an attack on a satellite ground station, includes a DNS cache poisoning exploit in its attack-defense flow. This exploit can be reused in a different storyline, such as disrupting a botnet. The configuration, representing a typical company network, can be easily reused for both attack and defense in various scenarios. Additionally, the same storyline could be reused for a completely different type of attack with minor changes, depending on its level of specificity.

Second, this master's thesis presents a practical implementation process for the **Helical Build Framework**, using **NixOS**[5], a declarative Linux distribution, to ensure modularity, reproducibility, and scalability in deploying cybersecurity scenarios. **NixOS** enables consistent and flexible configurations, aligning with the methodology's iterative principles and supporting reusable components through its declarative approach. The implementation involves selecting virtualization methods, such as virtual machines and containers, to match the infrastructure, while choosing **NixOS** as the base system for both. This choice allows quick, consistent changes and efficient deployment of complex configurations, improving the accessibility and scalability of training environments.

And third, as mentioned before, this master's thesis presents four scenarios developed to create and refine the methodology and implementation process. These scenarios serve as practical tests to improve and confirm the methodology's versatility and ensure the methodology's ability to create effective scenarios. The scenarios, available on the GitHub repository of this thesis[6], are:

- *Satellite Siege*: Hack a spy satellite ground base to intercept communication. The trainee must enter the internal network of a company and exploit network vulnerability to intercept internal packet.
- *Hotel Daemon*: Use IoT devices to take control of a hotel and trap a criminal. The trainee must control several IoT devices to find the criminal through cameras and lock them in their room.

[5]https://nixos.org/

[6]https://github.com/noahfraiture/thesis

- *Patch War*: Hack a game to patch an exploit. The trainee must exploit a vulnerability to access the game server and fix a code flaw that allows players to cheat.
- *Zheng Hijack*: Take down a botnet. The trainee must use forensic and reverse engineering skills to detect and dismantle a botnet deployed in a company's network.

1.5. Roadmap

This thesis starts by explaining the defining key concepts and exploring the literature in chapter *Background*, such as "scenario configuration", "attack-defense flow", and "story line". These terms describe the components of a cybersecurity scenario in detail. The next section reviews the current state of research on cybersecurity scenario development. Early research showed that literature on cyberranges and methods for building cybersecurity scenarios is limited, and this section will examine this gap.

The following chapter *Problem Statement* defines the problem and highlights gaps in the existing literature. It explains how this thesis aims to address these gaps through three research questions:

- *How can we create a strong cybersecurity scenario to simulate real-world cyber threats?*
- *How can we design and implement a cybersecurity scenario in a modular and reusable way?*
- *How can we evaluate the proposed methodology?*

The core of the thesis presents a theoretical approach to building scenarios. First, the *Methodology* chapter introduces the **Helical Build Framework**. It provides an overview of the methodology and describes its process step by step, using a practical example, to create a scenario. This process focuses on two main questions revisited multiple times:

- *What steps allow the trainee to progress to this stage of the scenario?*
- *What goal does this stage achieve in the scenario's design?*

Next, the *Implementation* chapter discusses technical foundation. It begins with a section on virtualization, exploring different options along with their advantages and disadvantages. It then explains the choice of NixOS as the base system for the developed scenarios.

The following chapter *Training Scenarios* describes the implemented scenarios: Patch War, Satellite Siege, Zheng Hijack, and Hotel Daemon. These scenarios are a key part of the thesis, as they are fully developed, tested, and used to refine the methodology and support the discussion of implementation.

The *Evaluation* chapter assesses the quality of the methodology to ensure its effectiveness but also discuss the proposed implementations and the importance of

narrative in scenarios. The methodology is discussed with objective quality such as the soundness and completeness, and via self validation through the four scenarios built in this thesis.

Finally, the thesis concludes with a *Discussion* chapter, which examines the study's limitations and proposes future research directions to enhance the methodology and technologies used, followed by the *Conclusion* chapter.

2. Background

The Background chapter offers essential context for understanding the methodology developed in thesis. It includes a lexicon of key concepts and a review of existing methodologies.

2.1. Lexicon

In this thesis, we use the terms “scenario configuration”, “attack-defense flow”, and “story line” frequently. It is important to define these terms clearly.

a) *Scenario Configuration*

The scenario configuration is the technical setup that supports a cybersecurity scenario. It provides the environment needed for the story line and attack-defense components. This setup includes workstations, servers, and their settings, designed to match the scenario’s goals and create a realistic training environment for the trainee.

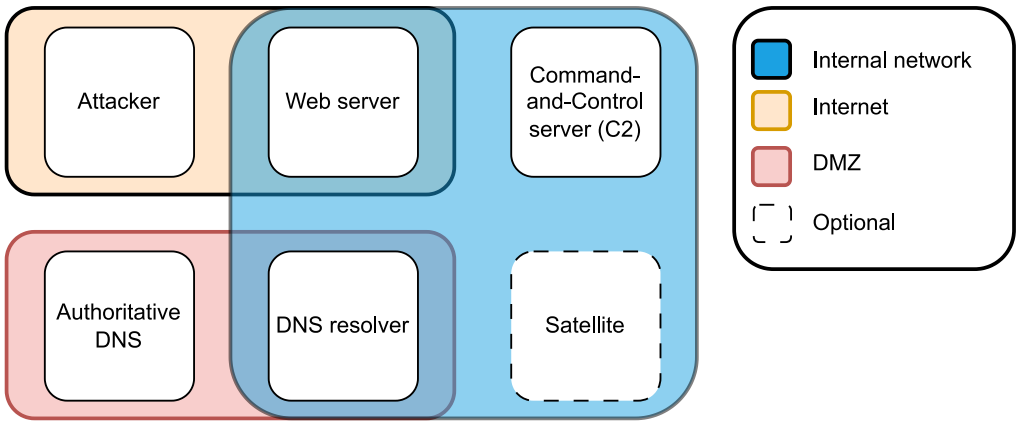

Fig. 1: *Satellite Siege* configuration

The configuration in Fig. 1, from the *Satellite Siege* scenario, supports a cyberattack involving a DNS cache poisoning exploit. The setup includes key components. The trainee’s workstation, acting as the attacker, is a virtual machine running Kali Linux, offering a familiar platform for the attack. A server hosts the target website, serving as the entry point to the satellite control station’s internal network. For the DNS cache poisoning attack, the setup includes an authoritative DNS server and a DNS resolver, allowing the trainee to manipulate network traffic. Another server simulates communication with the spy satellite, enabling message interception. Including this server in the setup, even though not required to emulate the satellite, helps make the scenario clearer.

b) *Attack-Defense Flow*

The attack-defense flow delineates a structured sequence of technical and strategic steps for trainees to follow. It utilizes MITRE ATT&CK Tactics, Techniques, and Procedures (TTPs)[7][8] and Common Vulnerabilities and Exposures (CVEs[9]) to construct realistic attack-defense paths represented as an attack-defense flow. This concept, derived from MITRE ATT&CK[10], encompasses all possible paths to achieve the scenario's objectives, linked to MITRE ATT&CK TTPs. An attack-defense path represents a specific route within the flow, connecting two nodes and integrating both offensive and defensive actions. This ensures that the trainee's actions within the technical environment align with the intended training or evaluation objectives.

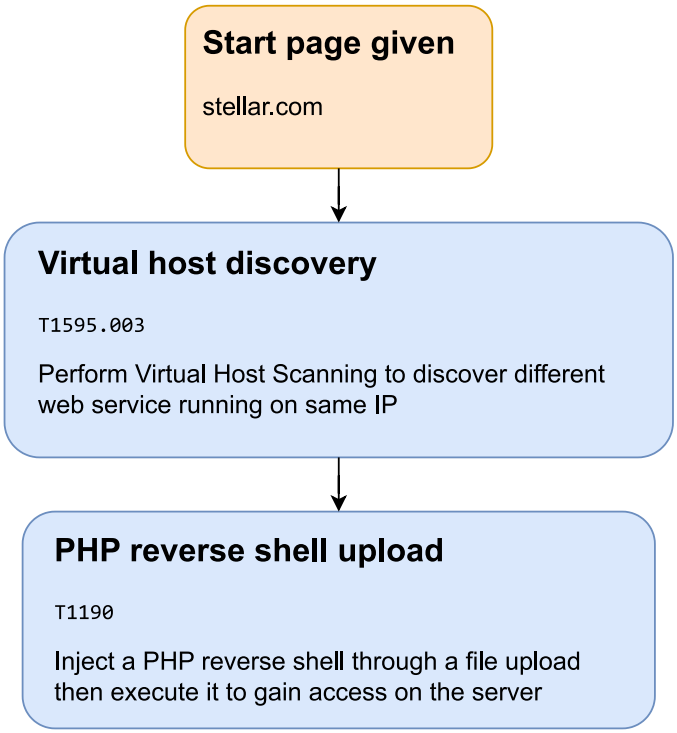

Fig. 2: *Satellite Siege* sample attack flow

The attack flow shown in Fig. 2 illustrates a scenario where a trainee compromises a satellite ground base to intercept satellite communications, which is a sample of the scenario *Satellite Siege*. This flow highlights *actions* (in blue) and *assets* (in orange). An *action* is a step the trainee takes to advance, often mapped to MITRE ATT&CK

[7]https://attack.mitre.org/
[8]https://www.ibm.com/think/topics/mitre-attack
[9]https://www.cve.org/
[10]https://center-for-threat-informed-defense.github.io/attack-flow/

Tactics, Techniques, and Procedures (TTPs). An *asset* is a resource gained or used in an action, essential for subsequent steps. For example, the trainee first accesses the internal network of the satellite ground base via its public website, *stellar.com*. The initial action involves performing a virtual host scan on the *stellar.com* server to identify a poorly protected admin dashboard, hidden by obfuscation techniques. From this dashboard, the trainee exploits a file upload feature to deploy a PHP reverse shell, gaining shell access to the web server and entry to the internal network.

c) *Story Line*

The story line provides the context for a cybersecurity scenario, creating a realistic and engaging backdrop that motivates the trainee. It includes the introduction given to the trainee before starting and contextual elements within the environment during the scenario. The narrative makes the technical and strategic components feel plausible, improving trainee engagement and guiding their decisions.

In the first scenario *Satellite Siege*, the trainee must infiltrate a satellite control station to intercept communications with a spy satellite. The story line presents this as a high-stakes espionage mission, with the trainee acting as a cyberattacker targeting critical infrastructure. A webpage representing the control station's interface serves both practical and contextual purposes. Practically, it provides the trainee with initial access to the target, including the IP address needed for the attack. Contextually, it makes the scenario realistic by showing the control station as a believable target with operational details that fit the espionage theme. A well-designed story line with strong storytelling improves the gamification aspect[5], leading to better training outcomes.

2.2. Existing Methods

Research on methods to create cybersecurity scenarios is scarce, as noted earlier. However, training experts with scenarios is not the only way to improve cybersecurity. Cybersecurity is a concern for everyone. A master's thesis from the University of Applied Sciences Northwestern Switzerland, *A Serious Game for Simulating Cyberattacks to Teach Cybersecurity*[6], aimed to raise awareness about cybersecurity. Instead of using traditional phishing email campaigns to show potential threats, the study presented a full attack to participants through a game, including social engineering, SQL injection, and bad USB. The study found that such exercises, even for non-technical people, help them understand how attacks work and how to prevent them at their level.

A review from Leiden University, *A Systematic Review of Current Cybersecurity Training Methods*[7], examined various training methods across multiple studies. These methods include game-based training (N=38), presentation-based training (N=19), simulation-based training (N=16), information-based training (N=15), video-based

training (N=14), text-based training (N=13), and discussion-based training (N=6). These methods target end-users, especially non-technical people. For example, one game-based method asks users to create a strong password with guidance on password rules. Many game-based and simulation-based trainings require little interaction, which may be enough for non-technical awareness but not for training experts who need experience in a safe environment. This review offers useful insights: most studies report positive outcomes, regardless of the cybersecurity topic or method. This suggests that many cybersecurity topics can benefit from training, supporting the need for a strong methodology to create scenarios that cover diverse topics. The review also notes that many studies use simple methods, highlighting the need for a methodology to develop more complex training scenarios easily.

The ECSO white paper, *Cyber Exercise Scenario Development*[8], provides guidance for creating cybersecurity scenarios, describing four types of exercises: tabletop, walkthrough, full-scale, and cyber drills. Each type has a specific purpose:

- **Tabletop**: A discussion-based exercise where participants review hypothetical cyber incidents and analyze policies, procedures, and resources. This low-cost, low-pressure format supports strategic planning.
- **Walkthrough**: A detailed review of an organization's incident response plan to assess its effectiveness.
- **Full-scale**: A realistic simulation of a cyber incident, where participants respond as they would in a real situation.
- **Cyber drills**: Practical exercises focusing on specific skills, such as security controls or malware analysis, to improve technical skills.

The **Helical Build Framework** proposed in this thesis is most similar to cyber drills, as it focuses on practical, skill-based scenarios. Unlike the ECSO white paper, which emphasizes blue-team (defensive) perspectives, the methodology in this thesis supports both offensive and defensive scenarios, with a stronger focus on offensive ones. The white paper lists five general steps for developing these exercises:

1) Define objectives.
2) Identify participants and stakeholders.
3) Determine the scenario system environment.
4) Design realistic scenarios based on relevant cyber threats.
5) Validate and evaluate the scenario.

The **Helical Build Framework** offers a detailed and systematic approach for steps 3 and 4, focusing on creating modular, reliable, and reproducible scenario environments and realistic designs. By combining technical setup, story elements, and attack-defense components, the methodology in this thesis improves the ECSO's general guidelines with a clear, flexible process. The ECSO white paper's steps and recommendations can work well with the **Helical Build Framework**, as it adds a practical, step-by-step process to its broader guidelines.

3. Problem Statement

The frequent and advanced nature of cybersecurity incidents shows the vital need for trained experts able to tackle diverse threats. As noted in the introduction, many current training platforms lack adaptability, rely on proprietary systems, and fail to provide modular or reusable scenarios that align with specific training needs. Furthermore, existing research reveals a significant gap in standardized methodologies for developing cybersecurity scenarios that are consistent, modular, and suitable for various settings. This chapter investigates this methodological gap and considers the questions essential for resolving it.

3.1. Evidence of the Gap

During the initial research phase of this study, a notable gap was identified in the literature concerning methodologies for designing cybersecurity scenarios. Existing methodologies are often tied to promoting specific cyberranges, driven by commercial interests. As a result, these methodologies are adapted to particular platforms and lack applicability in broader, platform-agnostic contexts. This gap restricts the development of reliable, modular scenarios essential for effective cybersecurity training. This thesis aims to address this need through the proposed **Helical Build Framework**, a flexible and systematic approach to scenario design.

The lack of a standardized methodology creates major challenges for developing cybersecurity scenarios. Without a clear methodology, designing scenarios becomes a time-consuming process that involves several difficulties, such as repetitive rework of components that do not integrate well with others or are incompatible, and difficulty reusing previously created components. Moreover, the quality of scenarios relies heavily on the expertise of their creators, which makes it hard for less experienced developers to produce reliable and realistic scenarios. Additionally, the absence of a shared methodology limits collaboration within the cybersecurity community, as there is no common structure to support the design, sharing, or improvement of scenarios. These challenges highlight the need for a systematic approach that promotes the creation of modular, high-quality scenarios.

3.2. Research Questions and Contributions

This thesis aims to develop a clear methodology for creating cybersecurity scenarios to improve training and address the implementation foundation of such scenarios. The research questions guide this thesis by addressing key challenges in scenario design, implementation, and evaluation.

- *How can we design a robust cybersecurity scenario to simulate real-world cyber threats?*

This question addresses the need for a systematic methodology to design cybersecurity scenarios that effectively simulate real-world threats. The literature lacks standardized approaches for scenario creation, which limits the development of consistent and scalable training environments. This thesis proposes an agile methodology, the **Helical Build Framework**, to create reliable, modular, and reusable scenarios, filling this gap and enhancing cybersecurity training.

- *How do we implement a cybersecurity scenario design in a modular and replicable way?*

This question focuses on the implementation of cybersecurity scenarios, specifically the technologies required to ensure modular and replicable designs. Technology choices are critical for efficient and consistent scenario deployment, yet they remain largely independent of the design methodology. This thesis explores technologies, such as NixOS, that align with the **Helical Build Framework**'s principles, enabling the creation of modular, reliable, and reusable scenarios while allowing flexibility to adapt to different methodologies or tools.

- *How can we evaluate the methodology?*

This question aims to assess the effectiveness of the **Helical Build Framework** in designing cybersecurity scenarios. A robust methodology must demonstrate two key qualities. First, it should be versatile, capable of producing any correct scenario that aligns with diverse training objectives, such as simulating varied real-world threats. Second, every scenario it produces must be correct, meaning it is reliable, functional, and educationally valuable. However, as illustrated in the *Infinite Monkey Theorem*, even random processes could theoretically generate valid scenarios given infinite attempts[9]. Thus, evaluation must confirm that the methodology consistently delivers correct scenarios without relying on chance, ensuring its reliability and distinguishing it from arbitrary approaches.

4. Methodology

The methodology developed in this thesis, termed the **Helical Build Framework**, provides an agile approach to designing cybersecurity scenarios, encompassing the scenario narrative, configuration, and attack-defense components. The term "agile" denotes a development process characterized by iterative cycles, incremental progress, and adaptability, enabling continuous refinement based on feedback and evaluation. The methodology starts with a core concept and expands it through iterative development, ensuring that scenarios are aligned with cybersecurity training and evaluation objectives while remaining flexible and reusable. This methodology is particularly well-suited for designing attack-defense flows, as it is naturally represented as a graph, which reflects the evolving structure of the methodology during its iterative process. This chapter gives us resources to answer the first question we asked ourselves in the problem statement : *How can we design a robust cybersecurity scenario to simulate real-world cyber threats?*.

4.1. Intuition

The **Helical Build Framework** adopts a helical development process, where scenario creation progresses through iterative cycles that revisit and refine previous stages. Unlike a linear approach, this spiral-like process allows each cycle to improve the configuration, attack-defense flow and the scenario's narrative, incorporating insights from earlier iterations. This iterative refinement ensures that scenarios are robust, modular, and aligned with cybersecurity training goals.

The methodology's strength lies in its emphasis on modularity, enabling scenarios to be built from interconnected yet independent components. For instance, a narrative depicting an espionage operation or a configuration involving a DNS server can be reused or adapted across different scenarios. This modularity creates a growing library of reusable components, such as attack sequences or network configurations, which supports rapid development of new scenarios. It also allows alternative pathways within scenarios, enhancing their versatility. For example, the attack-defense component of the *Satellite Siege* scenario, which involves a DNS cache poisoning exploit, can be modified to include other vulnerabilities, enriching the trainee's experience without requiring a complete redesign.

During the research phase, no structured approach was found for scenario design. Compared to a naive method where the trainer builds the scenario gradually through exploration, the **Helical Build Framework** offers clear advantages. Linear development methods, which follow a rigid sequence, struggle with revisions, as changes to foundational elements can disrupt the entire scenario. Parallel development approaches, where multiple components are built simultaneously and later merged, risk integration

challenges, especially if one component faces implementation issues. For example, in the *Hotel Daemon* scenario, developing multiple IoT attack steps in parallel led to difficulties when merging them due to incompatible software implementations. This showed the challenge of adapting to unforeseen issues in parallel methods, as revising one step required adjustments across others. By contrast, the helical methodology's iterative and modular nature allows continuous refinement without extensive redesign. By building and testing components incrementally, it minimizes the risk of large-scale revisions, making it more effective for creating scalable and engaging cybersecurity training environments.

Developed through the iterative creation of four cybersecurity scenarios, the **Helical Build Framework** streamlines scenario design by balancing flexibility and efficiency. The library of reusable components grows with each scenario, enabling faster and more effective development. This approach applies to designing configuration, attack-defense flow and the narrative, ensuring each benefits from the methodology's iterative principles. Additionally, the methodology offers flexibility in implementation timing. Developers can choose to implement scenario systems at the end of the design process or incrementally after each cycle. This adaptability supports rapid prototyping and testing, aligning with the methodology's goal of creating scalable, engaging, and educationally valuable cybersecurity training environments.

4.2. Process

The development process starts by choosing a single core idea, which forms the foundation of the scenario. This core idea, called the core node, does not need to be a critical element but serves as the basis for building the configuration, attack-defense flow and narrative. The scenario is designed as a directed graph, where nodes represent steps in the design. An incoming link shows that a node enables the next step, while an outgoing link indicates that a node leads to further steps.

As an example, consider the attack-defense flow of the first scenario *Satellite Siege*, where the trainee must hack a satellite ground base. The core node was a DNS cache poisoning attack. This exploit allows the interception of packets sent to another machine by manipulating domain name resolution. This core node is sufficient to begin building the scenario.

To guide the iterative and modular development of the scenario within the **Helical Build Framework**, two questions are used to shape the trainee's progress, set objectives, and maintain flexibility:

- *What steps allow the trainee to reach this stage of the scenario?*
- *What goal does this stage achieve in the scenario's design?*

These questions support the step-by-step creation of the scenario by adding new elements, such as narrative details, technical setups, or attack steps, which are improved through repeated cycles. They focus on specific parts of the scenario without needing a complete design, enabling flexible growth and adaptation across the configuration, attack-defense flow, and narrative. Each answer to a question creates a new step, represented as a graph node. Two special nodes exist: the starting node and the end node. The process stops when three conditions are met:

- A starting node exists. This node has no exploit and provides the context for the trainee's first action, often a workstation with a Linux distribution like Kali Linux.
- An end node exists. This node represents the trainee's goal, such as obtaining a document or accessing a specific machine.
- A path connects the starting node to the end node, ensuring the goal is reachable.

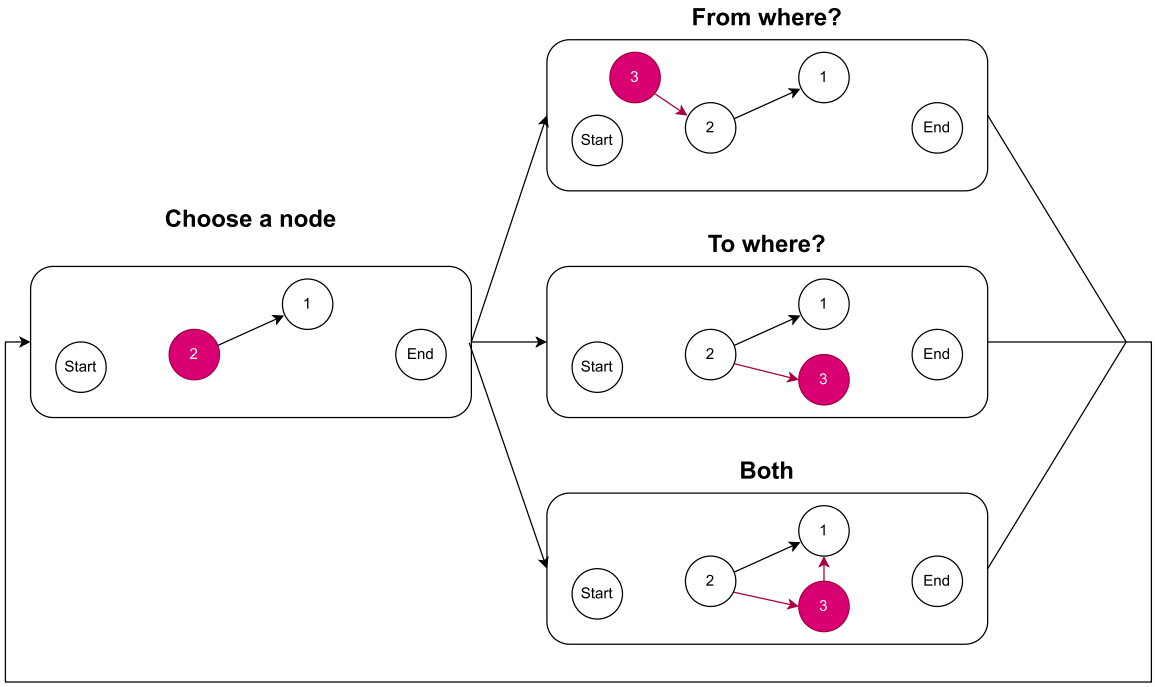

Fig. 3: Steps to build a cybersecurity scenario following the **Helical Build Framework**

To move forward in the creation process, as shown in figure Fig. 3, a node is selected, and one or both guiding questions are applied. The selected node can be any step, including those with existing incoming or outgoing links. This flexibility supports the creation of parallel paths, which offer points of variation in the scenario by allowing alternative routes (where completing one path is sufficient) or requiring both paths to be completed (where both are necessary to achieve the goal). The questions are revisited multiple times during design, with each answer adding a new node to the scenario. Since both questions can have multiple answers, a new node may connect to several other steps, creating a network of linked actions. Although nodes are connected,

they remain independent enough to allow different outcomes while keeping the same core structure. However, changing the core node can be difficult and may cause issues, as seen during scenario development. Thus, it is best to ensure each component is reliable before moving to the next, reducing the need for major changes later.

After the process is complete, a verification step can be done. Since the scenario is a directed graph, simple checks can be performed. The basic requirement is that at least one path exists from the starting node to the end node. Graph exploration algorithms can confirm this. Additionally, checking node reachability is useful. A node that cannot be reached or does not help reach the end node has no purpose and should be removed or revised.

4.3. Running example

This section illustrates the application of the **Helical Build Framework** using the *Satellite Siege* scenario. This scenario serves as a practical example of how the methodology's iterative and modular principles guide the development of a cybersecurity scenario's narrative, configuration, and attack-defense components.

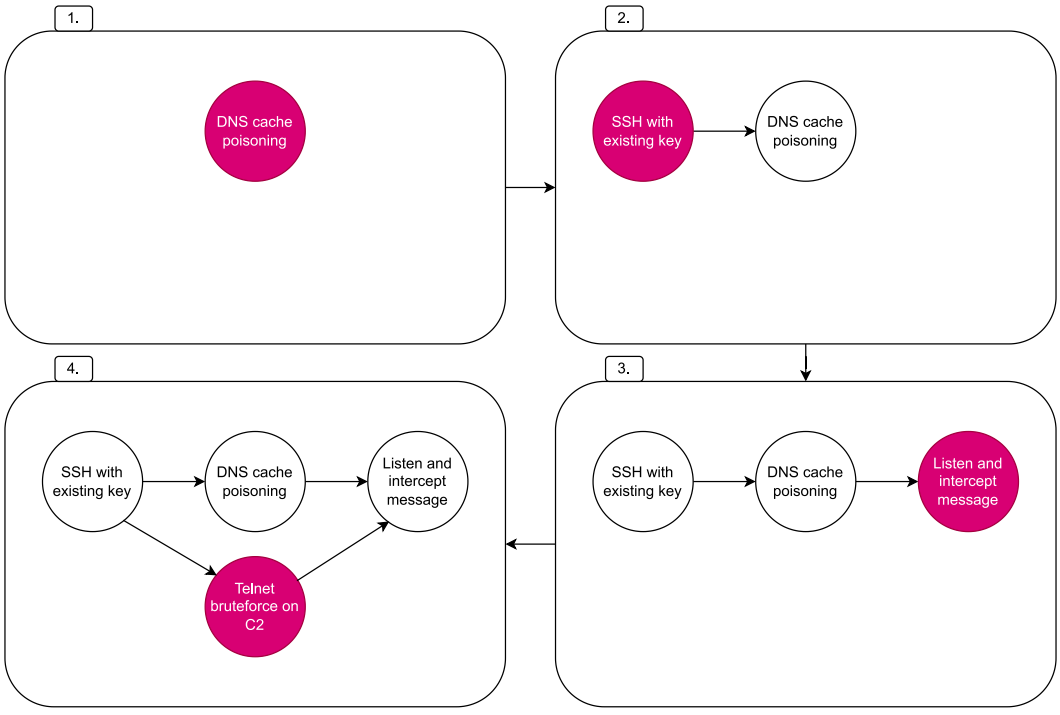

Fig. 4: Steps to build the *Satellite Siege* attack-defense flow

As shown in Fig. 4, we build the attack-defense flow for the *Satellite Siege* scenario. The first node, step **1.**, is a DNS cache poisoning attack. This exploit involves capturing DNS requests on a resolver and sending fake responses to redirect traffic to a machine

controlled by the attacker. To develop the scenario, we use the methodology's guiding questions. The first question is: *"What steps allow the trainee to reach the resolver with enough privileges for the DNS cache poisoning?"* The answer, step **2.**, involves using an *SSH* connection with a key found on another machine in the internal network. Next, we ask the second question about the same node: *"What does the DNS cache poisoning achieve?"* This exploit lets the attacker respond to DNS requests with a custom IP, redirecting traffic to a controlled machine. This leads to step **3.**, where the attacker listens for packets sent to the redirected machine, assuming another machine tries to contact the domain requested from the resolver. Finally, step **4.** addresses both questions by adding a parallel node, creating an alternative path. From the DNS resolver, the attacker can access a command-and-control (C2) machine that communicates with the satellite by brute-forcing a telnet connection, instead of redirecting traffic.

The **Helical Build Framework** enabled us to create a scenario starting from a single core node, building each step based on the previous one. This approach prevented connection issues later in the design process. For instance, the steps following the DNS cache poisoning rely on understanding how this exploit works to use it effectively, highlighting the need to build robust components before moving forward. The methodology allowed us to develop a scenario without restrictions, including an alternative path, which adds flexibility for trainees. The modular design supports reusing components, such as the DNS cache poisoning exploit with the following listening steps, in other scenarios. This example shows the methodology's strength in creating reliable, adaptable, and educationally valuable cybersecurity scenarios that simulate real-world threats effectively.

5. Implementation

The implementation of cybersecurity scenarios, as outlined in the methodology, requires careful consideration of the technical environment. Before beginning the implementation, key decisions must be made regarding the emulation mechanism and the base system for each machine in the scenario's infrastructure. These choices determine the level of virtualization and the operating environment, both of which are critical to ensuring a realistic and secure training experience. This chapter discusses the virtualization mechanisms and base system selections used in the development of the four scenarios, emphasizing their role in supporting the **Helical Build Framework**. The technologies discussed in this chapter align with the methodogoly's focus on reusable and reproducible components. This chapter gives the resources to answer the second question asked in the problem statement : *How do we implement a cybersecurity scenario design in a modular and replicable way?*.

5.1. Deployment

To deploy a cybersecurity scenario, an infrastructure is needed. When a physical network of multiple machines is not used, emulation on a single system becomes necessary. To simulate multiple machines in a company network, as required by the scenarios, different emulation methods can be used. This thesis selects two common approaches: virtual machines (VMs) and containers, often used with *Docker*[11] or *Podman*[12]. These methods provide isolation, creating secure and controlled environments for cybersecurity training. Virtualization is vital in cybersecurity scenarios for two key reasons. First, it creates a realistic environment for trainees, helping them apply skills learned in the scenario to real-world situations. Second, it ensures isolation and portability, providing a consistent environment that reduces the risk of unintended security issues. Such issues in the setup could confuse trainees, making them think these are part of the scenario, which would harm the learning experience.

These solutions need an underlying infrastructure and tool to manage them. This can be done on one machine using containers, managed by a tool like *Docker Compose*[13], to simulate networked environments. Alternatively, a dedicated system, such as a cyberrange, can host virtual machines.

Virtual machines emulate the hardware of a physical computer, allowing any operating system to run. By contrast, containers use an image built on an operating system, offering a lighter virtualization method without mimicking a full machine. As a result, a trainee with high privileges might notice they are in a container. However,

[11] https://www.docker.com/
[12] https://podman.io/
[13] https://docs.docker.com/compose/

this is unlikely to affect the scenario if it does not disrupt the tasks or exploration needed to complete it.

TABLE I: Advantages of Containers and VMs

Containers	**Virtual Machine**
Easy to deploy and manage	Hardware emulation
Hardware-agnostic	Full operating system
Portable	Consistent performance

The advantages of containers and virtual machines (VMs) are summarized in Table I. Containers are lightweight, easy to deploy, and portable, as their images and volumes can be reused across hosts. They scale on the host system without requiring hardware configuration, and tools like *Docker Compose* simplify network management for multiple containers. However, they are typically designed for single processes and may lack the precision of full system emulation. By contrast, VMs emulate hardware fully, supporting complete operating systems and enabling realistic, complex configurations suitable for corporate environments. They ensure consistent performance through fixed resource allocation but are resource-intensive and complex to manage, often requiring specialized systems like cyberranges.

Containers and VMs offer similar trainee experiences but differ in key aspects. Containers use fewer resources and deploy quickly, supporting rapid scenario development. VMs provide accurate emulation but demand more resources and setup time. Containers simplify network and IP management, while VMs need more effort for multi-machine scenarios. Although containers can handle complex setups with detailed base images, VMs are better suited for low-level emulation, ensuring reliability across diverse scenarios.

The choice of virtualization method greatly affects both the deployment and implementation of the scenario. The base system must match the chosen method, as containers and VMs require different setup approaches. While similar configurations are often possible with both methods, adapting configurations between containers and VMs may require extra effort or adjustments. Thus, choosing the right virtualization method is a key decision that shapes the scenario's technical design and implementation. Regardless of the emulation method, a base image must be chosen to define the operating environment. This can range from a full operating system to a customized image designed for the scenario's needs.

5.2. Base System

The selection of a base system for the infrastructure of cybersecurity scenarios is a critical decision that directly impacts the **Helical Build Framework**'s goals of modularity, reproducibility, and adaptability. The base system defines the operating environment for virtual machines (VMs) and containers, ensuring that configurations are consistent and reliable across different emulation methods. For VMs, a full operating system is necessary, while containers can leverage lighter images. However, given the complex requirements of cybersecurity scenarios, such as configuring multiple services, network parameters, and specific software versions, a robust base system, resembling a minimal operating system, is essential. This approach standardizes the configuration process across all emulated machines, simplifies development, and supports seamless transitions between VMs and containers, aligning with the methodology's emphasis on flexible and reusable scenario components.

a) *Challenges with Traditional Linux Distributions*

Setting up complex systems for cybersecurity scenarios is difficult when using traditional Linux distributions, such as Ubuntu[14] or other Debian-based systems[15]. These distributions are popular in industry because they support many proprietary software packages, making them a common choice for scenario infrastructure. For example, proprietary tools or specific vulnerable software versions needed for exploit demonstrations are often available in Debian-compatible formats, which simplifies setup. However, this benefit is limited because vendors regularly update packages to fix vulnerabilities, making older versions hard to find. Although Windows-based systems are useful for some enterprise scenarios, Linux was selected for this thesis due to its flexibility and open-source nature, which better support the iterative development process.

The main challenge with traditional distributions is their use of sequential commands, usually run through scripts, to configure systems. These scripts are hard to change reliably, as modifications can create unpredictable system states. For instance, in the *Satellite Siege* scenario, setting up a DNS server on Ubuntu requires installing packages, editing configuration files, and adjusting network settings. If changes are needed, such as updating the DNS resolver's cache settings, developers have four options:

- Add new commands to the existing script and run them manually. This is simple but risks conflicts with earlier configurations, leading to unexpected and non-reproducible system states.

[14] https://ubuntu.com/

[15] https://www.debian.org/

- Undo all previous commands with a cleanup script and reapply the updated script. This ensures correctness but depends on the cleanup script being accurate, which is difficult and often leads to silent errors and inconsistent setups.
- Rebuild the system from a clean state and apply the modified script. This guarantees correctness but is slow, especially for virtual machines, as it requires redeploying a new system disk, which delays the iterative process.
- Use Ansible, a tool that automates system setup with YAML-based playbooks. Ansible defines the desired system state, such as installed packages or services, and applies changes consistently. It simplifies complex setups, reduces errors compared to manual scripts, and allows configuring multiple systems at once, saving time. However, undoing changes requires creating new playbooks, which creates the same issue as mentioned before, and learning Ansible's syntax can slow initial use.

Another challenge is managing configuration files. In traditional Linux distributions, some settings are applied through commands that modify files, while others require direct edits to files located in various places on the system. Handling these files with different configuration methods is time-consuming and complex.

These options show a key limitation: traditional distributions lack an efficient and reliable way to modify configurations without risking errors or needing extensive rework. This gap highlights the need for a better approach that enables fast, reproducible, and modular configuration management, which is vital for the iterative process of the **Helical Build Framework**.

b) *NixOS*

To address the limitations of traditional Linux distributions, **NixOS**[16] was selected as the base system for the cybersecurity scenarios, offering a declarative, reproducible, and reliable approach to configuration management[10], [11]. By enabling deterministic system setups, centralized in a single file system location, NixOS provides a fifth possibility that overcomes the challenges of non-reentrant scripts, time-consuming rebuilds and complex configuration files management, aligning with the **Helical Build Framework**'s emphasis on modularity and efficiency.

Presentation of NixOS:

NixOS is a Linux distribution built around three core components: the Nix language, a functional language for defining packages and system settings; the Nix package manager, which builds and manages software based on Nix definitions; and the NixOS distribution, which leverages these tools to configure the entire system.

The Nix language is "*designed for conveniently creating and composing derivations – precise descriptions of how contents of existing files are used to derive new files. It is*

[16]https://nixos.org/

a domain-specific, purely functional, lazily evaluated, dynamically typed programming language" [10]. The Nix package manager, integrated within the distribution, ensures that software and dependencies are built consistently, pulling from the Nixpkgs repository, which contains over 120,000 packages[12]. This extensive collection supports the precise software versions needed for cybersecurity scenarios, such as specific vulnerable packages for exploit demonstrations.

The advantages and disadvantages of NixOS, derived from its official documentation and practical experience during scenario development, are summarized below[13].

TABLE II: Pros and Cons NixOS

Pros	**Cons**
Full system configuration centralized	Learning curve
Easy configuration swapping	Unavailable proprietary packages
Complete packages collection	
Reproducible builds	

The Table II lists the main benefits and drawbacks of NixOS. NixOS uses a declarative approach that allows the entire system configuration, including packages, services, users, and network settings, to be defined in a file, usually "/etc/nixos/ configuration.nix", or in a Nix Flake. This central setup reduces mistakes and makes it easy to copy the system across virtual machines (VMs) and containers. NixOS also has an optional tool called **Nix Flakes**, which makes managing multiple configurations simpler. For example, in the *Satellite Siege* scenario, one flake sets up different configurations for the DNS resolver and the trainee's workstation, making development easier across the scenario's setup. Nix Flakes also use a lock file to fix dependency versions, ensuring the same build works across different systems. While not required, Nix Flakes help by supporting reusable and version-controlled setups, fitting the methodology's focus on modularity.

NixOS's reproducible nature means systems can be rebuilt to match a specific setup, no matter what was configured before. This reliability comes from the build process, where configurations are stored in a Nix store, and symbolic links are used to add them to the system. If a configuration is wrong, the build fails, catching problems early and ensuring consistent results. This makes it easy to swap configurations. The extensive nixpkgs repository, with its vast collection of packages, is highly likely to support the specific software versions required for cybersecurity scenarios, such as vulnerable packages for exploit demonstrations[12]. However, since NixOS is less popular than distributions like Ubuntu, some proprietary packages may not be available, which is

a limitation. This was addressed during scenario development by using open-source alternatives or custom Nix configurations.

The learning curve for NixOS, especially with the Nix language and tools like Nix Flakes, is a challenge during early scenario development. Setting up services or flakes for the first time takes a lot of effort because the declarative approach is different from traditional Linux methods. However, through practice across scenarios, skills improve, showing the methodology's focus on getting better through cycles. Nix Flakes, in particular, makes later iterations easier by supporting modular, reusable configurations that help build scenarios faster.

NixOS as the Solution:

NixOS provides a better way to manage configurations by using declarative, reproducible builds that allow quick and reliable changes. Unlike the four difficult options with traditional distributions, adding new commands, reverting changes with cleanup scripts, rebuilding systems from scratch, or using Ansible scripts, NixOS lets developers edit the "/etc/nixos/configuration.nix" file and rebuild the system to a new, consistent state. This approach ignores previous setups, removing the need for manual reversions or slow VM redeployments. In the *Satellite Siege* scenario, the DNS resolver and authoritative server were set up in one file, and changes were made by updating the file and rebuilding, ensuring consistency without extra work.

Compared to traditional distributions like Ubuntu, which rely on multiple scripts that can lead to configuration errors, NixOS simplifies complex setups, such as *systemd* services or network settings, in a single file. This was essential during scenario development, where quick iterations were needed to improve components. By enabling modular configurations and easy replication, NixOS helps to build a library of reusable parts, supporting the methodology's goal of creating modular cybersecurity scenarios. Overall, NixOS's declarative and reproducible features make it a strong base system for the **Helical Build Framework**, addressing the weaknesses of traditional approaches and improving the scenario development process.

As an example, Listing 1 shows a part of the DNS resolver machine's configuration, highlighting its key components. First, the firewall is set to allow incoming DNS queries on port 53. Second, the *unbound* service is configured as the DNS resolver, forwarding all requests to the authoritative DNS server with a short cache duration to simplify DNS cache poisoning. Third, a cron job is created for privilege escalation, running a file in the writable /tmp directory as root. Finally, a user is added with an *SSH* authorized key from a separate Nix configuration file for better organization. Note that the password "admin" is included for debugging purposes and should be removed to prevent brute-force access to a higher-privileged user.

This method supports the **Helical Build Framework**'s modularity by allowing reusable configuration parts. For instance, a DNS server setup created for one scenario

```
# ...
  networking.firewall = {
    enable = false;
    allowedUDPPorts = [ 53 ];
  };

  services.unbound = {
    enable = true;
    settings = {
      server = {
        interface = [ "0.0.0.0" ];
        port = 53;
        access-control = [ "0.0.0.0/0 allow" ];

        # Cache settings
        cache-max-ttl = 2;
        cache-min-ttl = 1;
      };

      forward-zone = [
        {
          name = ".";
          forward-addr = [ config.authoritative ];
        }
      ];
    };
  };

  services.cron = {
    enable = true;
    systemCronJobs = [
      "*/5 * * * *      root    /tmp/log.sh >> /tmp/cron.log"
    ];
  };

  users.users = {
    admin = {
      isNormalUser = true;
      description = "resolver admin";
      extraGroups = [
        "networkmanager"
        "wheel"
      ];
      password = "admin";
      openssh.authorizedKeys.keys = [ config.web_public ];
    };
  };
# ...
```

Listing 1: Snippet of the Nix configuration of the DNS resolver

can be adjusted for another by changing the configuration file, keeping the core structure intact. In the scenarios, **NixOS 24.11 GNOME** was used for VMs as a full operating system, while the **nixos/nix** image, which includes the Nix package manager, was used for containers in addition of traditional purpose-built image. Although **nixos/nix** is not a full NixOS distribution, it supports detailed configurations comparable to NixOS, enabling it to overcome the single-process limitations of containers and maintain consistent setups across various emulation methods.

6. Training Scenarios

This chapter describes four cybersecurity scenarios developed in this thesis: Satellite Siege, Hotel Daemon, Patch War, and Zheng Hijack. Each scenario includes its story line, configuration, and attack-defense flow. The configurations are shown through diagrams that illustrate the minimal setup needed for each scenario. To make scenarios more realistic, additional workstations or servers can be included in a company's internal network, for example. Some machines are marked as optional, as they are theoretically needed for the scenario and play a key role, but they do not require emulation to fulfill their purpose. Instead of using the provided attack-flow builder of MITRE ATT&CK, these diagrams were recreated from scratch for better clarity. For each scenario, the attack-defense flow is explained with a specific example of an attack-defense path to reach the scenario's objective.

6.1. Satellite Siege

The first scenario, named "Satellite Siege", tasks the trainee with preventing a rogue base from spying on the population. To accomplish this, the trainee must hack a satellite ground base to intercept messages exchanged with a spy satellite. This forms the core of the scenario's storyline. To enhance the narrative, elements like a front page for the target website are included, which are detailed later in the scenario description. The exercise focuses on studying the security of a typical company network, requiring the trainee to perform network exploits to achieve the goal.

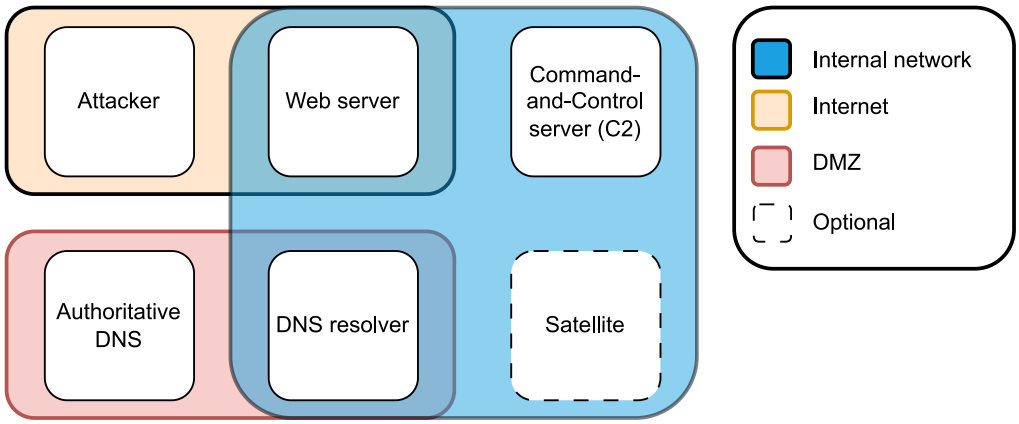

Fig. 5: *Satellite Siege* configuration

The scenario depicted in Fig. 5 has specific network requirements. The trainee starts outside the company network with internet access. The company operates a website hosted on a web server within its internal network, intentionally exposed to the internet, serving as the initial entry point for the attack. The network comprises three main

components: the web server, hosting the website and accessible to the attacker; the DNS component, consisting of a local resolver in the company network that queries an authoritative DNS server to map domain names to IP addresses; and the command-and-control (C2) server, which communicates with a satellite component via a domain name resolved by the DNS system, avoiding hard-coded IP addresses. This setup enables dynamic communication between the C2 server and the satellite. The attack focuses on accessing the C2 server and DNS resolver to achieve the objective. The satellite is optional and can be emulated as a simple service that periodically sends messages.

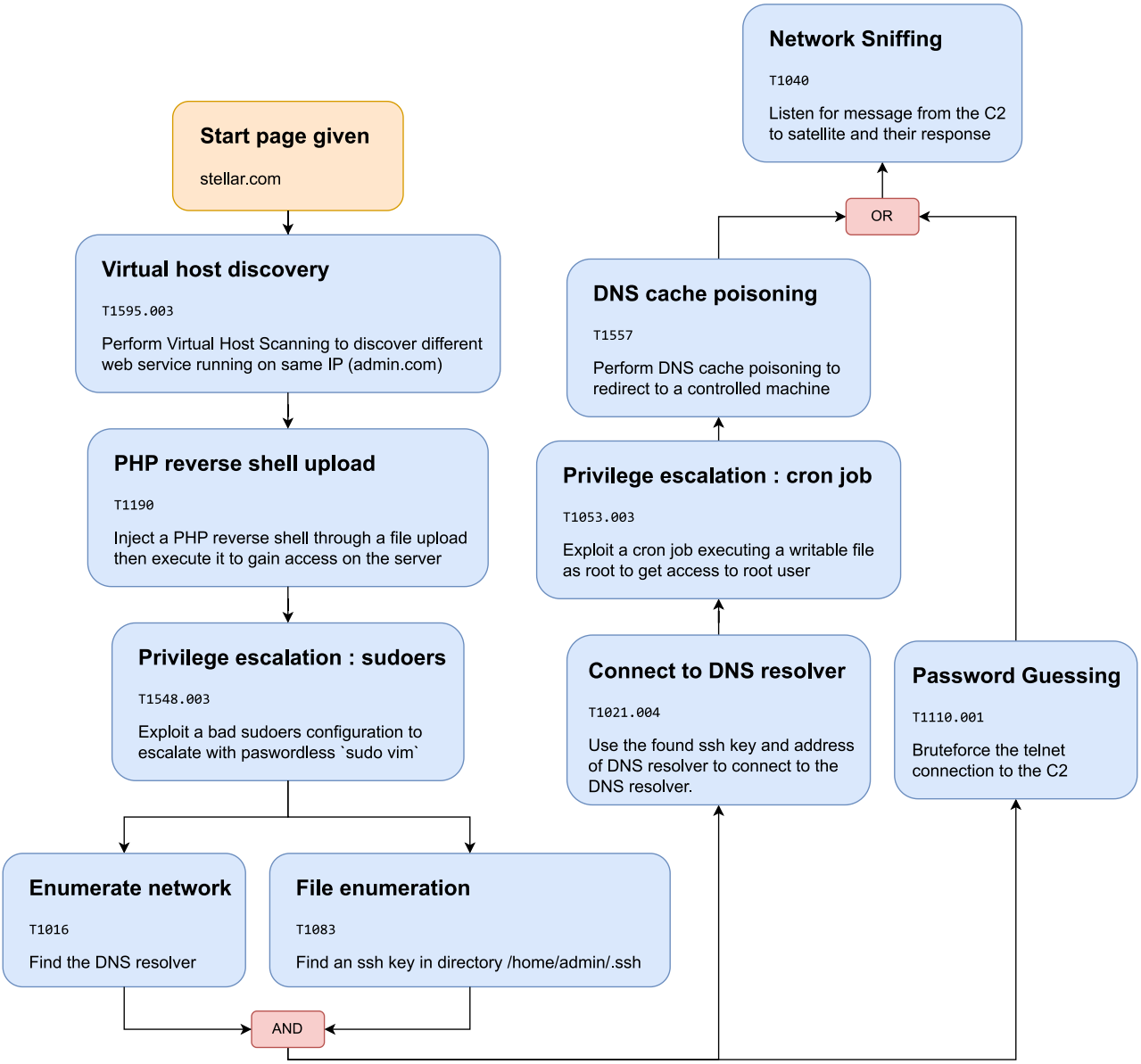

Fig. 6: *Satellite Siege* attack flow

The attack flow Fig. 6 begins with a machine connected to the internet accessing the company's official website, *stellar.com*. The attack starts by performing virtual host discovery (T1590.005[17]) on the web server IP to identify hidden websites. This is done by modifying the *Host* header in HTTP requests using a tool like *FFUF*[18] with the command `ffuf -c -w <wordlist> -H 'Host: FUZZ.com' -u "http://stellar.com/"`, revealing a hidden site: *admin.com*.

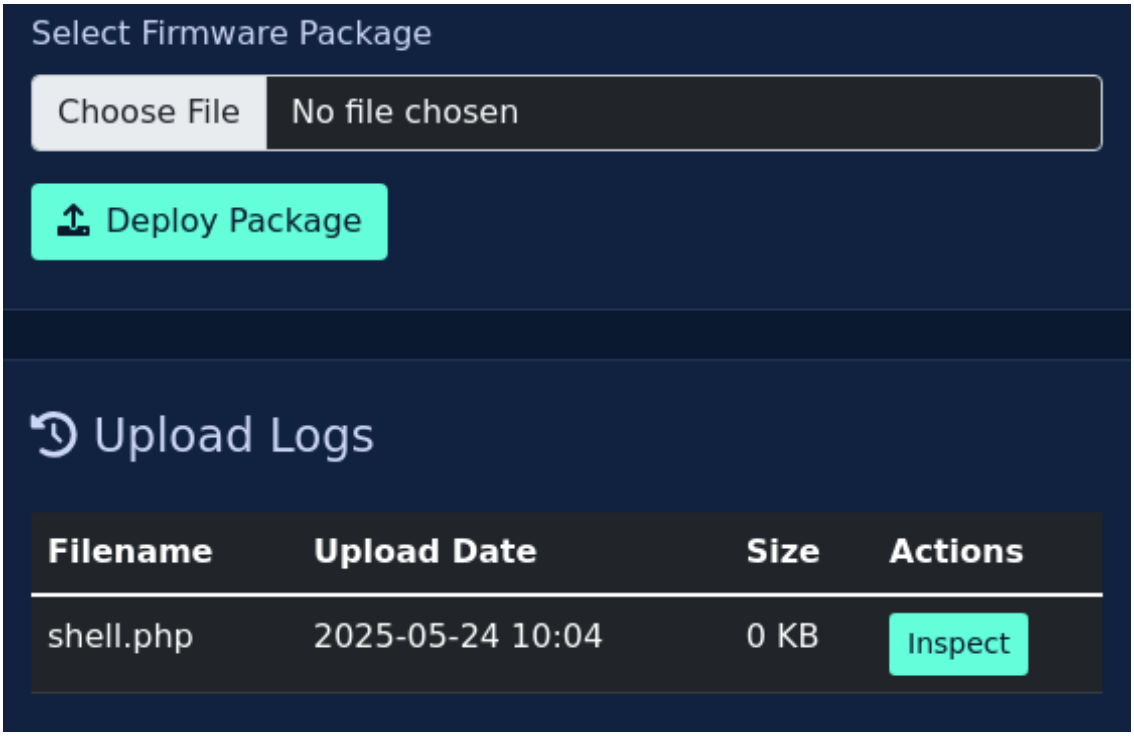

Fig. 7: Screenshot of file upload section

This website includes a form allowing file uploads and viewing, as shown in Fig. 7. The trainee can exploit this by uploading a PHP reverse shell (T1190[19]), enabling unauthorized access to the web server. A possible tool to generate the payload and establish a stable reverse shell is *Metasploit*[20]. At this stage, the trainee is connected as the web server's user, which typically has limited privileges.

To proceed, the trainee must escalate privileges. Various tools can enumerate a Linux system to identify vulnerabilities, such as Linpeas.sh[21]. The vulnerability comes from a misconfiguration in the sudoers file, allowing passwordless use of vim (likely for administrators to edit root configurations easily). This can be exploited to gain

[17] https://attack.mitre.org/techniques/T1590/005/
[18] https://github.com/ffuf/ffuf
[19] https://attack.mitre.org/techniques/T1190/
[20] https://www.metasploit.com/
[21] https://github.com/peass-ng/PEASS-ng/tree/master/linPEAS

root access by opening a shell using the command `sudo vim -c ':!/bin/sh'` (T1548.003[22]).

At this point, two paths are available. The first involves a DNS cache poisoning attack, while the second allows direct connection to the C2 server.

The DNS cache poisoning path requires two initial steps: enumerating the network to identify a machine with an open DNS port using a tool like *nmap*[23] (T1016[24]), and enumerating the file system for an *SSH* key belonging to another user (T1083[25]). With the *SSH* key, the trainee can connect to the DNS resolver (T1021.004[26]) and escalate privileges to perform the poisoning. The resolver has a cron job executing a file in the */tmp* directory as root: `*/5 * * * * root /tmp/log.sh >> /tmp/cron.log`, which is writable by any user (T1053.003[27]). For example, the trainee could use this to write an *SSH* authorized key, allowing a root connection to the resolver. As root, the trainee can initiate DNS cache poisoning by intercepting DNS queries and responding with a fake IP (T1557[28]), such as that of the controlled web server, using a tool like *Scapy*[29]. A snippet of a potential script is shown in Listing 2. This allows the trainee to capture network communications by listening for packets on the web server with *tcpdump* or another tool to capture network traffic (T1040[30]).

The Scapy script in Listing 2 performs DNS cache poisoning by intercepting DNS queries. It checks for DNS request packets (`qr == 0`) targeting a specific domain (`TARGET_DOMAIN`). When found, it crafts a spoofed response, mapping the queried domain to a fake IP (`FAKE_IP`). This response is sent back to the resolver, to redirect traffic to the attacker's chosen IP, enabling interception of communications.

The alternative path is more straightforward. The trainee enumerates the network to locate the C2 server and discovers an open Telnet port. By brute-forcing the Telnet connection, the trainee gains root access to the C2 server using a tool like *Hydra*[31] with the command `hydra -L <username_list> -P <password_list> telnet://<target_ip>` (T1110.001[32]). From there, they can monitor messages sent from or received by the satellite using a tool like *tcpdump*, achieving the scenario's objective.

---

[22] https://attack.mitre.org/techniques/T1548/003/
[23] https://nmap.org/
[24] https://attack.mitre.org/techniques/T1016/
[25] https://attack.mitre.org/techniques/T1083/
[26] https://attack.mitre.org/techniques/T1021/004/
[27] https://attack.mitre.org/techniques/T1053/003/
[28] https://attack.mitre.org/techniques/T1557/
[29] https://scapy.net/
[30] https://attack.mitre.org/techniques/T1040/
[31] https://github.com/vanhauser-thc/thc-hydra
[32] https://attack.mitre.org/techniques/T1110/001/

```
def dns_spoof(pkt):
    if pkt.haslayer(DNS) and pkt.getlayer(DNS).qr == 0:
        dns_layer = pkt.getlayer(DNS)
        query_name = dns_layer.qd.qname
        # Check if this is a query for the targeted domain.
        if TARGET_DOMAIN in query_name:
            # Build DNS answer packet
            spoofed_pkt = (
                IP(src=pkt[IP].dst, dst=pkt[IP].src)
                / UDP(sport=pkt[UDP].dport, dport=pkt[UDP].sport)
                / DNS(
                    id=dns_layer.id,
                    qr=1,
                    aa=1,
                    qd=dns_layer.qd,
                    an=DNSRR(rrname=query_name, ttl=10, rdata=FAKE_IP),
                )
            )

            # Send the spoofed response
            send(spoofed_pkt, verbose=0)


def main():
    sniff(filter="udp port 53", prn=dns_spoof, iface=INTERFACE, store=0)

if __name__ == "__main__":
    main()
```

Listing 2: Scapy script for DNS cache poisoning

6.2. *Hotel Daemon*

The second scenario, named “Hotel Daemon”, requires the trainee to locate and trap a criminal hacker inside a hotel room. The scenario takes place in a hotel environment, where the trainee must compromise the network by exploiting Internet of Things (IoT) devices to identify and lock the criminal in their room. This exercise emphasizes the importance of secure access management for IoT devices to protect users and prevent unauthorized control.

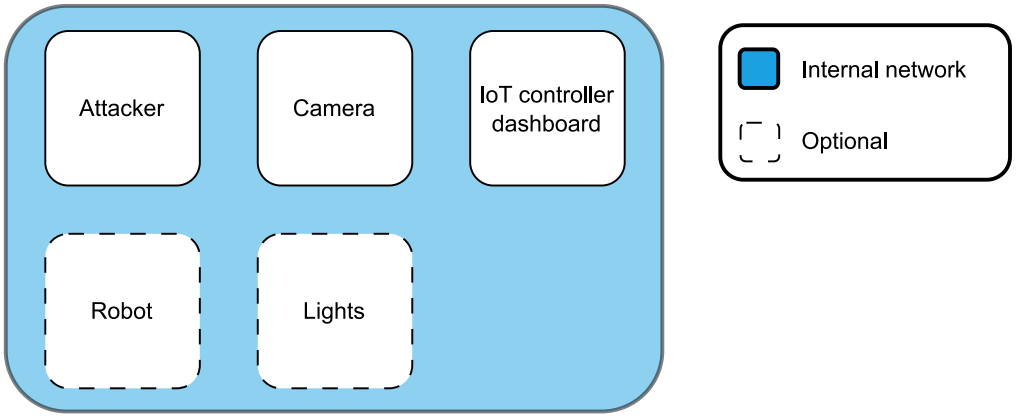

Fig. 8: *Hotel Daemon* configuration

The configuration for the “Hotel Daemon” scenario Fig. 8 consists of a single network, where the trainee begins as the attacker, for example, by connecting via WiFi. The network includes various IoT devices, such as lights, robots, and cameras. However, the lights and robots are not directly accessible to the attacker and thus do not require separate instances in the configuration. Additionally, an IoT controller server manages all the hotel’s IoT devices with *Node-RED*[33], serving as a central component of the setup.

The attack Fig. 9 begins with network enumeration with tool like *nmap* (T1046[34]), which provides the trainee with the IP addresses and open ports of two key machines: the camera controller and the IoT device management server. During enumeration, an HTTP request on an open port reveals that the cameras run a vulnerable version of *Hikvision* software, susceptible to CVE-2017-7921[35]. The trainee exploits this vulnerability[36] to bypass authentication (T1190[37]), gaining access to camera images, configuration details, and software credentials for later use. Initially, the camera image is completely black, suggesting an issue with the camera.

Using the credentials obtained, the trainee connects to the IoT management server via *SSH* (T1078.002[38]). At this point, the attack splits into two paths, both of which must be completed to achieve the scenario’s objective. This description starts with the right path.

On the IoT server, earlier enumeration identified an open MQTT port running a vulnerable version of *Mosquitto*[39] (1.4.14) (T1046[40]), which is susceptible to CVE-2017-7651[41]. Exploiting this vulnerability[42] allows the trainee to crash the *Mosquitto* service. After doing so, the trainee checks the camera feed again and notices that the room is now illuminated, indicating that the *Mosquitto* service acts as a keep-alive mechanism for the lights, activating them in case of a network incident (T1499.003[43]). We can now see the criminal in the room Fig. 10.

[33] https://nodered.org/
[34] https://attack.mitre.org/techniques/T1046/
[35] https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-7921
[36] Example of exploit script https://github.com/JrDw0/CVE-2017-7921-EXP
[37] https://attack.mitre.org/techniques/T1190/
[38] https://attack.mitre.org/techniques/T1078/002/
[39] https://mosquitto.org/
[40] https://attack.mitre.org/techniques/T1046/
[41] https://nvd.nist.gov/vuln/detail/cve-2017-7651
[42] An attack script can be found in the github of this thesis https://github.com/noahfraiture/thesis
[43] https://attack.mitre.org/techniques/T1499/003/

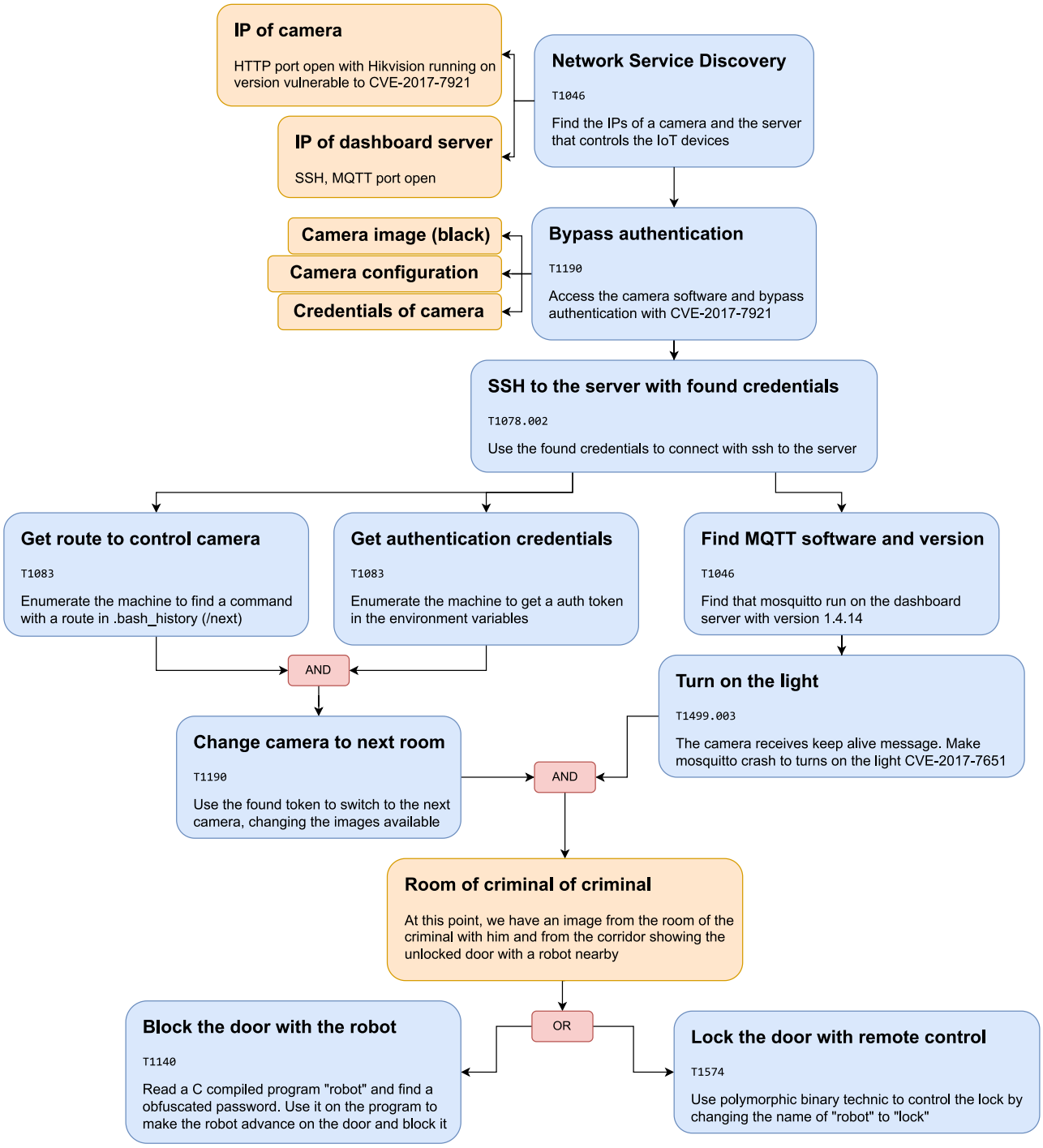

Fig. 9: *Hotel Daemon* attack flow

Fig. 10: Criminal room

The left path focuses on switching the camera to view other areas. By enumerating the IoT server's file system with tool like *LinPEAS*, the trainee discovers an environment variable used as a token and a route in the *".bash_history"* file, *"/next"*, which uses the same token to switch cameras (T1083[44]). With this information, the trainee can toggle the camera view between two rooms and the corridor Fig. 11 (T1190[45]).

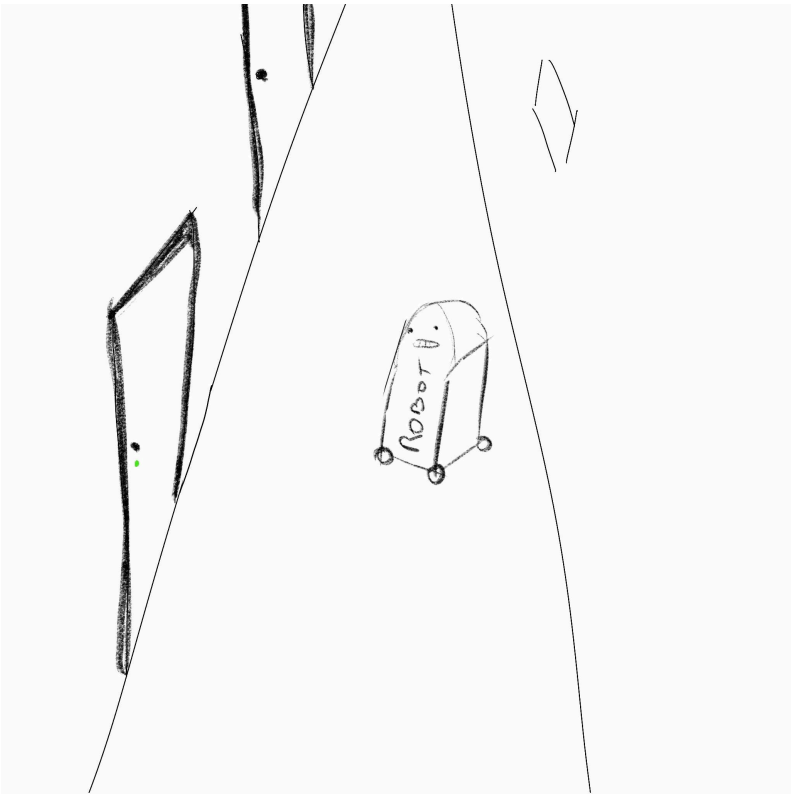

Fig. 11: Corridor

[44]https://attack.mitre.org/techniques/T1083/
[45]https://attack.mitre.org/techniques/T1190/

After completing both paths, the trainee locates the room of the criminal from the corridor, with the door unlocked, as seen from camera. Two methods are available to lock the criminal in, both involving a binary file named “robot” on the IoT server. Both methods require binary analysis to understand the file’s behavior. The first method involves a buffer overflow attack to bypass an authorization check (T1140[46]) shown at Listing 3, allowing the trainee to control the robot. By directing the robot correctly, the trainee blocks the door, Fig. 12, trapping the criminal.

```
struct robot_control rc;
rc.authorized = 0;

printf("Available robots:\n");
printf("1. RobotA\n");
printf("2. RobotB\n");
printf("3. RobotC\n");

printf("Enter the ID of the robot you want to control: ");
gets(rc.id);
```

Listing 3: C snippet of robot.c for robot movements

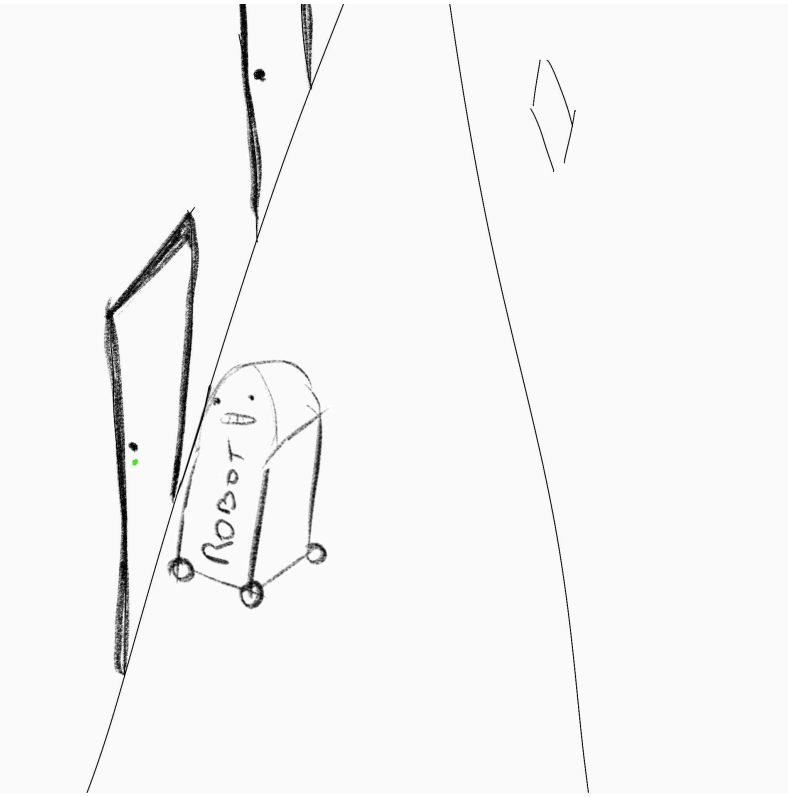

Fig. 12: The robot blocks the door

The second method relies on recognizing that the “robot” file is a polymorphic binary, as shown at Listing 4, a technique where the behavior of the program changes based on

[46]https://attack.mitre.org/techniques/T1140/

its name (using `argv[0]`), often implemented with symlinks. By renaming the file to "lock", the trainee accesses a different function, which requires finding an obfuscated password within the binary (T1574[47]). Using this password, the trainee executes the file to activate a smart lock, securing the door and trapping the criminal inside Fig. 13.

```
if (strstr(argv[0], "robot") != NULL) {
  return robot_control();
} else if (strstr(argv[0], "lock") != NULL) {
  return lock_control();
```

Listing 4: C snippet of robot.c for the lock mechanism

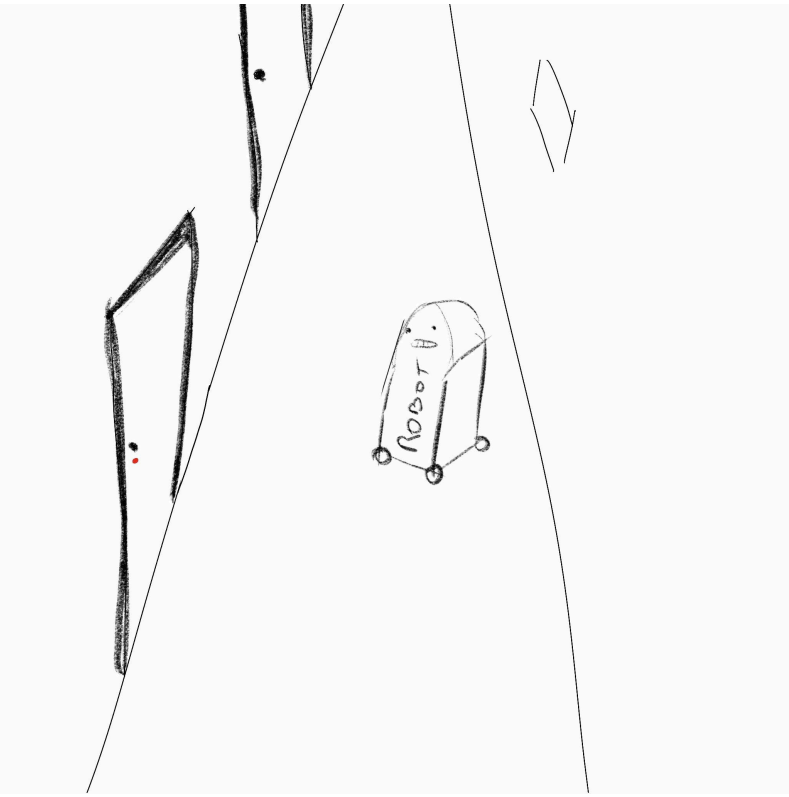

Fig. 13: Smart lock locked

6.3. Patch War

The third scenario, named "Patch War", tasks the trainee with patching a vulnerable online game. In the scenario, the trainee notices players exploiting a flaw in their favorite game, causing scores to increase unnaturally quickly. The trainee must infiltrate the game server by exploiting a vulnerability, patch the flawed code, and restart the game to deploy the fix. This exercise explores the basics of forensics and highlights the critical role of secure coding in defensive cybersecurity, demonstrating that even a secure network is ineffective if the application code contains weaknesses.

[47] https://attack.mitre.org/techniques/T1574/

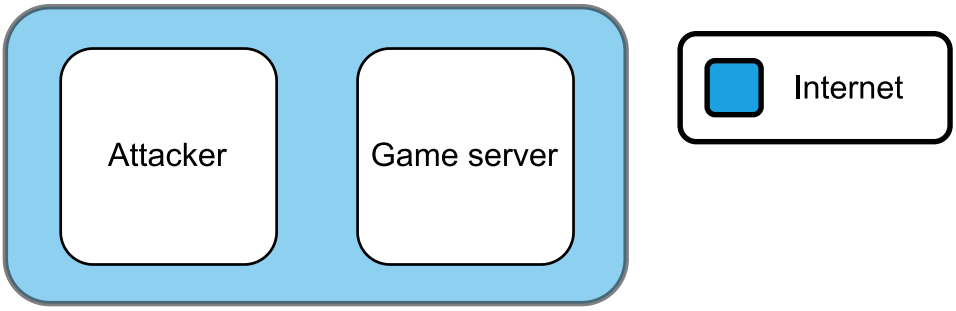

Fig. 14: *Patch War* configuration

The configuration for the “Patch War” scenario Fig. 14 is straightforward. The trainee, acting as the attacker, operates from a machine with internet access. The game is hosted on a separate server, which exposes an HTTP port to allow players to interact with the game via a website. All potential players, including the trainee, have similar access to the game server from their machines.

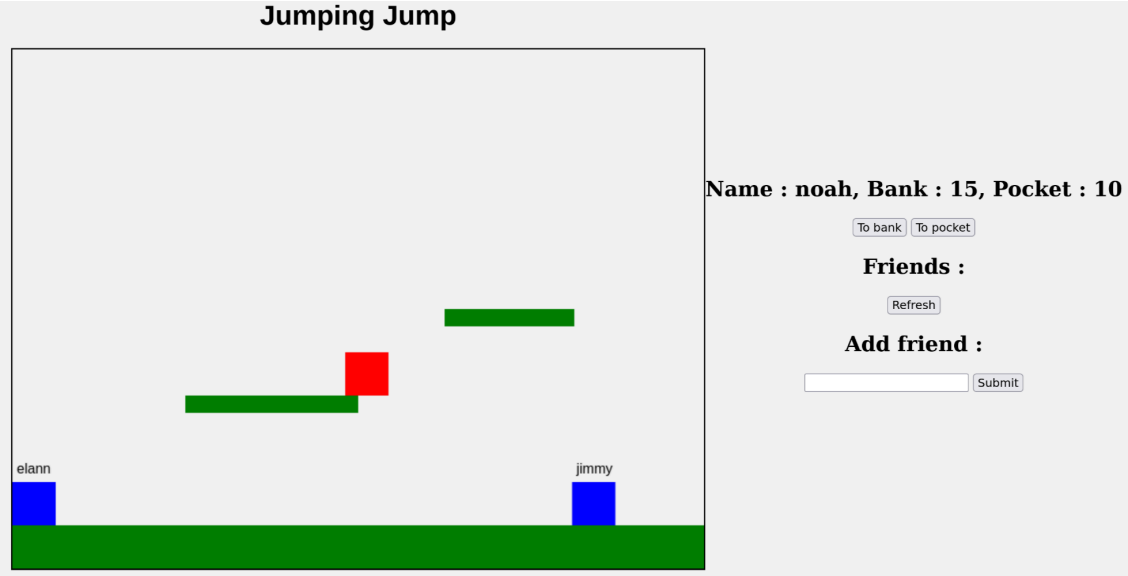

Fig. 15: Game Jumping Jump

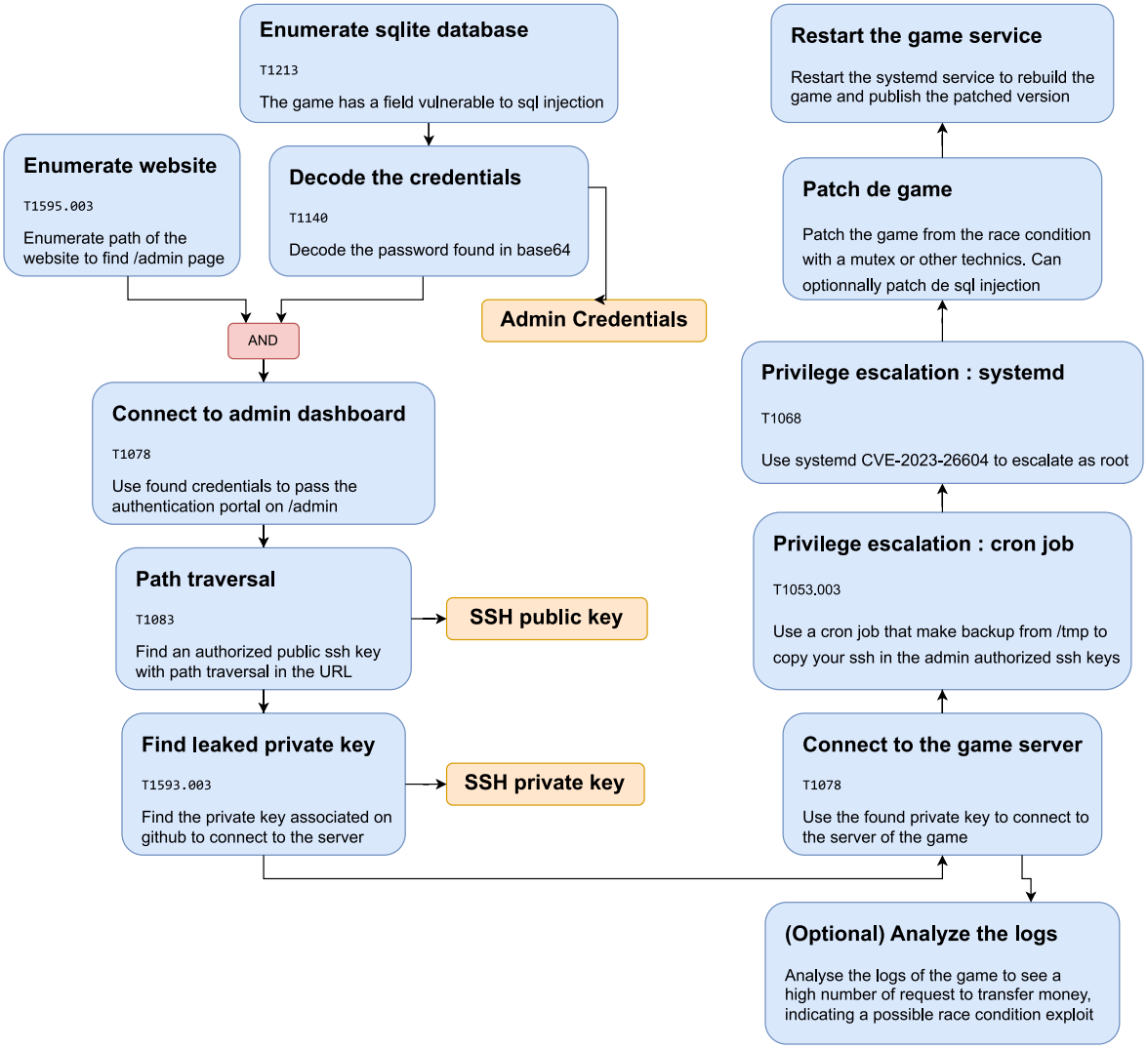

Fig. 16: *Patch War* attack flow

The attack flow Fig. 16 begins at the page of the game Fig. 15. The player can use arrow keys to move, the space key to jump, and click on other players to add them as friends. It is also possible to add them as friends by entering their name in the field for that purpose. This field can be used for an SQL injection. By carefully probing the form, the trainee can determine the database table structure and extract a table named `secrets` (T1213[48]), which contains encoded credentials. The first and only row of this table can be extracted with the input Listing 5 for example.

```
`jimmy'); INSERT INTO players(name) VALUES ((SELECT pass FROM secrets LIMIT
1)); INSERT INTO friends (p1, p2) VALUES ('noah', (SELECT pass FROM secrets
LIMIT 1)); --`
```

Listing 5: SQL query injection

[48]https://attack.mitre.org/techniques/T1213/

This results in two friends Fig. 17.

Fig. 17: Friends gained with SQL injection

Decoding the name of the first friend in base64 reveals the credentials *"admin"* (T1140[49]). In parallel, enumeration of the website with a tool like *FFUF* yields interesting results Listing 6 (T1595.003[50]). A protected page, *"/admin"*, is accessible with the credentials obtained earlier: *"user: admin, pass: admin"* (T1078[51]).

```
profile     [Status: 200, Size: 35, Words: 9, Lines: 1, Duration: 4ms]
admin       [Status: 401, Size: 13, Words: 1, Lines: 2, Duration: 0ms]
friends     [Status: 200, Size: 89, Words: 17, Lines: 1, Duration: 2ms]
ping        [Status: 200, Size: 4, Words: 1, Lines: 1, Duration: 1ms]
jump        [Status: 200, Size: 0, Words: 1, Lines: 1, Duration: 1015ms]
```

Listing 6: Output of directory enumeration

Next, the trainee uses the admin credentials to access the *"/admin"* page, which contains unauthorized files. While the files themselves are not significant, the trainee notices that the URL is vulnerable to a path traversal attack (T1083[52]). By exploiting this vulnerability, the trainee accesses the list of authorized *SSH* keys for the server at the path `/admin?file=../../.ssh/authorized_keys`. Using open-source intelligence (OSINT) techniques, the trainee searches for the corresponding private key, which has been inadvertently leaked on GitHub[53] (T1593.003[54]). With the private key, the trainee establishes an *SSH* connection to the game server (T1078[55]).

At this stage, the trainee's goal is to patch the game. Optionally, the trainee can review the game's logs to identify the source of the exploit used by other players. However, the initial *SSH* credentials provide limited privileges, requiring privilege escalation to modify the game files. The first escalation exploits a cron job responsible for creating backups: `"*/5 * * * * web rsync -a /tmp/server/ /home/`

[49] https://attack.mitre.org/techniques/T1140/
[50] https://attack.mitre.org/techniques/T1595/003/
[51] https://attack.mitre.org/techniques/T1078/
[52] https://attack.mitre.org/techniques/T1083/
[53] https://github.com/noahfraiture/thesis/tree/main/scenario-patch/tools/keys
[54] https://attack.mitre.org/techniques/T1593/003/
[55] https://attack.mitre.org/techniques/T1078/

`admin"` (T1053.003[56]). The trainee writes a custom file that adds a new authorized *SSH* key, which the cron job copies to the appropriate location, granting access to a higher-privileged user account.

This new account still lacks sufficient privileges to patch the game. A second privilege escalation leverages CVE-2023-26604[57], a vulnerability in *systemd* that allows the trainee to gain root access (T1068[58]). The vulnerability involves a sudoers configuration that permits the trainee to run a *systemd* command, such as `systemctl status`, on a small screen, enabling the execution of a command from the *less* program as root, such as *bash*. As root, the trainee can modify the game files, which are stored on the server and managed by a *systemd* service that builds and runs the game on startup. To deploy the patch, the trainee edits the game code and restarts the *systemd* service, triggering a rebuild and launching the patched version.

To patch the game, the trainee manually analyzes the code and identifies a race condition between two database calls, which causes the scoring exploit. The trainee can fix this by implementing a lock mechanism Listing 7 or consolidating the two calls into a single SQL command. Additionally, to secure the website, the trainee can patch the SQL injection vulnerability by replacing the vulnerable form with parameterized queries, preventing further exploits.

[56] https://attack.mitre.org/techniques/T1053/003/

[57] https://nvd.nist.gov/vuln/detail/cve-2023-26604

[58] https://attack.mitre.org/techniques/T1068/

```
<<<BEFORE
func (p *player) toBank(db *DB) error {
  if _, err := db.Exec("UPDATE players SET bank = bank + pocket WHERE name
= ?", p.Name); err != nil {
    return err
  }
  if _, err := db.Exec("UPDATE players SET pocket = 0 WHERE name = ?",
p.Name); err != nil {
    return err
  }
  p.Bank += p.Pocket
  p.Pocket = 0
  return nil
}
===
func (p *player) toBank(db *DB) error {
  db.Lock()
  defer db.Unlock()
  if _, err := db.Exec("UPDATE players SET bank = bank + pocket, pocket = 0
WHERE name = ?", p.Name); err != nil {
    return err
  }
  p.Bank += p.Pocket
  p.Pocket = 0
  return nil
}
>>>AFTER
// Same for toPocket function
```

Listing 7: Change to make in the game of the code to patch the exploit

6.4. Zheng Hijack

The final scenario, “Zheng Hijack”, challenges the trainee to dismantle a botnet causing unusual network traffic and data leaks within a company’s internal network. The trainee, acting as an experienced cybersecurity expert, begins on a workstation suspected of being infected. This scenario emphasizes forensic analysis and reverse engineering, focusing on a complex reverse engineering task to address a novel, previously unstudied threat. The exercise aims to develop skills in identifying and neutralizing sophisticated malware in a realistic setting.

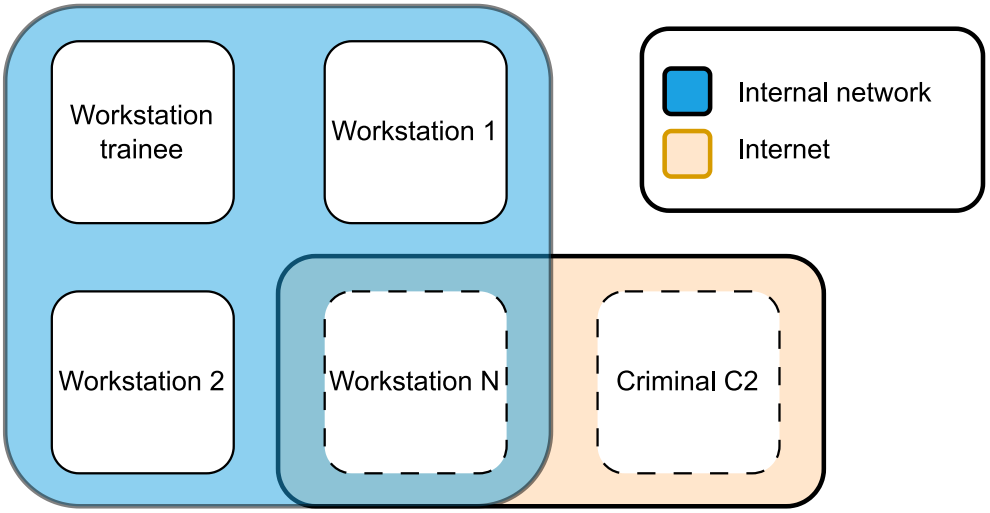

Fig. 18: *Zheng Hijack* configuration

The configuration Fig. 18 is simple. The company operates an internal network with at least three workstations, one of which is the trainee's starting point. The exact number of workstations is flexible, as it does not significantly affect the scenario, but a minimum of three ensures the botnet's expected behavior. Additionally, a command-and-control (C2) server, managed by the attacker, connects to infected machines via a backdoor exposed to the internet.

The trainee's first task of the attack Fig. 19 is to identify the botnet, which hides its process name to evade detection. Several forensic techniques can reveal it, such as analyzing network usage, inspecting dumped memory, or identifying unusual open ports (T1057[59]). Terminating the process is ineffective, as it restarts automatically. Instead, the trainee must reverse engineer the botnet to understand its behavior and develop a strategy to shut it down permanently. A tool such as *ss* can find the process. For example the command `ss -tunap` shows us a process with port 8080 open and an unusual process name `main` Listing 8.

```
Netid  State    Recv-Q  Send-Q  Local Address:Port   Peer Address:Port
Process
udp    UNCONN   0       0          127.0.0.11:56075       0.0.0.0:*
udp    UNCONN   0       0             0.0.0.0:68          0.0.0.0:*
users:(("dhcpcd",pid=301,fd=12))
tcp    LISTEN   0       4096       127.0.0.11:40383       0.0.0.0:*
tcp    LISTEN   0       4096       172.20.0.2:8080        0.0.0.0:*
users:(("main",pid=183,fd=3))
tcp    LISTEN   0       4096               *:22                *:*
users:(("systemd",pid=1,fd=145))
```

Listing 8: *ss* tool output

[59]https://attack.mitre.org/techniques/T1057/

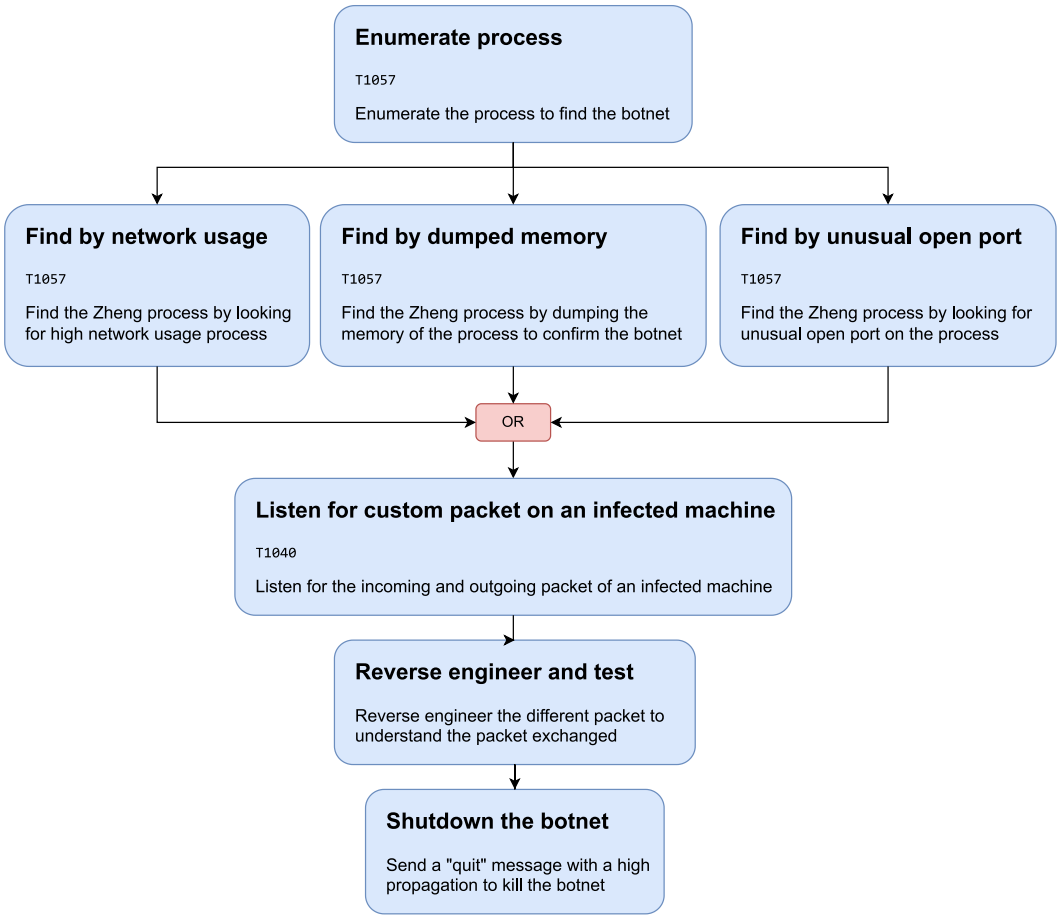

Fig. 19: *Zheng Hijack* attack flow

The botnet operates as a tree-like structure within the company's internal network, with infected machines communicating hierarchically. Each infected machine periodically polls its parent machine to check for commands, minimizing external internet traffic to evade detection. The C2 server, controlled by the attacker, sends commands to infected machines, which can propagate these commands to their child nodes based on a "hop" parameter. If a parent machine is disabled, its child machines automatically reconnect to the C2 server, ensuring the botnet's resilience. The botnet spreads to new machines through *SSH* brute-force attacks and deletes its initial infection file to cover its tracks, making it difficult to remove by shutting down individual machines or processes. The company cannot halt all systems, as critical servers may also be infected.

The C2 server can issue four commands to infected machines:

- **Sleep**: Pause polling for a specified duration.
- **Spread**: Attempt to infect other machines.
- **Exec**: Execute a command and return the result.
- **Quit**: Terminate the botnet process.

The trainee's objective is to send the "Quit" command to an infected machine with a high "hop" value, ensuring it propagates to all child nodes and shuts down the entire botnet. To achieve this, the trainee captures and analyzes network packets (T1040[60]) to understand the botnet's communication protocol. By reverse engineering the botnet, the trainee can create a process that mimics the C2 server's behavior, sending the "Quit" command to an infected machine. This approach leverages the botnet's own propagation mechanism to disable it, demonstrating a modular and reusable component in the scenario design, as the reverse-engineered process can be adapted for similar threats.

[60]https://attack.mitre.org/techniques/T1040/

7. Evaluation

To address the research question *"How can we evaluate the methodology?"*, this chapter examines the effectiveness of the **Helical Build Framework**. Evaluating a methodology is complex and ideally involves peer review, which was not conducted in this thesis but could be explored in future work. Instead, this chapter focuses on validating the methodology to ensure a minimum level of quality and effectiveness through several factors. The importance of narrative design is also discussed, a well-crafted narrative enhances trainee engagement, making scenarios more effective by providing a compelling context for learning.

7.1. Validation through Scenario Creation

Since this thesis lacks peer review, validation was conducted internally.

The **Helical Build Framework** was developed alongside the four scenarios, with each influencing the other's growth. Initially, a naive approach was adopted, exploring various possibilities that gradually shaped the final agile methodology. For the *Satellite Siege* scenario, development began with a DNS cache poisoning attack as the core concept, around which the scenario was constructed. This naive approach resembled the **Helical Build Framework** by starting with a single point and expanding iteratively. The parallel development of scenarios and methodology allowed each to refine the other, creating a stronger framework through continuous improvement. The development of the scenarios shaped the **Helical Build Framework**, and this methodology remains effective for creating the four scenarios presented in this thesis, demonstrating its ability to produce any scenario.

For the *Hotel Daemon* scenario, a different approach was tested, designing and configuring each IoT device separately before connecting them later. The goal was to use IoT devices in the scenario, so configuring each device independently seemed intuitive. However, this method faced significant challenges.

First, configuring IoT devices was difficult because real-world devices often rely on proprietary software, which is hard to include in a scenario, even with known vulnerabilities. To address this, simplified software versions were developed to replicate key features and vulnerabilities of the *Hikvision* system used in the final scenario, as described in *Hotel Daemon*. This issue also occurs with the final methodology, but the simplified software may behave differently, requiring adjustments to the preceding and following steps in the attack-defense flow. When multiple steps must be connected, this becomes a growing problem, unlike the step-by-step approach where the scenario creator knows exactly what the preceding and following steps are.

Second, connecting devices was challenging because IoT devices operate differently, and some require a central server with specific software. Without established attack-defense flow, it is difficult to know what each device needs during configuration. For example, when setting up a camera, its purpose, such as brute-forcing another device, is unclear, making it hard to define a minimal realistic configuration. Emulating a device like a connected camera using a full Linux distribution, such as Kali Linux, is not accurate without understanding its real-world internal system and intended use. As a result, this approach produced several incomplete devices that could not be realistically linked, highlighting the limitations of non-iterative methods.

Redesigning the *Hotel Daemon* scenario using an approach closer to the **Helical Build Framework** improved the process. Development started with a single device, ensuring its configuration was clear before adding the next. This clarified how devices could connect or what was needed to access them, giving information to run an adequate system on the machine instead of a full Linux distribution. The iterative and modular approach enhanced the scenario's design, creating reliable and reusable components. This work and the next scenarios, *Patch War* and *Zheng Hijack*, developed using an approach increasingly similar to the **Helical Build Framework** showed that this methodology provides a clear and flexible process for creating effective scenarios. The iterative improvement of both the methodology and the scenarios ensured that insights from each scenario enhanced the framework for future use.

7.2. Soundness and Completeness

One way to assess a methodology is correct is by examining its soundness and completeness. Soundness means that every scenario produced by the methodology is correct, while completeness means that every correct scenario can be created. To define correctness of scenarios, we refer to the verification techniques described in the *Methodology* chapter. A scenario is considered correct if there is a clear path from the starting point to the goal, ensuring the scenario is achievable.

Is the **Helical Build Framework** sound? By design, the methodology continues until a correct scenario is produced, meaning every scenario it creates is correct. Therefore, the methodology is sound.

Is the methodology complete? It allows adding new steps anywhere in the attack-defense flow, as long as they connect to another step. The only limitation is that it cannot create steps that are not linked to others, as these would be unreachable and lead nowhere. Such steps are considered meaningless, so the methodology can be regarded as complete. However, even with post-validation using graph analysis, the methodology may allow scenarios with nodes that have no incoming links (unreachable steps) or no outgoing links (steps that lead nowhere). While nodes without outgoing

links can serve as dead ends, which may be useful, nodes without incoming links are useless.

7.3. Narrative importance

The storyline of a cybersecurity scenario is vital, providing the context for the trainee's actions and enhancing the learning experience. While scenario configurations and attack-defense flow, are essential for functionality, the narrative might seem less important. However, it plays a key role in engaging trainees[14]. This section examines the importance of narrative design by exploring scenario fidelity and the benefits of user-focused, engaging narratives, drawing on relevant literature to support the methodology developed in this thesis.

a) *Fidelity and Learning Outcomes*

Fidelity refers to the degree of realism in a scenario, reflecting how closely it mirrors real-world conditions. Research on simulation-based training, including studies in medical education and aviation, suggests that high fidelity does not consistently lead to superior learning outcomes[15], [16]. For instance, low-fidelity simulators, which are less realistic but task-focused, are often as effective as high-fidelity counterparts, particularly for novice learners. Factors such as deliberate practice, structured feedback, and task-specific training, are more significant determinants of learning success than realism alone. These findings indicate that cybersecurity scenarios need not prioritize hyper-realistic environments to be effective, allowing flexibility in narrative design.

b) *Engagement through Fun and Gamification*

Beyond fidelity, the engagement fostered by a scenario's narrative significantly influences learning outcomes. Research on gamification demonstrates that fun, user-experience-focused scenarios can enhance motivation and skill acquisition, regardless of their realism[14], [17], [18], [19]. For example, gamified cybersecurity training has been shown to improve the self-efficacy of learners and reduce risky behaviors, such as clicking phishing links, by making the experience enjoyable and immersive. The effectiveness of gamification depends on factors like tailored design, sufficient duration, and alignment with the learner's context, but its potential to increase engagement is clear. A well-crafted narrative, whether depicting a realistic espionage operation or a fictional cyberadventure, can leverage gamification principles to create a compelling *"suspension of disbelief"*[20], thereby enhancing the trainee's investment in the scenario[17], [18].

c) *Narrative Design in the Helical Build Framework*

The literature underscores that effective cybersecurity scenarios prioritize engagement over fidelity[15], [16], [17], [18]. A fun and immersive narrative, supported by gamification strategies, can improve learning outcomes by motivating trainees and

fostering active participation. In developing the four scenarios for this thesis, narrative design was prioritized to create engaging experiences that balance technical rigor with user-focused appeal. For instance, the *Satellite Siege* scenario uses an espionage-themed narrative to draw trainees into the task, demonstrating that a compelling story can enhance the learning process without requiring strict adherence to real-world conditions.

8. Discussion

This thesis explored three research questions: how to create a cybersecurity scenario (*Methodology*), how to apply it in practice (*Implementation*), and how to assess the methodology's effectiveness (*Evaluation*). This chapter reviews the limitations to the solutions offered, and suggests future research to improve the methodology's effectiveness and applicability.

8.1. Limitations

The *Evaluation* chapter revealed limitations in the virtualization methods used for implementation: virtual machines (VMs) and containers. VMs provide full hardware emulation for a complete operating system, offering high realism but requiring significant resources. Containers are lightweight and easier to manage but may not fully replicate real-world scenarios, particularly for low-level techniques like kernel exploits (e.g., CVE-2016-5195[61]). This limitation reduces their suitability for some cybersecurity scenarios, and no solution currently combines the strengths of both VMs and containers. As a result, scenario developers must select the appropriate method based on the scenario's requirements and the available infrastructure for deployment.

The evaluation of the **Helical Build Framework** and its scenarios relied on self-assessment, focusing on soundness (ensuring scenarios are valid) and completeness (ensuring the methodology can generate any valid scenario). However, this approach lacked external validation, such as peer review or user testing. The methodology could be validated by external cybersecurity trainers who create cybersecurity scenarios using it, and the scenarios could be tested by external trainees who attempt to complete the scenarios of this thesis. Without feedback from actual users, the methodology's usability, clarity, and the scenarios' effectiveness in training settings remain unverified.

The choice of NixOS as the base operating system introduced additional limitations. While NixOS offers advantages for reproducibility, many proprietary packages are only available for Debian-based distributions. Potential solutions, such as hybrid systems combining Nix with a Debian-based distribution or using emulation layers, were noted but not investigated[21], restricting the framework's flexibility for broader use.

8.2. Future work

The **Helical Build Framework**, developed through the iterative design of four cybersecurity scenarios, requires further validation to establish its reliability. Creating only four scenarios is insufficient to confirm the methodology's robustness. Wider adoption

[61] https://www.redhat.com/en/blog/understanding-and-mitigating-dirty-cow-vulnerability

by diverse developers could reveal additional limitations and provide evidence of the framework's ability to consistently produce reliable, modular, and reusable scenarios. Large-scale testing would help assess whether the methodology is complete, i.e., capable of generating any correct scenario, a property difficult to prove with limited examples, as discussed in the *Evaluation* chapter.

Future research should involve collaboration with external peers. They could apply the **Helical Build Framework** to create scenarios, offering feedback on its process and usability. Trainers could create scenarios with the **Helical Build Framework** and trainees could test the scenarios, evaluating their engagement and educational value. Such user testing would identify practical challenges and refine the methodology to better meet training needs.

Improved evaluation methods are needed to assess scenario quality and effectiveness. A study by Prümmer et al. (2024) reviewed cybersecurity training methods, noting that the pretest-posttest approach is commonly used to measure trainee performance before and after scenarios[7]. This method works well for short-term learning but does not capture long-term retention or real-world applicability. Future work could explore standardized evaluation techniques, such as long-term performance studies or metrics for scenario design quality, to validate the framework's outcomes and ensure scenarios support diverse training goals.

The *Implementation* chapter highlighted trade-offs between VMs and containers. Containers offer manageability but lack realism for complex infrastructures, while VMs are resource-intensive. Research could focus on hybrid approaches that combine the strengths of both, enabling modular and realistic scenario development. Similarly, addressing NixOS's limitations with proprietary packages, through solutions like hybrid systems or emulation layers, could enhance the framework's applicability[21].

Additionally, future work could connect the **Helical Build Framework** with educational frameworks like REWIRE[62] or ECSF[63]. By linking the four scenarios to specific cybersecurity roles, such as network security specialist for *Satellite Siege*, IoT security specialist for *Hotel Daemon*, secure coding specialist for *Patch War*, and forensic and reverse engineering analyst for *Zheng Hijack*, the framework can support the goals of these educational frameworks to train professionals with focused skills. This connection would improve the scenarios' usefulness, ensuring they meet the needs of different cybersecurity roles while supporting standard training objectives in the industry.

[62] https://rewire-project.eu/

[63] https://www.enisa.europa.eu/publications/european-cybersecurity-skills-framework-ecsf

9. Conclusion

This thesis explored cybersecurity scenarios as essential tools for training experts to address the increasing demand in this field. Existing solutions, such as TryHackMe[64] and Root-Me[65], are often proprietary, which limits customization and detailed feedback. Moreover, the literature lacks clear methodologies for designing cybersecurity scenarios. To fill this gap, this thesis proposes the **Helical Build Framework**, a methodology that emphasizes modular and reusable components for scenario design. This approach starts with a core idea and develops the scenario iteratively, like a helix, by repeatedly answering two key questions:

- *What steps allow the trainee to reach this stage of the scenario?*
- *What goal does this stage achieve in the scenario's design?*

Alongside the design methodology, this thesis offers recommendations for scenario implementation. It examines two virtualization methods: containers and virtual machines (VMs), each with distinct advantages and limitations. Containers are easier to manage, while VMs provide more realistic emulation, enabling complex exploits, such as those targeting the kernel. The choice of the base system is also discussed, with **NixOS** selected for its single configuration file and ability to switch setups easily, supporting modularity and reproducibility.

To validate the methodology and implementation recommendations, four scenarios were developed, each focusing on a specific cybersecurity area:

- **Satellite Siege**: The trainee hacks a satellite ground base to intercept spy satellite communications, focusing on networking skills.
- **Hotel Daemon**: The trainee locates and traps a criminal hacker in a hotel room by exploiting the hotel's internal network and IoT devices, emphasizing secure access for IoT systems.
- **Patch War**: The trainee infiltrates a game server to fix a vulnerability that allows cheating and unnaturally high scores, highlighting secure coding practices.
- **Zheng Hijack**: The trainee dismantles a botnet stealing company data through forensic and reverse engineering skills.

Finally, this thesis explored ways to evaluate the quality and effectiveness of the **Helical Build Framework**. It assessed the methodology's soundness and completeness, which refer to its ability to produce only correct scenarios and to create any correct scenario. The evaluation also included self-validation through the four developed scenarios. These efforts show that the methodology can create reliable, modular scenarios with reusable components, addressing the need for structured cybersecurity training environments.

[64] https://tryhackme.com
[65] https://www.root-me.org

References

[1] F. Cremer *et al.*, “Cyber risk and cybersecurity: a systematic review of data availability,” *The Geneva Papers on Risk and Insurance. Issues and Practice*, vol. 47, no. 3, pp. 698–736, 2022, doi: 10.1057/s41288-022-00266-6.

[2] “2024 ISC2 Cybersecurity Workforce Study.” Accessed: May 03, 2025. [Online]. Available: https://www.isc2.org/Insights/2024/10/ISC2-2024-Cybersecurity-Workforce-Study

[3] “Understanding the full cost of cyber security breaches,” *Computer Fraud & Security*, vol. 2020, no. 12, pp. 6–12, Dec. 2020, doi: 10.1016/S1361-3723(20)30127-5.

[4] “White paper.” Accessed: May 16, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=White_paper&oldid=1287145483

[5] L. Leung, B. Duhoux, A. Legay, and S. Kieffer, “Integrating Gamification, Recommender System, and Chatbot in Cybersecurity Training: A User-Centered Approach for Enhanced Engagement and Motivation,”

[6] C. Scherb, L. B. Heitz, F. Grimberg, H. Grieder, and M. Maurer, “A Serious Game for Simulating Cyberattacks to Teach Cybersecurity.” Accessed: Apr. 21, 2025. [Online]. Available: http://arxiv.org/abs/2305.03062

[7] J. Prümmer, T. van Steen, and B. van den Berg, “A systematic review of current cybersecurity training methods,” *Computers & Security*, vol. 136, p. 103585, Jan. 2024, doi: 10.1016/j.cose.2023.103585.

[8] “White Paper: Cyber Exercise Scenario Development.” Accessed: May 03, 2025. [Online]. Available: https://ecs-org.eu/?publications=white-paper-cyber-exercise-scenario-development

[9] N. Cercone, “Infinite monkey theorem - Wikipedia, the free encyclopedia.”

[10] E. Dolstra, M. de Jonge, and E. Visser, “Nix: A Safe and Policy-Free System for Software Deployment.”

[11] E. Dolstra, *The purely functional software deployment model*. S.l.: s.n., 2006.

[12] “NixOS/nixpkgs.” Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/NixOS/nixpkgs

[13] “NixOS Manual.” Accessed: Apr. 28, 2025. [Online]. Available: https://nixos.org/manual/nixos/stable/

[14] “Tell me a story: The effects that narratives exert on meaningful-engagement outcomes in antiphishing training,” *Computers & Security*, vol. 129, p. 103252, Jun. 2023, doi: 10.1016/j.cose.2023.103252.

[15] A. K. Lefor, K. Harada, H. Kawahira, and M. Mitsuishi, “The effect of simulator fidelity on procedure skill training: a literature review,” *International Journal of Medical Education*, vol. 11, pp. 97–106, May 2020, doi: 10.5116/ijme.5ea6.ae73.

[16] R. T. Hays, J. John W., P. Carolyn, , and E. Salas, “Flight Simulator Training Effectiveness: A Meta-Analysis,” *Military Psychology*, vol. 4, no. 2, pp. 63–74, Jun. 1992, doi: 10.1207/s15327876mp0402_1.

[17] M. Li, S. Ma, and Y. Shi, “Examining the effectiveness of gamification as a tool promoting teaching and learning in educational settings: a meta-analysis,” *Frontiers in Psychology*, vol. 14, Oct. 2023, doi: 10.3389/fpsyg.2023.1253549.

[18] “Revealing the theoretical basis of gamification: A systematic review and analysis of theory in research on gamification, serious games and game-based learning,” *Computers in Human Behavior*, vol. 125, p. 106963, Dec. 2021, doi: 10.1016/j.chb.2021.106963.

[19] “(PDF) Story-Based Learning: The Impact of Narrative on Learning Experiences and Outcomes,” *ResearchGate*, doi: 10.1007/978-3-540-69132-7_56.

[20] V. C. Muckler, “Exploring Suspension of Disbelief During Simulation-Based Learning,” *Clinical Simulation in Nursing*, vol. 13, no. 1, pp. 3–9, Jan. 2017, doi: 10.1016/j.ecns.2016.09.004.

[21] tobiasBora, “Different methods to run a non-nixos executable on Nixos.” Accessed: May 25, 2025. [Online]. Available: https://unix.stackexchange.com/q/522822

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**