MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Generator of Vulnerable Web Applications

Bachelor's Thesis

MAREK GELETA

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Generator of Vulnerable Web Applications

Bachelor's Thesis

MAREK GELETA

Advisor: doc. RNDr. Jan Vykopal, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2025

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Geleta

Advisor: doc. RNDr. Jan Vykopal, Ph.D.

Acknowledgements

I would like to express my sincere gratitude to my thesis consultant, doc. RNDr. Jan Vykopal, Ph.D., for his invaluable guidance, support, and expertise throughout the course of this thesis.

Funded by the European Union under Grant Agreement number 101087529. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Use of AI Tools During the preparation of this thesis, the following AI tools were utilized:

- **Grammarly**: For grammar checking and correcting language mistakes.
- **GitHub Copilot**: To accelerate code development and provide code suggestions.
- **ChatGPT**: For improving writing style, refining explanations, and suggesting structural improvements to the text.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Abstract

Capture-the-Flag competitions frequently employ the attack-defence format, but bringing that format into university courses is hindered by the labor of creating fresh vulnerable services—re-using past targets encourages solution sharing between students. This thesis introduces a generator that, from a brief text specification selecting vulnerability types and difficulty levels from a predefined catalogue, builds a containerised service ready for deployment into the FAUST attack-defense framework. The core business logic is fixed, yet the generator randomises API identifiers, interface theme, and the exact placement of each vulnerability, yielding a distinct exploit path for every requested flaw. It produces a runnable container image together with FAUST-compatible checker scripts, enabling instructors to publish a fully functioning, exploitable target without manual adjustment. A pilot classroom trial confirmed that the tool generated applications that deployed out of the box and behaved as intended, showing that automated diversification can make attack-defence CTF exercises practical for teaching while significantly cutting preparation effort.

Keywords

vulnerability generator, web application security, cybersecurity education, Capture The Flag, SQL injection, authorization flaws, FAUST Attack-Defence

Contents

1 Introduction **1**

2 Background **4**
2.1 CTF Formats 4
2.2 Related Work and Existing Solutions 5
2.3 FAUST CTF Framework 6
2.4 Technologies Used 10
2.5 Software Security Vulnerabilities 13

3 Design **18**
3.1 Requirements 18
3.2 Generator design 21

4 Implementation **24**
4.1 Generator core 24
4.2 Configuration generator 31
4.3 Baseline code template 32
4.4 Validation scripts 36
4.5 Automation 38

5 Evaluation and Testing **44**
5.1 Testing Procedure 44
5.2 Observations 45
5.3 Improvements 47

6 Conclusions and Future Work **48**

A Content of the attachment **51**

Bibliography **52**

List of Figures

2.1 Scoreboard from FAUST CTF 2024 [14] 8
2.2 A FAUST CTF architecture diagram illustrating interaction between the architecture components and players during a game. 11
3.1 High-level architecture diagram 21
4.1 User Interface Screenshots of the Application 34
4.2 CI/CD Job continuing into validation phase 41

1 Introduction

The landscape of digital threats continues to grow and evolve, with cybersecurity incidents affecting organizations of all sizes across all sectors. As digital infrastructure becomes increasingly more important to modern society, the sophistication and frequency of cyberattacks have grown in parallel. According to the European Union Agency for Cybersecurity (ENISA), the number of reported data compromises increased by 78% in 2023 compared to 2022, with this upward trend continuing into 2024. ENISA also observed a significant rise in the exploitation of vulnerabilities and reported that the impact of system and human errors more than tripled in 2023 [1]. This rapidly evolving threat landscape demands skilled cybersecurity professionals capable of defending against increasingly complex attacks.

Education in cybersecurity presents unique challenges that traditional pedagogical approaches struggle to address. While theoretical knowledge forms an essential foundation, practical skills development requires hands-on experience with real-world scenarios. Gamification has emerged as a particularly effective educational strategy in this domain. Capture The Flag (CTF) competitions provide dynamic, immersive, and challenging learning environments. Participants actively apply security concepts to identify and exploit vulnerabilities in controlled settings. These competitions transform abstract security principles into concrete problems, targeting the improvement of both technical skills and analytical thinking.

Universities, corporations, and cybersecurity communities worldwide have embraced CTF competitions as valuable educational tools. [2, 3] Academic institutions incorporate them into curricula to complement theoretical coursework, while companies utilize them for employee training and recruitment. Events such as DEF CON CTF [4] and picoCTF [5] draw thousands of participants each year, building strong and engaging learning ecosystems.

Among CTF formats, Attack-Defense (A/D) competitions stand out for their comprehensive approach to security education. Unlike jeopardy-style CTFs that focus primarily on exploitation, A/D competitions require participants to simultaneously defend their systems while attacking others'. This environment develops professional skills

often overlooked in other CTF formats: vulnerability patching, intrusion detection, teamwork, and strategic prioritization under time constraints, where participants must make real-time decisions about resource allocation between offensive and defensive operations.

Educational institutions face a significant challenge when incorporating CTF exercises into recurring courses: maintaining assessment integrity across terms. Reusing identical challenges enables students to share solutions from previous iterations, undermining the educational value of the exercise. Conversely, developing entirely new scenarios for each course iteration requires substantial resources and expertise. Instructors must balance the need for novel challenges against practical constraints of time and expertise, particularly when creating complex A/D scenarios that require multiple interconnected vulnerabilities and supporting infrastructure.

This thesis addresses these challenges through the development of a specialized generator for vulnerable web applications. The generator produces applications with consistent core functionality but randomized implementation details: different vulnerability types and locations, varied visual appearances, and randomized API endpoints. This approach preserves the pedagogical objectives while presenting students with unique technical challenges each time. The generator supports integrating SQL injection and authorization flaws, with configurable difficulty levels to accommodate different learning stages and educational objectives.

The generated applications are fully compatible with the FAUST Attack-Defense framework [6], an established platform for hosting A/D competitions. Each application includes essential components for deployment: containerized services, database initialization scripts, and checker scripts for automated functionality verification and flag placement. This integration significantly reduces the overhead associated with creating and deploying new A/D scenarios, enabling instructors to focus on educational aspects rather than technical implementation details.

The remainder of this thesis is organized as follows: Chapter 2 provides background information on CTF competitions, describes the FAUST Attack-Defense framework, reviews existing vulnerability generation tools, and covers essential technologies and vulnerabilities addressed in this work. Chapter 3 presents the design requirements

and architectural decisions that guided the development of the generator. Chapter 4 details the implementation of key components, the core structure of the generator, validation scripts, and the intended vulnerability details. Chapter 5 evaluates the generator through testing in a real educational environment and describes improvements made following user feedback. Finally, Chapter 6 summarizes the contributions of this work and outlines directions for future research.

2 Background

To provide context, the chapter first introduces Capture The Flag competitions and their most common formats, then surveys current automated vulnerability generation tools and their limitations, introduces the FAUST Attack-Defense architecture, and closes with a brief overview of the key technologies and security flaws used in the later implementation.

2.1 CTF Formats

Kucek et al. defines Capture the flag (CTF) as "a computer security competition with challenging exercises from different categories (e.g., network security, application security, mobile security, etc.). Typically, the goal of this game is to find a flag, i.e., the solution to the given problem (which is often solved through multiple steps or tasks within an exercise). A flag can be, for example, a string in the format `flag{sometext}` but also an image or another element defined by the CTF game designers." [7] The two most used formats in competitions are jeopardy-style and attack-defense (A/D).

Jeopardy-Style CTFs

Jeopardy-style CTFs are organized around independent challenges categorized by topic (e.g., cryptography, web, reverse engineering). Each challenge, once solved, reveals a unique flag that can be submitted for points. [8] An example of a web-based Jeopardy-style CTF challenge involves a web application with an administrator panel that is vulnerable to unauthorized access. Participants are tasked with identifying and exploiting a security flaw that allows them to bypass authentication mechanisms and access the administrator panel, where the flag is located.

Attack-Defense CTFs

In A/D CTFs, each team runs its own instance of intentionally vulnerable services (typically inside a virtual machine, or "Vulnbox"). Teams

must defend these services from exploitation while simultaneously attacking the services of other teams. Points are awarded for capturing flags from other teams, protecting your own, and maintaining service availability (SLA).

Because this format forces participants to handle realistic conditions – patching, monitoring, writing exploits, maintaining uptime–it is widely used in education and team training.

2.2 Related Work and Existing Solutions

Several tools are available to automate the generation of vulnerable applications and environments for cybersecurity training. They aim to facilitate the creation of realistic attack scenarios for education and assessment. However, each exhibits limitations in terms of flexibility, modern technology adoption, and integration with Attack-Defense (A/D) competition frameworks.

SiteGenerator One of the earliest attempts is the OWASP SiteGenerator project [9]. SiteGenerator was designed to dynamically generate websites containing a configurable set of vulnerabilities for benchmarking security scanners and training purposes. However, the project remains incomplete and has not been actively maintained since 2007. Additionally, its generated applications are primarily based on outdated technologies such as ASP.NET and PHP5. Vulnerability integration in SiteGenerator operates on a binary on-off basis – developers must explicitly include both vulnerable and secure code paths, letting users choose which version to deploy. This limits both the granularity of difficulty configuration and the variety of scenarios that can be generated automatically.

VWGen VWGen [10] shares similar limitations. Although it automates the generation of web applications with vulnerabilities like SQL injection and local file inclusion, it is an experimental tool without active development since 2017 that primarily targets older web stacks and requires manual definition of vulnerable code snippets. VWGen lacks support for modern web development frameworks and does

not provide fine-grained control over vulnerability difficulty or placement. Furthermore, neither SiteGenerator nor VWGen allows for easy integration with modern A/D competition platforms, requiring substantial manual adaptation to incorporate their outputs into structured competitions.

SecGen In contrast, SecGen [11] approaches the problem from a higher abstraction level by generating entire vulnerable virtual machine environments rather than individual applications. SecGen is primarily used to create randomized security scenarios for Capture The Flag (CTF) events and academic exercises. While powerful for infrastructure-level challenge generation, it does not focus on fine-grained control over web application vulnerabilities and lacks direct compatibility with established A/D platforms like the FAUST framework. Generated scenarios require additional effort to integrate into competition environments and lack modularity at the application level.

This thesis addresses these limitations by introducing a novel generator for vulnerable web applications built on modern technologies, specifically Node.js and Express. Unlike previous tools, the proposed generator allows granular specification of vulnerability types and difficulty levels, enabling educators to tailor scenarios to their learning objectives. Most importantly, the generated applications are fully compatible with the FAUST Attack-Defense framework, facilitating seamless deployment in A/D CTF competitions without requiring additional adaptation or manual integration efforts.

2.3 FAUST CTF Framework

The FAUST CTF framework [12] is an open-source system developed by the FAU Security Team [6], designed to facilitate attack-defense (A/D) style Capture The Flag competitions. Its architecture comprises of several components that cooperate in order to manage services, monitor their availability, handle flag placement and retrieval, and calculate scores as shown in Figure 2.2.

Vulnbox

Each participating team is provided with a virtual machine (VM) known as a *vulnbox*. This VM hosts a set of intentionally vulnerable services that are identical across all teams. Teams are responsible for defending their own Vulnboxes while attempting to exploit vulnerabilities in other teams' Vulnboxes to capture flags [12]. Usually, teams have superuser access to the Vulnbox, allowing for more efficient administration and creative ways of patching vulnerabilities and detecting vulnerabilities being exploited – for example, running a packet capture.

Services

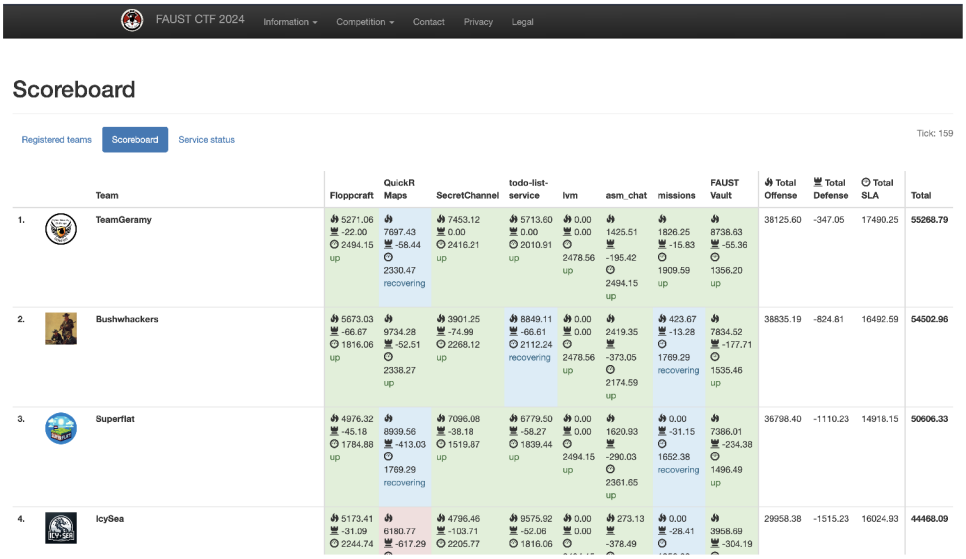
Services are applications running on the Vulnbox, deliberately designed with security flaws to be exploited during the competition. The services are commonly implemented in the form of Docker containers running on the vulnbox. Each service is accompanied by a checker script that verifies its functionality and availability [12, 13].

Checker Scripts (Checkerbots)

For each service, a corresponding *checker script*—often referred to as a *checkerbot*—is implemented in Python or Go. These scripts perform flag placement, service functionality validation, and flag retrieval. Checker scripts utilize the provided `checkerlib` library, which simplifies their development and ensures compatibility with the scoring system [13].

Gameserver (Controller)

The *Gameserver*, also known as the *controller*, orchestrates the competition by managing ticks, generating new flags, executing checker scripts, accepting flag submissions, calculating scores, and hosting a scoreboard. [12, 6]. Throughout the game, players can see the scores in real time on a scoreboard as show in Figure 2.1



FAUST CTF 2024 Information Competition Contact Privacy Legal

Scoreboard

Registered teams | Scoreboard | Service status

Tick: 159

	Team	Floppcraft	QuickR Maps	SecretChannel	todo-list-service	ivm	asm_chat	missions	FAUST Vault	Total Offense	Total Defense	Total SLA	Total
1.	TeamGeramy	5271.06 -22.00 2494.15 up	7697.43 -58.44 2330.47 recovering	7453.12 0.00 2416.21 up	5713.60 0.00 2010.91 up	0.00 0.00 2478.56 up	1425.51 -195.42 2494.15 up	1826.25 -15.83 1909.59 up	8738.63 -55.36 1356.20 up	38125.60	-347.05	17490.25	**55268.79**
2.	Bushwhackers	5673.03 -66.67 1816.06 up	9734.28 -52.51 2338.27 up	3901.25 -74.99 2268.12 up	8849.11 -66.61 2112.24 recovering	0.00 0.00 2478.56 up	2419.35 -373.05 2174.59 up	423.67 -13.28 1769.29 recovering	7834.52 -177.71 1535.46 up	38835.19	-824.81	16492.59	**54502.96**
3.	Superflat	4976.32 -45.18 1784.88 up	8939.56 -413.03 1769.29 recovering	7096.08 -38.18 1519.87 up	6779.50 -58.27 1839.44 up	0.00 0.00 2494.15 up	1620.93 -290.03 2361.65 up	0.00 -31.15 1652.38 recovering	7386.01 -234.38 1496.49 up	36798.40	-1110.23	14918.15	**50606.33**
4.	IcySea	5173.41 -31.09 2244.74	6180.77 -617.29	4796.46 -103.71 2205.77	9575.92 -52.06 1816.06	0.00 0.00	273.13 -378.49	0.00 -28.41	3958.69 -304.19	29958.38	-1515.23	16024.93	**44468.09**

Figure 2.1: Scoreboard from FAUST CTF 2024 [14]

Ticks

The competition is segmented into intervals called *ticks*. These intervals are usually in the range of 30 seconds to 5 minutes, set by the competition administrators. Each tick, the gameserver re-runs the checker scripts and updates the scoring. [12].

Flags

Flags are random strings following a predetermined format (e.g., `FAUST_[32 random characters]`). They are central to the Attack-Defense CTF format. Flags serve as concrete goals for attackers and defenders, creating a measurable competitive element. In the FAUST framework [12], flags follow a specific lifecycle: Flag Generation and Placement, Flag Stealing, and Flag Defense. All of the mentioned can be seen in Figure 2.2 except for Flag Defense, which would be hard to illustrate as successful flag defense results in other teams' inability to run an exploit successfully.

Flag IDs

Each flag is associated with a unique identifier known as a *flag ID*. This identifier references specific instances of data where flags are stored, for example, an identifier of a user who stored the flag in their private notes. The flag IDs facilitate interactions between the gameserver and participants, important for both service functionality validation and exploit development [13]. Flag IDs are public to all the teams as some vulnerabilities may not be exploitable without their knowledge. On the contrary a service may not provide flag IDs if their authors decide so. This is the case when the service's vulnerabilities are exploitable without arbitrary information. An example of this is an SQL injection vulnerability that allows users to gather all flags from the database without any additional information.

Flag Generation and Placement

At the beginning of each tick, the Gameserver generates unique flags for each team and service. The checkerbot script then inserts these flags into each team's service instance, storing them in predefined locations such as database records, text files, or application data structures. After the checkerbot inserts a flag into the service's data store, it saves the corresponding flag ID so it can be accessed by all the teams.

The insertion process usually involves:

1. Creating a regular user account in the service.
2. Storing the flag as private information (e.g., in a private message or note).
3. Recording the flag ID in the gameserver database.

Older flags are automatically invalidated after a predefined number of ticks, creating a rolling window of valid flags.

Flag Stealing

Teams exploit vulnerabilities in opponents' services to extract flags placed by the checkerbot. Every team is provided with flag IDs of other teams as for some vulnerabilities The Flag IDs are important

when crafting exploits. They provide a direct reference to specific data entries within the application. For instance, if a service has an authorization flaw that allows access to any user's data given their user ID, knowing the flag ID, which corresponds to a user ID, enables an attacker to retrieve the flag without proper authorization. A successful attack involves:

1. Identifying and exploiting a vulnerability in the target service.
2. Using the exploit to access the protected flag data.
3. Extracting the flag from the service.
4. Submitting the stolen flag to the Gameserver.

The Gameserver validates submitted flags and awards attack points if the flag is valid and was stolen from another team's service.

Flag Defense

Teams must also defend their own services by patching vulnerabilities to prevent flag theft. The checkerbot periodically verifies that:

1. The service is accessible and responding correctly.
2. Previously placed flags can still be retrieved using legitimate access methods.
3. New flags can be placed successfully.

A team earns defense points when their flags remain secure and their service maintains functionality. Points are deducted when flags are successfully stolen or when the service becomes unavailable or dysfunctional.

2.4 Technologies Used

This section presents an overview of the technologies applied in the development of the individual parts of the vulnerability generator described in Chapter 4. Each section also provides a brief explanation of the purpose each technology serves within the generator or the resulting vulnerable application.

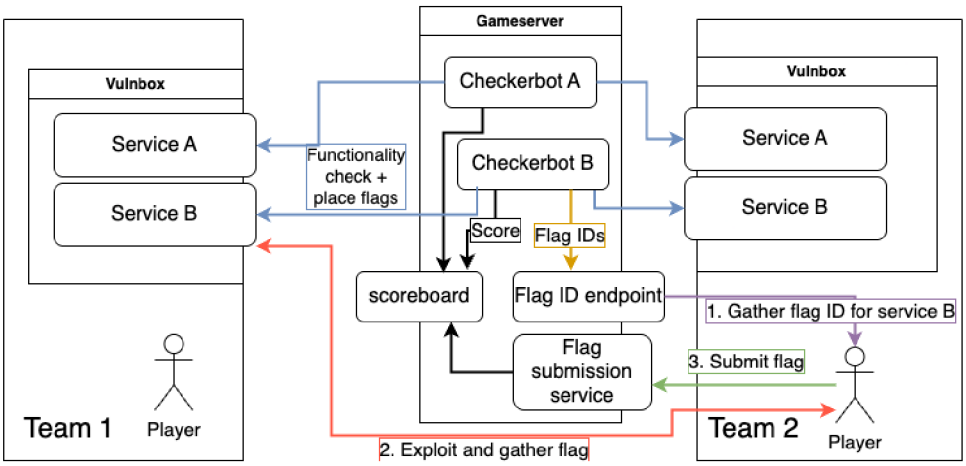


Figure 2.2: A FAUST CTF architecture diagram illustrating interaction between the architecture components and players during a game.

Core Libraries

Jinja2 Jinja2 is a templating engine that supports embedding control structures, variable substitutions, and expressions within text-based templates. It is primarily used in conjunction with the Python programming language. Template engines of this kind facilitate the automated generation of structured text, such as HTML or configuration files, based on predefined templates and variable input [15]. In the presented vulnerability generator, Jinja2 is used to generate randomised source code of the vulnerable application from a template.

SQLGlot SQLGlot is a software library for parsing, transforming, and generating SQL queries. It converts SQL statements into abstract syntax trees (ASTs), which can be programmatically modified and re-rendered in various SQL dialects. This functionality is relevant in contexts where applications need to interact with multiple database systems using different dialects of SQL [16]. This library is used to allow support for multiple SQL dialects in the generated vulnerable application, allowing for more variability in the generated application.

Express.js Express.js is a web application framework for the JavaScript programming language, designed to operate on the Node.js [17] run-

time environment. It provides a set of features for building web servers and APIs, including mechanisms for handling HTTP requests and responses, defining routing logic, and managing middleware functions. Express.js follows a modular approach, allowing developers to integrate third-party libraries and middlewares to extend its core functionality. While it is often used in the development of RESTful APIs and web applications, it does not impose specific architectural patterns, leaving design decisions to the developer. This flexibility makes it suitable for a range of applications, from simple web servers to more complex backend services [18]. The generated vulnerable application uses Express.js in its source code to handle HTTP requests, define routing, and manage middlewares for server-side functionality.

Data Formats

YAML YAML Ain't Markup Language is a human-readable data serialization standard designed for configuration files and data exchange. It supports the representation of hierarchical data structures and is commonly used to define configuration settings in a readable format. YAML files are often employed in automation pipelines and software configuration tasks [19]. The generator uses YAML configuration files for specifying details about the vulnerabilities that can be inserted into the resulting application.

Development and Testing Tools

SQLMap SQLMap is an open-source software tool used to automate the detection and exploitation of SQL injection vulnerabilities. It supports a range of techniques for identifying and verifying SQL injection flaws, and it can interact with various relational database management systems. SQLMap is used both in security testing and educational environments to demonstrate the impact of SQL injection vulnerabilities. One of its advanced features is the usage of tamper scripts that can tamper the SQL injection payload in order to evade an input filter or sanitization [20]. SQLMap is used in this work as a means to automatically validate the presence of SQLi vulnerabilities.

Infrastructure Components

Docker Docker is a platform for containerization that allows software and its dependencies to be packaged into isolated environments called containers. Containers provide a consistent execution environment across different systems by encapsulating application code and dependencies while sharing the host operating system kernel. This approach is widely used to improve reproducibility and isolation in software deployment [21]. The generated vulnerable application is provided with a Docker configuration that allows it to be containerised and deployed on the Vulnbox.

GitLab CI/CD GitLab CI/CD is a remote system for automating software build, test, and deployment processes using declarative configuration files. It supports the definition of pipelines, execution of jobs in defined stages, and integration with containerized environments. This tool is used to implement continuous integration and deployment practices in software development workflows [22]. GitLab CI/CD simplifies the use of the generator by running it in a remote environment, eliminating the need for local setup by the user.

2.5 Software Security Vulnerabilities

"A 'weakness' is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities." [23] A software vulnerability is an instance of one or more weaknesses in the design or implementation of software that could be exploited to compromise the confidentiality, integrity, or availability of a system. [24] These vulnerabilities arise from errors in code, design decisions, configurations, or operational practices that create opportunities for unauthorized access or manipulation of software components.

The Common Weakness Enumeration (CWE) provides a standardized classification system for software and hardware weaknesses. This framework enables the categorization, identification, and mitigation of vulnerabilities across different platforms and technologies [25].

Assigning CWE identifiers to vulnerability types allows for more precise identification and communication about the core issue and enables researchers and developers to observe trends in software weaknesses over time. This categorization facilitates the development of targeted mitigation strategies and supports the creation of comprehensive security standards. For instance, the projects like OWASP Top Ten, or CWE Top 25 utilize CWE classifications to group vulnerabilities by their underlying causes, rather their specific technical details, providing a more logical framework for identification and remediation guidance [26, 27].

Unexploitable Vulnerabilities Not all code weaknesses that appear to be vulnerabilities can be meaningfully exploited in practice. These "unexploitable vulnerabilities" may exist due to environmental constraints, compensating controls, or implementation details that prevent an attacker from constructing a viable attack path. In educational contexts, unexploitable vulnerabilities create particular challenges as they can frustrate learners who correctly identify a theoretical weakness but cannot demonstrate its impact. For vulnerability generators intended for educational use, it is important to ensure that the introduced flaws are exploitable – students should be able to confirm their findings through successful exploitation.

SQL Injection

SQL Injection (SQLi) vulnerabilities occur when untrusted user input is improperly handled by a web application and directly incorporated into SQL queries without adequate sanitization or validation. This allows attackers to alter the intended execution of SQL commands, leading to unauthorized access, data leakage, or even data destruction [28]. The CWE-89 "Improper Neutralization of Special Elements used in an SQL Command" [29] is commonly used with SQLi vulnerabilities.

Example of SQL Injection in Python

```python
def login(request):
    username = request.get("username")
```

```
password = request.get("password")

query = f"""SELECT * FROM users
WHERE username = '{username}'
AND password = '{password}';"""
database.execute(query)
...
```

Attacker's Input:

- **username**: `admin`
- **password**: `' OR '1'='1`

Resulting SQL Query:

```
SELECT * FROM users
WHERE username = 'admin'
AND password = '' OR '1'='1';
```

The injected condition `OR '1'='1'` always evaluates to `TRUE`, allowing the attacker to bypass authentication and gain unauthorized access.

Second-order SQL Injection

Second-order SQL injection occurs when malicious input is stored by the application and later used in a SQL query without proper re-validation or sanitization. Unlike traditional first-order SQLi, where the injection is immediately executed, second-order SQLi exploits rely on delayed execution, often triggered during a subsequent action or request. This makes detection more challenging, as the initial input may appear harmless. For example, an attacker may insert a payload into a user profile field, which is later used in a vulnerable backend query during login or reporting processes.

Relevance of SQL Injection in Cybersecurity Education

While the prevalence of SQLi vulnerabilities has declined in modern applications due to the widespread adoption of Object-Relational Mapping (ORM) frameworks and parameterized queries, SQLi remains

a critical component in cybersecurity education. Teaching SQLi provides foundational insights into input validation and filter bypasses, query manipulation, and the broader category of injection attacks. Injection attacks, including SQLi, remain highly relevant, as they continue to appear in the OWASP Top Ten Web Application Security Risks, highlighting the persistent threat posed by improperly handled user input [27].

Authorization Flaws

Authorization flaws occur when a system fails to properly enforce access controls, allowing users to perform actions or access resources that should be restricted. After examining various types of authorization vulnerabilities in the CWE framework, we have chosen to focus on CWE-863 "Incorrect Authorization" [30] for our implementation. While the

We selected CWE-863 as our primary focus because it represents a realistic scenario in real-world applications: developers typically attempt to implement authorization controls but fail to do so correctly. According to the CWE definition, this occurs when "the product performs an authorization check when an actor attempts to access a resource or perform an action, but it does not correctly perform the check" [30]. Unlike completely missing authorization checks (CWE-862 [31]), applications with CWE-863 flaws implement some form of access control logic that contains subtle implementation errors, such as failing to verify permissions in all execution paths, applying checks inconsistently across different functions, or improperly handling edge cases. These flaws are particularly insidious as they may not be apparent during normal application usage, but can be abused by attackers who can exploit the flawed logic.

Relevance in Cybersecurity Education

Authorization vulnerabilities continue to be widespread in modern applications, with "Broken Access Control" ranking first in the OWASP Top Ten Web Application Security Risks since 2021 [27]. These flaws often arise from complex business logic and can be challenging to identify through automated testing alone. Teaching proper autho-

rization control implementation prepares students to design secure applications where every access operation is properly verified.

3 Design

This chapter first states the generator's functional and non-functional requirements and then explains its architecture at two levels: the overall design and the structure of each component.

3.1 Requirements

The generator is meant to be mainly used by academic staff and CTF administrators when preparing a CTF competition or assignment; hence, we expect the user to have an essential experience and understanding of the technologies used.

Objectives

The primary objective of the design process was to develop an architecture that satisfies the following requirements.

Easy to use For the tool to be easy to use, it has to require minimal input from the user while providing the maximum outcome possible using that input. The input is expected to be in human-readable text format, such as "1:easy:sqli" – meaning the user wants to generate an application with one easily exploitable SQL injection vulnerability. It should also allow using more advanced features for in-depth customization. This feature should be strictly optional, as the tool has to be able to function only with minimal configuration from the user.

Platform independent Platform independence refers to a tool's ability to operate seamlessly across various modern CPU architectures and operating systems without needing any source code alterations. One effective strategy to achieve this is using containerization technologies such as Docker [21] and Podman [32]. These solutions not only simplify the development and deployment but also enhance user experience, as the containerized environment eliminates the necessity for users to install additional software on their local machines.

Furthermore, by hosting the containerized tool in an external remote environment (configurable Gitlab CI/CD), users can engage

with an online service dedicated to executing the tool. This approach dramatically improves the platform independence, allowing users to access and utilize the tool from any device, while also enhancing the overall usability of the system. The combination of these technologies ensures a flexible, platform-independent, and user-friendly experience.

Capable of generating fully functional and intentionally vulnerable web applications The primary function of the tool is to generate web applications that contain specific security vulnerabilities. These vulnerabilities must be deliberately embedded in a controlled manner. In addition, the generator must include a validation mechanism to ensure that the created applications function correctly, contain the intended vulnerabilities, and allow for their successful exploitation.

This involves generating test scripts that check for vulnerabilities and confirm the application's expected behaviour. The validation process should be reliable, automated, and capable of detecting both functional errors and missing or unexploitable vulnerabilities in the expected locations. The generated application should also mimic real-world scenarios to provide a realistic training environment.

Compatibility with FAUST AD Service Format The generated application should be a plug-and-play FAUST A/D service that does not require any additional code or configuration apart from the expected A/D CTF setup configuration overhead. This means that the generated service has to be deployable using Docker and contain a checkerbot [13] responsible for healthchecks and flag distribution.

Reusable and randomized The main purpose of the tool is to be utilized in an educational setting for exams, student competitions, or other types of cybersecurity training. To ensure fairness in these exercises, it is important that the assignments vary each term. The tool operator should be able to generate a relatively large number of applications that differ in vulnerabilities, database schema, code structures, variable names, and API endpoint names. By doing so, the tool can produce unique but functionally similar applications for

different use cases, preventing students or competitors from relying on predefined solutions.

Functional Requirements

1. The tool must generate web applications vulnerable to:
   - SQL Injection
   - Authorization flaws
2. The tool must support configuration parameters that define:
   - Type and number of vulnerabilities
   - Complexity or security level of the application
3. The generated applications must be containerized using Docker or Podman.
4. The generator must randomize:
   - Variable names and values
   - Database schemas and SQL dialect used
   - Endpoint paths
5. The tool must provide a validation script that:
   - Confirms that vulnerabilities are exploitable
   - Ensures core application features are working
6. The generator must support exporting the generated application in a format compatible with FAUST AD Service.

Non-functional Requirements

1. The generator should be operable on major operating systems (Linux, Windows, macOS) and CPU architectures (x64, arm64).
2. The tool should be extensible, allowing the addition of new vulnerability types or templates in the future.

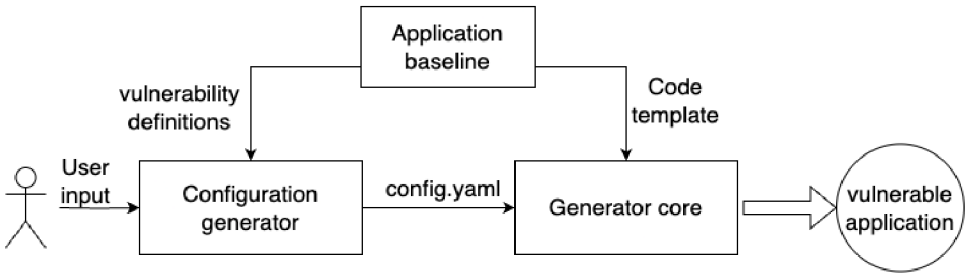


Figure 3.1: High-level architecture diagram

3. Generation and validation should complete in a reasonable time (less than 5 minutes for basic configurations).

4. Documentation and usage instructions must be provided.

5. The generated application should be lightweight and not require external proprietary software or services.

3.2 Generator design

The design we chose satisfies all requirements and consists of three main interconnected parts (shown in Figure 3.1) that work in a chain to produce the requested vulnerable web application.

Generator core

The generator core, written in Python, serves as the central component responsible for processing a configuration file in a predefined YAML format. This configuration defines key parameters of the requested service, including the database management system to be used, database table names, frontend theme, and specific vulnerability characteristics. These parameters are used in combination with a baseline code template for the vulnerable application. Based on this input, the generator core assembles the final output. As part of this process, it transforms the requested SQL queries into intentionally vulnerable forms. By default, the generated queries are in the form of prepared statements [33] that are not vulnerable to SQLi. Trans-

formation into the vulnerable form first includes transpilation[1] of the query to the selected SQL dialect. After the query is transpiled, the SQLi vulnerability is introduced by using direct string concatenation of user-supplied variables into the SQL query instead of the prepared statement. The generator core also outputs a FAUST checkerbot script and validation scripts to test the functionality and vulnerability of the generated application.

Application Baseline

The application baseline contains a code template of the vulnerable application with specific spots pre-marked where vulnerabilities or specific code blocks can be inserted. This way, the generator core can modify the template dynamically based on the provided configuration. Another part of the baseline is a configuration file with vulnerability definitions.

The codebase handles the following functionality: user authentication, authorized data operations, private text notes creation, sharing, and retrieval. Registered users can view a list of all their notes and their content. They can also use the sharing feature to make the selected notes accessible to other users of their choice.

This functionality was chosen as it is simple for the student to understand in a relatively short period of time, which is necessary in a typically timeboxed scenario of CTFs, trainings, or exams. The selected functionality also allows for efficiently storing CTF flags in the note content and using note identifiers as flag IDs. Most importantly, the authorized data retrieval functionality: sharing and viewing content of personal or shared notes allows the presence of authorization flaws, and the overall feature design allows for a possible SQL injection in each functionality, as all of it is designed to be database-centric.

The template is built to be flexible and easy to modify and extend, making it simple to adapt to different database systems, frontend styles, and vulnerability types.

We have been following Squarcina's A/D Service Development Guidelines [34] to make the generated application enjoyable to exploit

1. In this setting, transpilation denotes the automatic translation of a query from one SQL dialect to another (e.g., PostgreSQL → MySQL) while preserving its semantics.

and patch. The guidelines also helped us design the generated application to be resilient against different variations of Denial of Service attacks. These attacks happen both intentionally and unintentionally during an A/D CTF, and they are often not preventable by the rules of the competition. The typical unintentional scenario is a high load caused by a large number of teams exploiting a single service. Or an inefficient exploit that consumes a significant part of the resources available to the CTF service.

The generated application can also be used outside of the A/D environment, for example, as a standalone challenge in a typical jeopardy-style CTF.

Configuration generator

Since the configuration of the generator core requires the specification of numerous interrelated parameters, we introduce a layer responsible for creating the configuration file for the generator core from user input. It works by accessing a configuration file with vulnerability definitions for a given Application Baseline. The configuration generator can also list the available vulnerabilities, descriptions, and unique identifiers.

The configuration generator takes the following input:

- Vulnerability specification:
  - Number, type, and difficulty of the vulnerabilities
  - Unique identifier of a vulnerability
- DBMS specification (optional)
- Stylesheet specification (optional)

After processing the user input, the generator determines a set of vulnerabilities to be used. It selects the optimal configuration and DBMS based on the requirements of all vulnerabilities, as some vulnerabilities may require specific conditions in order to be exploitable. The output of this module is a configuration file that can be directly supplied to the generator core. An overview of this architecture is presented in Figure 3.1.

4 Implementation

This chapter describes how the vulnerable service generator is implemented. It first covers the core functionality, illustrating how vulnerabilities are dynamically inserted into an existing application template, and explains the technical details behind the vulnerabilities and associated hardening mechanisms. Secondly, it introduces a configuration generator that translates user input into ready-to-use configuration files. Next, it describes the baseline code structure and technology stack of the generated vulnerable application. The chapter then discusses the validation scripts produced alongside the application. Finally, it explains how the entire generation-to-deployment process is integrated into an automated workflow. The chapter concludes with an estimation of how many distinct applications the system can generate, considering combinations of vulnerabilities, hardening strategies, and randomization of visual and structural elements.

4.1 Generator core

The core functionality of the generator is implemented in Python, which leverages the flexibility of Jinja2 for template processing and code generation. The implementation employs an object-oriented approach through the central `VulnerableQueryGenerator` class that orchestrates the entire generation process.

Configuration and vulnerability processing

The generator begins by loading the vulnerability configuration from a YAML file created by the configuration generator. This configuration specifies the database type, table names, and most importantly, the intended vulnerability locations. The generator extracts details about which components should contain vulnerabilities.

SQL transpilation

One of the features of the generator is its ability to support multiple database management systems. The implementation utilizes the

```
// Secure version using a prepared statement
await query("SELECT * FROM users WHERE username = ?",
            [username]);

// Transformed vulnerable version using
// string concatenation
await query("SELECT * FROM users WHERE username = '" +
            username + "'");
```

Listing 1: Comparison of secure (parameterized) and insecure (concatenated) SQL queries

`SQLGlot` library (Section 2.4) to transpile schemas between different SQL dialects, ensuring that the generated database schema is compatible with the selected database system. The dialects supported are SQLite, PostgreSQL, and MySQL. The original schemas and queries are written in MySQL and automatically transpiled into the other dialects when necessary.

Query transformation

The implementation transforms secure queries into vulnerable ones only at specified locations, applying these changes to the designated parameters while preserving the functionality of the application. A secure prepared statement [33] can be transformed into a vulnerable version using string concatenation, as illustrated in Listing 1.

SQL Injection Hardening Mechanisms

The generator implements a hardening mechanism that modifies the difficulty of SQL injection vulnerabilities by adding input filtering code as an ExpressJS middleware processing all relevant user input from an HTTP request. This middleware detects and blocks malicious input, but it also contains intentional weaknesses that can be bypassed through specific techniques listed below.

1. **Backslash escape filter bypass [hard bypass difficulty]** This filter blocks single quotes in user input. The filter can be bypassed

using backslash characters to escape quote delimiters. This technique works specifically in scenarios with multiple sequential parameters, such as:

```
SELECT * FROM users WHERE
username='userinput1' AND password='userinput2'
```

An attacker can inject `malicious\` as `userinput1`. This causes the `'AND password='` portion to be interpreted as part of the first string literal. The `userinput2` is then executed as SQL syntax rather than treated as a string parameter, for example:

```
SELECT * FROM users WHERE
username='malicious\' AND password=' OR 1=1--'
```

The database treats everything between the first quote and the quote after `password=` as a single string. This allows arbitrary SQL injection in the second parameter.

However, this behavior is not consistent across the supported DBMS.

According to the SQLite documentation, "C-style escapes using the backslash character are not supported because they are not standard SQL" [35]. Therefore, SQLite does not support backslash escaping in string literals. PostgreSQL supports backslash escapes only when using an explicit escape string syntax (E'...') [36]. In contrast, MySQL supports backslash escaping by default, as noted in the MySQL reference manual [37].

Given that PostgreSQL requires a non-default syntax to enable this behavior, we considered this an unrealistic scenario within the context of the vulnerable web application and its expected functionality. As a result, we chose to support only the MySQL DBMS for this bypass mechanism. This resulted in limited DBMS variability on this hardening mechanism in order to provide a more realistic attack scenario.

2. **Recursive filter bypass [medium bypass difficulty]** This filter checks for SQL keywords but only examines one level of input recursion. Attackers can bypass it by wrapping parameters in

```yaml
sqli:
  filter_enabled: true
  filter_patterns: ["SELECT", "UNION","  '","  --"]
  array_bypass: true
  uppercase_bypass: true
```

Listing 2: Filter YAML configuration example

arrays. The server processes these array parameters while the detection mechanism fails to identify the payload inside the array structure. Specifically, JavaScript's implicit type coercion converts array parameters to strings when concatenated, resulting in the payload bypassing keyword checks. For instance, the expression `'cc' + ['aa','bb']` evaluates to `'ccaa,bb'`, as arrays are implicitly joined into comma-separated strings [38]. Attackers leverage this behavior to evade detection mechanisms designed to inspect string inputs directly, thus bypassing the filter.

3. **Uppercase filter bypass [medium bypass difficulty]** This filter blocks SQL keywords in uppercase form only. Attackers can use lowercase or mixed-case variations such as `select` or `SeLeCt` instead of `SELECT` to bypass the filter.

   These methods can also be combined, resulting in unique and more difficult-to-exploit scenarios.

The filter configuration is controlled through the YAML configuration file, which, if needed, is automatically created by the configuration generator when vulnerabilities of higher difficulty are requested. A configuration example is shown in Listing 2.

This configuration creates middleware that intercepts and filters requests before they reach vulnerable endpoints.

SQL injection Hardening selection

The configuration generator conducts hardening selection during vulnerability specification processing. This selection determines appropriate SQL injection protection mechanisms based on the requested difficulty level and database compatibility requirements. A hardening

can also be used as a means to raise the difficulty level of a vulnerability with a lower difficulty. If enough vulnerabilities with the requested difficulty are available, the generator may decide not to use any hardening mechanism.

The system uses a compatibility-driven selection algorithm rather than a simple one-to-one vulnerability mapping. The algorithm follows these steps:

1. Filters vulnerabilities according to the requested type and difficulty level.
2. Identifies hardening mechanisms compatible with the requested difficulty.
3. Creates groups of vulnerabilities based on hardening option compatibility.
4. Chooses an appropriate group using compatibility rules and randomization.
5. Verifies database system compatibility across all selected components.
6. Samples the required number of vulnerabilities from the compatible set.

The output of this algorithm is a part of YAML configuration with a set of vulnerabilities and the optionally selected hardening strategy as shown in Listing 3.

Authorization vulnerabilities

The generator incorporates multiple authorization vulnerability types that exemplify common access control flaws. Each vulnerability represents a specific implementation error:

1. **No return on failed response:** This vulnerability occurs when middleware sends an error response without terminating request processing. The code sends a 404 status code but lacks a `return` statement before `next()` is called. Request execution continues despite the error condition, and protected resources become accessible.

```yaml
sqli:
  filter_enabled: true
  filter_patterns: ["SELECT", "'", " -- ", "/*", "*/", "UNION"]
  array_bypass: true
  uppercase_bypass: false
vulns:
- component: get_user_for_sharing
  difficulty: medium
  id: sqli_share_user
  name: error-based SQLi in get_user_for_sharing
  parameters:
  - shareWithUsername
  type: sqli
```

Listing 3: Example YAML output from SQLi hardening selection algorithm

2. **Incorrect parameter order:** This vulnerability occurs when application code checks request parameters in an improper sequence. The code examines `req.body` parameters before `req.params`. Attackers can override URL parameters by submitting HTTP body parameters alongside legitimate URL parameters. Example code with this behavior is shown in Listing 4.

   If an adversary submits a request to the endpoint `/post/1`, the framework correctly dispatches the call to the handler responsible for resource 1. Nevertheless, by embedding an HTTP body content such as `itemId=42`, the handler selects the value 42 from the request body in preference to the URL parameter. The application therefore retrieves and authorises access to post 42, disclosing data to which the requester is not entitled (post with id 1).

3. **Wrong parameter name checks:** In this implementation, middleware examines incorrectly named parameters (`itemId` instead of `id` or vice versa). The parameter name mismatch results in null checks that always pass, which circumvents authorization checks for specific endpoints.

```
const id = req?.body?.itemId ?? req.params?.id;
if (!canAccess(id)){
    return res.send(403)
}
...
```

Listing 4: Incorrect parameter precedence: body parameters overriding URL parameters

4. **Default allow on empty username:** This flaw stems from incomplete authentication validation. The code verifies parameter existence but not user authentication state. When a username is empty or null, the middleware permits the request without proper authorization verification.

These vulnerabilities are implemented as alternative middleware modules selected according to the configuration file. The generator replaces secure authentication middleware with one of these flawed implementations. Each middleware file represents a distinct vulnerability variant. The middleware files are stored in the directory named `/vuln_code/auth_middleware/`.

Authorization flaws differ from SQL injections as they typically involve logic errors rather than database query manipulation. The provided Proof of Concept (PoC) scripts demonstrate methods to circumvent these controls and access restricted resources.

Template rendering

The final step in the generation process is applying the Jinja2 [15] templates to create the complete application. The generator walks through all files in the template directory, processing any file with the `.j2` extension using Jinja2 templating. Files without this extension are copied directly to maintain directory structure and non-template assets.

4.2 Configuration generator

The configuration generator, implemented in `generate_vuln_conf.py`, converts user inputs into configuration files required by the generator core.

Command-line interface

The configuration generator accepts command-line arguments that specify the number, difficulty, and type of vulnerabilities to include or a specific identification of a requested vulnerability.
Example usage:

```
python generate_vuln_conf.py 1:easy:sqli 1:medium:auth
python generate_vuln_conf.py vuln:sqli_notes_param
```

This approach removes the need for users to interact directly with the configuration file format of the generator core.

Vulnerability definitions

The configuration generator relies on the presence of a configuration file named `vulns.yaml`, which contains a structured list of predefined vulnerabilities, hardening methods, and stylesheets. This file was manually created and can be easily extended, allowing configurations compatible with the baseline application template to be added or modified as needed. The configuration generator uses this file to determine which vulnerabilities can be selected and supplied into the resulting configuration file for the generator core.

Vulnerability selection logic

This generator implements a selection algorithm for configuring vulnerabilities based on user specifications. It filters available vulnerabilities by type and difficulty, and then groups them based on their compatibility with various hardening mechanisms, ensuring compatibility among the selected vulnerabilities.

DBMS compatibility handling

The configuration generator determines compatible database systems for the selected vulnerabilities by analyzing the DBMS requirements for each vulnerability and hardening mechanism, then computing the intersection of all compatibility sets. This process identifies database systems that can support all selected components.

Configuration output

The final output is a structured YAML configuration file containing specifications for the database system, table names, stylesheet, SQL injection hardening mechanisms, and the selected vulnerabilities, as shown in Listing 5.

4.3 Baseline code template

The baseline code template, located in the `test_service/` directory, functions as the foundation for all generated applications. This template implements a web application with features designed to demonstrate security vulnerabilities in a contextual environment.

Technology stack

The baseline application is built using modern web technologies:

- Backend: Node.js with Express.js framework
- Database: Compatible with SQLite, PostgreSQL, and MySQL
- Frontend: HTML, CSS, and JavaScript with templating support
- Authentication: Session-based with optional middleware configurations

This technology stack was chosen for its widespread use in web development and for being easy for the CTF players to understand.

```yaml
config:
  database: mysql
service_config:
  stylesheet_name: modern.css
  tables:
    items_table: items
    shared_items_table: shared_items
    users_table: users
sqli:
  array_bypass: false
  filter_enabled: true
  filter_patterns:
  - ''''
  - ;
  - /*
  - '*/'
  uppercase_bypass: false
vulns:
- component: check_shared_item_access
  difficulty: easy
  id: sqli_shared_access_id
  name: SQLi (auth bypass) via id in check_shared_item_access
  parameters:
  - id
  type: sqli
- component: auth_middleware
  difficulty: medium
  id: auth_bypass_incorrect_order
  method: incorrect_order
  name: Authentication bypass - incorrect order of using arguments
  type: auth
```

Listing 5: Example YAML configuration file generated by the configuration generator

Figure 4.1: User Interface Screenshots of the Application

Application structure

The baseline application implements a note-sharing service with the following features:

- User registration and authentication
- Creation and retrieval of personal notes
- Sharing notes with other users
- Viewing shared notes from other users

The application structure supports the systematic integration of security vulnerabilities within a coherent and realistic usage scenario. The user interface corresponding to this structure is shown in Figure 4.1.

Database schema

The baseline template defines a database schema that includes tables for users, items, and shared items. The schema is designed to be compatible with multiple database systems while supporting the core functionality required for demonstrating SQL injection vulnerabilities.

```
// --- Template version (before transformation) ---
const { username, password } = req.body;
const row = await {{
  query(
  "SELECT * FROM users WHERE username = ? AND password = ?",
    placeholder_vars=["username", "password"],
    query_id="login_check_user")
}}

// --- Transformed version (vulnerable to SQLi) ---
const { username, password } = req.body;
const row = await
  query(
  "SELECT * FROM users WHERE username = '" +
  username + "' AND password = ?", [password]
);
```

Listing 6: Template marker and its transformed (vulnerable) version

Template markers

Throughout the baseline code, special template markers with function calls indicate where the generator should inject code or modify existing code. These markers use Jinja2 syntax to invoke the query transformation function from the generator core, allowing for precise control over where vulnerabilities are introduced.

In the example shown in Listing 6, the template marker is the `{{ query(...) }}` block, which uses Jinja2 templating syntax `{{}}` to specify the structure of the query, the JavaScript variables to use in the query, and the `query_id` which uniquely identifies the query enabling us to mark it as vulnerable as.
In a configuration where the `username` parameter is vulnerable to SQLi, the resulting code would look like shown in Listing 6.

API endpoints

The baseline application defines RESTful API [39] endpoints that implement core functionality, including user registration, authentication,

and note operations. The generator selects endpoint URLs randomly from a predefined wordlist rather than using fixed paths.

This randomization creates variability between generated applications. Each instance receives unique API paths that differ from those of other generated instances. The wordlist can be extended by administrators who wish to add new path name options for each functionality type. All application components, including verification scripts and proof of concept exploits, receive these randomized endpoints and must adapt their requests accordingly. This prevents students with prior knowledge of other generated services from relying on hard-coded paths during exploitation attempts.

Stylesheets

The baseline contains three different CSS stylesheets to visually distinguish the generated applications. All three stylesheets are being applied to the same underlying HTML code structure, but create different visual appearances. This design allows the generated applications to have distinct looks without requiring changes to the functional HTML structure.

The available themes include:

- **90s**: Retro design with bright colors and animated effects
- **Modern**: Clean minimal interface with subtle shadows
- **Cyber**: Dark theme with neon text and terminal aesthetics

The stylesheet selection is determined during the generation process, either randomly if not specified or based on user preference through a command line argument. This visual differentiation serves several purposes in educational settings: it helps distinguish between different exercises, reduces student recognition of previously seen applications, and increases the perceived variety of generated services.

4.4 Validation scripts

The validation scripts, located in `test_service/` under `checker` and `poc` directories, evaluate both functionality and exploitability of gener-

ated applications. These scripts verify that generated services operate correctly within a Capture The Flag environment.

Checker scripts

The `checker/basic_checker.py` script tests the functional correctness of generated applications. It performs tests of application features from user registration to note sharing. This script is used in the Gitlab CI/CD pipeline to verify the functionality of the application. The checkerbot script in `checker/checker.py` implements the FAUST A/D `checkerlib` interface [13]. This interface allows the placement and verification of flags within the application and dynamically tests the functionality.

We have decided to duplicate the testing functionality of the FAUST checkerbot into the `basic_checker` script, as the `checkerlib` library, on which the checkerbot relies, is not available in the testing environment. The checkerbot also requires a connection to a running FAUST server to test the service's functionality successfully. Mocking this library would be possible, but it would require significantly more effort to implement compared to a rewrite into a simple Python script. The `basic_checker` script is designed to be run locally, without the need to connect to the FAUST server. Depending on the user's preference, it can be run in a Docker container or directly on the host machine. This also makes implementing manual changes in the application code easier because it allows testing the generated application locally before deploying it into a production environment.

Proof of Concept (PoC) scripts

The templates of PoC scripts in `test_service/poc/` show exploitation methods for the generated vulnerabilities. When a new vulnerable application is generated, these templates are used to generate PoC scripts that will be able to demonstrate and therefore verify the intended vulnerabilities in the generated application. These scripts serve two purposes – they validate vulnerability presence and function as educational material for the students who did not manage to find the solution during the exercise.

SQL Injection

The SQL injection PoC script (`poc/sqli.py`) creates HTTP request files with pre-marked injection points according to the vulnerability configuration. These request files are then used with the SQLMap tool to verify the exploitability of the vulnerabilities. Depending on the configuration, this script also incorporates the use of SQLMap tamper scripts to bypass an input filter and the use of the array bypass technique mentioned in Section 4.1. The output of this script is a markdown file containing the HTTP request along with the payload used by SQLMap in order to exploit the injection.

At the current state, this script can verify the exploitability of first-order injections in HTTP POST parameters and HTTP path parts. It can also apply measures to bypass the input filtering for the Recursive and Uppercase filter bypasses described in Section 4.1. The support for second-order SQLi and bypassing the Backslash escape filter is planned as future work in Chapter 6.

Authorization flaws

Unlike the SQLi, the authorization flaw PoCs are mostly static. They usually require only minor changes after generation, such as updating HTTP path names. PoC for each authorization vulnerability contains a set of helper functions that manage making HTTP requests to the application. The PoCs then use these functions to exploit the vulnerability. A snippet of such PoC is shown in Listing 7.

4.5 Automation

The infrastructure, centered around the GitLab CI/CD [22] pipeline, automates the process of generating, building, deploying, and testing the vulnerable applications.

CI/CD pipeline structure

The GitLab CI pipeline consists of a main `generate` job that performs four logical phases operating in sequence. This design choice of having four logical phases within a single CI stage rather than separate stages

```
...
# Create sessions for each user
session1 = requests.Session()
session2 = requests.Session()
session3 = requests.Session()

# Register and login first user
...

# Share the item with the second user
share_item(session1, item_id, user2)

# Register and login third user
user3 = generate_random_string()
password3 = generate_random_string()
register(session3, user3, password3)
login(session3, user3, password3)

# Use the "no_return" vulnerability to
# share the item with the third user
share_item(session3, item_id, user3)

# Verify that the third user has the shared item
shared_items = get_shared_items(session3)
assert any(item["id"] == item_id for item in shared_items),
       "Shared item not found for third user"
...
```

Listing 7: Snippet from `no_return.py.j2` with example usage of the helper functions

stems from several important considerations: the tight sequential dependency between phases where each phase requires immediate outputs from the previous one, shared environment requirements across all phases, the need for atomic success or failure of the entire process, management and communication with ephemeral Docker resources that exist only during job execution, and simplified pipeline logic for comprehensive error handling and maintenance. The pipeline also includes a documentation job (`push_available_vulns`) that automatically executes whenever code is pushed to the repository, generating and updating a list of all available vulnerability configurations (`available_vulns.md` for users to reference.

Configuration Generation Phase This initial phase processes user-specified vulnerability parameters supplied through the `VULN_SPECS` environment variable. The content of this variable is directly used as the input for the configuration generator to create a structured YAML specification that guides the subsequent generation process. The configuration defines which vulnerabilities to include, their difficulty levels, and any special requirements such as database compatibility.

Service Generation Phase In this phase, the pipeline takes the configuration file and transforms it into a complete application codebase. The generator core parses the template files and injects vulnerabilities at predefined locations. This process includes transpiling database schemas, randomizing endpoint names, and implementing the specified vulnerabilities according to the configuration created in the previous step.

Validation Phase The validation phase builds the application within Docker containers and subjects it to automated testing. The pipeline executes both basic functionality tests `basic_checker.py` and targeted vulnerability checks located in `generated_service/poc`, failing if any critical issues are detected. As shown in Figure 4.2, this phase follows the service generation phase and transitions directly into building the Docker containers within the CI/CD pipeline.

CYBERSEC / theses / geleta-vuln-generator / Jobs / #720910

Search visible log output

```
94  Downloading argparse-1.4.0-py2.py3-none-any.whl (23 kB)
95  Downloading requests-2.32.3-py3-none-any.whl (64 kB)
96  Downloading certifi-2025.1.31-py3-none-any.whl (166 kB)
97  Downloading charset_normalizer-3.4.1-cp312-cp312-musllinux_1_2_x86_64.whl (146 kB)
98  Downloading idna-3.10-py3-none-any.whl (70 kB)
99  Downloading MarkupSafe-3.0.2-cp312-cp312-musllinux_1_2_x86_64.whl (23 kB)
100 Downloading urllib3-2.3.0-py3-none-any.whl (128 kB)
101 Installing collected packages: argparse, urllib3, sqlglot, pyyaml, MarkupSafe, idna, charset-normalizer,
    certifi, requests, jinja2
102 Successfully installed MarkupSafe-3.0.2 argparse-1.4.0 certifi-2025.1.31 charset-normalizer-3.4.1 idna-
    3.10 jinja2-3.1.6 pyyaml-6.0.2 requests-2.32.3 sqlglot-26.9.0 urllib3-2.3.0
103 $ python generate_vuln_conf.py $VULN_SPECS | tee /tmp/generate_output.txt
104 Selected vulnerabilities:
105 - error-based SQLi in get_user_for_sharing (SQLI, Easy)
106 - Authentication bypass - incorrect order of using arguments (AUTH, Medium)
107 $ python generate.py
108 Successfully generated vulnerable service code in /builds/cybersec/theses/geleta-vuln-generator/generate
    d_service
109 $ BASE_DIR=$(pwd)
110 $ cd generated_service/src
111 $ docker compose up -d --build
112 time="2025-03-08T16:19:53Z" level=warning msg="/builds/cybersec/theses/geleta-vuln-generator/generated_s
    ervice/src/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it
    to avoid potential confusion"
113 Compose now can delegate build to bake for better performances
114 Just set COMPOSE_BAKE=true
115 #0 building with "default" instance using docker driver
116 #1 [app internal] load build definition from Dockerfile
117 #1 transferring dockerfile: 243B done
118 #1 DONE 0.1s
119 #2 [app internal] load metadata for docker.io/library/node:18.18.0-slim
120 #2 DONE 1.5s
121 #3 [app internal] load .dockerignore
122 #3 transferring context: 2B done
123 #3 DONE 0.1s
124 #4 [app internal] load build context
```

Duration: 1 minute 28 seconds
Finished: 2 months ago
Queued: 0 seconds
Timeout: 1h (from project)
Runner: #707 (u6UHNjtB)
Source: Web
Tags: docker

Commit d590a11d

Pipeline #421751 Passed for main

generate

Related jobs

→ generate

Figure 4.2: CI/CD Job continuing into validation phase

Deployment Preparation Phase The final stage prepares the service for deployment in an A/D CTF environment. It creates Ansible configurations for deploying the service on the FAUST infrastructure and commits the generated service to a new branch in the repository, making it easy to access and deploy the service.

User interaction

Users interact with the automation system through the GitLab web interface or CLI, where they can start new pipelines, monitor progress, access generated code, and deploy services to their CTF environment. This approach removes the need for users to have detailed knowledge of the underlying generation process.

Estimating Combinatorial Diversity

In this section, we provide an estimate of how many distinct application configurations the generator can produce when combining two

SQL injection vulnerabilities (of any difficulty) and one authorization flaw (also of any difficulty). First, we calculate this estimate without considering additional randomization options, such as endpoint names and visual themes, to establish a baseline. Subsequently, we extend the calculation to include these randomization options—endpoint selection and CSS themes—to illustrate the full combinational diversity achievable with the generator.

Baseline Configuration Diversity

To estimate the baseline diversity, we start by calculating combinations of two SQL injection vulnerabilities from the existing set of nine unique injection points. Using the combinational formula $nC2 = \frac{n!}{2!(n-2)!}$, where $n = 9$, we get $\frac{9!}{2!(9-2)!} = 36$ pairs of SQL injection vulnerabilities

When factoring in DBMS compatibility and available hardening measures, these pairs expand to a total of 293 distinct, compatible configurations.

The generator also provides five distinct authorization flaws. Because all authorization flaws share the same middleware, only one can be used at a time, resulting in 5 possible configurations for authorization flaws.

Thus, the total baseline diversity (without randomization) is:

$293 \times 5 = 1,465$ unique baseline combinations

Extended Configuration Diversity with Randomization

To increase the perceptual diversity among generated applications, the generator randomizes endpoint names and applies visual themes (CSS). The endpoint randomization selects from pre-defined lists of semantically equivalent names across seven endpoint categories, yielding:

$11 \times 11 \times 6 \times 6 \times 6 \times 6 \times 6 = 940,896$

unique endpoint combinations

Additionally, three distinct visual CSS themes—**90s**, **Modern**, and **Cyber**—further differentiate the appearance of the generated applications.

Taking into account these randomizations, the total combinational diversity is calculated as follows:

$$
\begin{aligned}
1,465 \quad & \text{(baseline combinations)} \\
\times\ 940,896 \quad & \text{(endpoint combinations)} \\
\times\ 3 \quad & \text{(CSS themes)} \\
= 4{,}134{,}182{,}640 \quad &
\end{aligned}
$$

Thus, considering all available randomization and configuration options, the generator can produce a theoretical maximum of:

4,134,182,640 unique application configurations

5 Evaluation and Testing

To evaluate the applicability of the tool in a real cybersecurity exercise, we have conducted a testing session in collaboration with instructors of the PV297 Cybertraining [40] course at the Faculty of Informatics, Masaryk University in Brno. The main objective of the session was to deploy a working service into a FAUST AD environment and assess the usability and quality of the tool and the resulting generated service. A secondary objective was to identify any bugs, shortcomings, and places for further improvement of the tool.

The testing session was performed by Daniel Gerzicak – an instructor of PV297 responsible for maintaining the Attack Defence infrastructure for cybersecurity exercises in the course. He had prior experience with Capture the Flag competitions and IT infrastructure deployment. While not a specialist in web security, the tester had a working understanding of common vulnerabilities, aligning with the profile of a typical end user of the tool – namely, educators and CTF event organizers.

The tester's goal was to generate a working web application compatible with the FAUST AD environment with two simple and easy-to-exploit vulnerabilities that are suitable for use in an introductory exercise within a controlled educational setting. The selected vulnerabilities were intended to provide a low entry barrier for students with limited prior experience in web security. Additionally, the generator was expected to produce an application with a functional user interface, minimal deployment overhead, and a predictable vulnerability behavior that would align with the learning objectives of the session.

5.1 Testing Procedure

The testing process was structured into four main stages: introduction, usage, deployment, and feedback. In the introductory phase, the tester was presented only with the general purpose and scope of the generator and was given access to the documentation. No in-depth technical explanation of the tool was given to the tester. This approach was chosen to avoid shaping the tester's expectations or revealing information that could bias their natural interaction with the tool. During

the usage phase, the tester accessed the GitLab repository and was provided with the generator's documentation, which contained the necessary steps for initiating the generation process. While support was available throughout the session, the tester was encouraged to proceed independently and only request assistance if necessary. Once the application was successfully generated, it was deployed to the FAUST Attack-Defense infrastructure using the provided deployment templates. The final stage of the test consisted of a structured feedback session, where the tester shared observations regarding usability, documentation clarity, deployment experience, and overall impressions of the generated application.

5.2 Observations

The documentation was adequate for completing the setup without direct assistance. Configuration through environment variables and using GitLab's interface provided a functional and repeatable workflow, even for users without extensive experience in DevOps tools. The use of GitLab CI proved to be an accessible solution, reducing the need for local installations or environment setup.

The tool produced a fully functional web application with correct syntax. It implemented all the required features and integrated the intended vulnerabilities.

The inclusion of a pre-generated exploit script significantly contributed to the pedagogical value of the tool. During the exercise, the students were unable to exploit the flaw on their own. The instructor used the script to demonstrate the vulnerability and its exploitation in practice after the exercise ended.

Deployment into the FAUST environment proceeded with only minor adjustments. The output of the generator conformed to the expected directory structure and conventions. The checkerbot confirmed service availability, validating compatibility with the existing infrastructure.

Several shortcomings were also identified. The tester selected an unsupported vulnerability type (`xss`) during the initial generation attempt. The pipeline failed without issuing a clear error message, and there was no prior indication that the chosen combination was

invalid. At the time of testing, the documentation did not include a complete list of valid configurations.

Several shortcomings were identified during testing. The tester selected an unsupported vulnerability type (`xss`) during the initial generation attempt. The pipeline failed without providing a clear error message, and there was no prior indication that the selected configuration was invalid. Although `xss` was documented as an example configuration option at the time, its inclusion was erroneous, as the configuration was not supported.

One stability issue emerged during runtime when using MySQL, one of the randomly picked Database Management Systems. As the service started, the generated application created a long-lived MySQL connection that was later used to execute all queries. The connection had a pre-set timeout of 8 hours, after which it automatically closed, causing an unhandled `ECONNREFUSED` exception on an arbitrary query run. This resulted in a service crash post-deployment. The randomness of the generator, while useful for variety, introduced unpredictability that affected reliability when the user opted not to select a DBMS. A similar issue was also identified on the PostgreSQL DBMS. Both problems were fixed before the generated application was used in the course.

The tester initially selected vulnerabilities classified as low difficulty. However, students found them difficult to exploit. For example, the embedded SQL injection was error-based—a technique that turned out to be more complex than initially assessed. This discrepancy indicates that the difficulty classification was inaccurate and should be reconsidered to better reflect the actual technical complexity.

While testing the GitLab CI interface, the tester identified minor usability issues. Configuration via CI/CD variables was not immediately intuitive for users unfamiliar with GitLab CI/CD, and the placement of the user input within the interface was not obvious.

Finally, minor inconsistencies in directory names and paths required manual correction before deployment could proceed. These issues, although not critical, could be disruptive in time-constrained settings.

5.3 Improvements

Following the initial testing session, a number of the identified issues were addressed in a revised version of the generator. The documentation was updated to include clear guidance on supported configurations, minimizing the risk of selecting unsupported options and encountering silent failures. The issue with long-lived MySQL connection objects was addressed. The difficulty ratings assigned to vulnerabilities were revisited, with the aim of better aligning their classification with both technical complexity and user-facing clarity. Improvements were also made to the GitLab CI pipeline, including more informative error messages and early validation of input parameters to prevent avoidable build failures.

Additionally, several of the tester's suggestions were implemented. These included the addition of a human-readable list of available vulnerabilities, frontend themes, and database types. Users can now explicitly select which components they want the generator to use, reducing reliance on random selection and improving reproducibility in both testing and educational contexts.

6 Conclusions and Future Work

This thesis has presented the design, implementation, and evaluation of a specialized generator for intentionally vulnerable web applications that addresses an unmet need in cybersecurity education and training. While several vulnerability generators exist in the educational technology landscape, this work makes a distinct contribution by specifically targeting Attack-Defense CTF competitions with a level of granularity not previously available. The generator enables precise specification of which database queries and parameters should contain vulnerabilities, providing instructors with fine-grained control over the educational scenarios they create. This approach differs significantly from existing tools that typically offer only binary vulnerability inclusion without the ability to target specific application components or control vulnerability placement with such precision.

The generator produces randomized yet functionally consistent web applications with configurable vulnerabilities. Its support for precise vulnerability targeting and adjustable difficulty levels makes it particularly well-suited for A/D CTF scenarios, enabling instructors to design varied and pedagogically meaningful challenges across course iterations.

The generator produces randomized yet functionally consistent web applications with configurable vulnerabilities while maintaining full compatibility with the FAUST framework infrastructure. This combination of targeted vulnerability placement, A/D CTF compatibility, and configurable difficulty levels represents an advancement in educational technology for cybersecurity training, enabling instructors to create varied challenges that meet specific pedagogical objectives across multiple course iterations.

The evaluation conducted with the instructor of the PV297 Cybertraining course demonstrated the generator's practical utility in educational settings. The generated applications successfully integrated into the FAUST Attack-Defense framework with minimal adjustment, validating the compatibility with existing CTF infrastructure. The tool's automatic generation of proof-of-concept exploit scripts provided additional value for instructors, enabling them to demonstrate vulnerability exploitation when students faced challenges.

However, the evaluation also revealed important areas for improvement. The mismatch between the technical classification of vulnerability difficulty and the practical discoverability by students highlights the need for more nuanced difficulty metrics that consider both implementation complexity and user-facing characteristics. Additionally, the current random selection of some parameters, while increasing variety, occasionally introduces unpredictable behavior that could impact reliability in time-sensitive educational contexts.

While the current implementation successfully supports SQL injection and authorization vulnerability types with varying levels of difficulty, several limitations remain to be addressed in future work.

Future Work

Enhanced SQLi Validation The current validation capabilities focus primarily on first-order SQL injections with basic hardening mechanisms. Future work should extend validation support to include second-order SQL injection vulnerabilities, where the malicious input is stored in a database and later retrieved and executed in a different context. Additionally, the validation scripts should be enhanced to test the backslash escape filter bypass mechanism, which is currently not supported by the PoC scripts. These improvements would provide more comprehensive coverage of SQL injection vulnerability types.

Optimized Vulnerability Selection Algorithms The current algorithm for selecting compatible vulnerabilities exhibits inefficiencies when generating applications with a larger number of SQL injection vulnerabilities. Users occasionally encounter errors such as "Not enough vulnerabilities for selected hardening and DBMS" and must restart the generation process. Future work should refactor this algorithm to optimize the selection process, potentially implementing a constraint satisfaction approach that more effectively identifies compatible combinations of vulnerabilities, hardening mechanisms, and database systems.

User-Controlled Hardening Selection The current implementation randomly selects hardening mechanisms based on requested difficulty

levels. A beneficial enhancement would be to allow users to explicitly specify which hardening mechanisms should be applied, providing greater control over the generated vulnerabilities. This would enable instructors to tailor scenarios to specific learning objectives, such as teaching students how to bypass specific types of input filtering.

Modularized Application Baseline Architecture The current generator architecture contains tight coupling between the generator core and the baseline application template. Certain features that conceptually belong at the application baseline level are implemented within the generator core, creating potential compatibility issues when developing alternative baselines. Future work should refactor the architecture to more clearly separate these concerns, making it easier to develop new application baselines in different programming languages or frameworks without requiring changes to the generator core.

Additional Vulnerability Types The current implementation focuses on SQL injection and authorization vulnerabilities. Future extensions could incorporate additional vulnerability types such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), server-side template injection, and other OWASP Top Ten risks. This would enable more comprehensive security training scenarios covering a broader range of web application security concerns.

By addressing these limitations, future versions of the generator will provide an even more versatile and robust tool for cybersecurity education, supporting a wider range of educational objectives and training scenarios while maintaining the essential qualities of ease of use and compatibility with existing CTF infrastructures.

Usability testing Further usability testing is planned as part of the Summer School of the Czech National Cybersecurity Competition in June 2025. The event will involve participants with stronger experience in A/D CTFs, providing valuable input on the A/D aspects of the generated service. Feedback from this advanced user group will support further refinement of the tool's functionality.

A Content of the attachment

This appendix contains the source code for the Vulnerability Generator developed for this bachelor's thesis. The key components included are:

1. **README.md** – Detailed usage instructions and documentation.
2. **run-docker.sh** – Script for running the generator in a Docker container.
3. **test_service/** – Template directory containing the base service.
4. **generate.py** – Core code for generating vulnerable services.
5. **vulns.yaml** – Definitions of all available vulnerabilities.
6. **resources/** – Supporting files including CSS styles and database setup templates.
7. **LICENSE** – Specifies that the project is released under the MIT License.

Bibliography

1. EUROPEAN UNION AGENCY FOR CYBERSECURITY (ENISA). *ENISA Threat Landscape 2024* [online]. 2024. [visited on 2025-05-09]. Available from: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024.
2. ŠVÁBENSKÝ, Valdemar; ČELEDA, Pavel; VYKOPAL, Jan; BRIŠÁKOVÁ, Silvia. Cybersecurity Knowledge and Skills Taught in Capture the Flag Challenges. *Computers & Security*. 2020, vol. 102. ISSN 0167-4048. Available from DOI: 10.1016/j.cose.2020.102154.
3. BALON, Tyler; BAGGILI, Ibrahim. Cybercompetitions: A survey of competitions, tools, and systems to support cybersecurity education. *Education and Information Technologies*. 2023, vol. 28, no. 9, pp. 11759–11791. ISSN 1573-7608. Available from DOI: 10.1007/s10639-022-11451-4.
4. *DEF CON Cybersecurity Conference* [online]. 2025. [visited on 2025-05-14]. Available from: https://defcon.org/.
5. CARNEGIE MELLON UNIVERSITY. *picoCTF: Free Cybersecurity Education Platform* [online]. 2025. [visited on 2025-05-14]. Available from: https://picoctf.org/.
6. FAU SECURITY TEAM. *FAUST CTF Gameserver GitHub Repository* [https://github.com/fausecteam/ctf-gameserver]. 2024. [visited on 2025-05-05].
7. KUCEK, Stela; LEITNER, Maria. An Empirical Survey of Functions and Configurations of Open-Source Capture the Flag (CTF) Environments. *Journal of Network and Computer Applications*. 2020, vol. 151, p. 102470. ISSN 1084-8045. Available from DOI: https://doi.org/10.1016/j.jnca.2019.102470.
8. BEURAN, Razvan. Capture the Flag Platforms. In: *Cybersecurity Education and Training*. Singapore: Springer Nature Singapore, 2025, pp. 193–219. ISBN 978-981-96-0555-2. Available from DOI: 10.1007/978-981-96-0555-2_10.

9. OWASP PROJECT. *OWASP SiteGenerator Project* [online]. 2007. [visited on 2025-05-09]. Available from: https://web.archive.org/web/20191029114828/https://www.owasp.org/index.php/OWASP_SiteGenerator.

10. SU, Syue Siang. *VWGen: Vulnerable Web Application Generator* [online]. 2017. [visited on 2025-05-09]. Available from: https://github.com/qazbnm456/VWGen.

11. SCHREUDERS, Z. Cliffe; SHAW, Thomas; SHAN-A-KHUDA, Mohammad; RAVICHANDRAN, Gajendra; KEIGHLEY, Jason; ORDEAN, Mihai. Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events. In: *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. Vancouver, BC: USENIX Association, 2017. Available also from: https://www.usenix.org/conference/ase17/workshop-program/presentation/schreuders.

12. FAU SECURITY TEAM. *FAUST CTF Gameserver Architecture* [https://ctf-gameserver.org/architecture/]. [N.d.]. Accessed: 2024-05-05.

13. FAU SECURITY TEAM. *Checker Scripts Documentation* [https://ctf-gameserver.org/checkers/]. 2024. [visited on 2025-05-05].

14. FAU SECURITY TEAM. *FAUST CTF 2024 Scoreboard* [https://2024.faustctf.net/competition/scoreboard/]. 2024. Accessed: 2025-05-22.

15. PALLETS TEAM. *Jinja - Full Featured Template Engine for Python* [online]. 2025. [visited on 2025-05-04]. Available from: https://jinja.palletsprojects.com/.

16. MAO, Tobias. *SQLGlot: SQL Parser, Transpiler, and Optimizer* [online]. 2023. [visited on 2025-05-12]. Available from: https://github.com/tobymao/sqlglot.

17. OPENJS FOUNDATION. *Node.js JavaScript Runtime* [online]. 2009. [visited on 2025-05-12]. Available from: https://nodejs.org/en/.

18. OPENJS FOUNDATION. *Express.js Web Framework* [online]. 2010. [visited on 2025-05-12]. Available from: https://expressjs.com/.

19. EVANS, Clark; NET, Ingy döt; BEN-KIKI, Oren. *YAML Ain't Markup Language (YAML™) Version 1.2.2*. 2021. Available also from: https://yaml.org/spec/1.2.2/. Accessed: 2025-05-12.

20. BELLUCCI, Daniele; GUIMARAES, Bernardo Damele Assumpcao; STAMPAR, Miroslav. *SQLMap: Automatic SQL Injection and Database Takeover Tool* [online]. 2006. [visited on 2025-05-12]. Available from: http://sqlmap.org/.

21. DOCKER INC. AND RED HAT INC. *Containerization Technologies* [online]. 2025. [visited on 2025-05-04]. Available from: https://www.docker.com/.

22. GITLAB INC. *GitLab CI/CD Documentation* [online]. 2025. [visited on 2025-05-04]. Available from: https://docs.gitlab.com/ee/ci/.

23. MITRE CORPORATION. *Common Weakness Enumeration (CWE) – Frequently Asked Questions*. 2025. Available also from: https://cwe.mitre.org/about/faq.html. Accessed: 2025-05-13.

24. MITRE CORPORATION. *CVE Glossary*. 2025. Available also from: https://www.cve.org/ResourcesSupport/Glossary. Accessed: 2025-05-22.

25. MITRE CORPORATION. *Common Weakness Enumeration (CWE)*. 2025. Available also from: https://cwe.mitre.org/about/index.html. Accessed: 2025-05-13.

26. MITRE CORPORATION. *2024 CWE Top 25 Most Dangerous Software Weaknesses*. 2024. Available also from: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: 2025-05-13.

27. OWASP FOUNDATION. *OWASP Top Ten Project*. 2021. Available also from: https://owasp.org/www-project-top-ten/. Accessed: 2025-05-13.

28. SU, Zhendong; WASSERMANN, Gary. The essence of command injection attacks in web applications. *SIGPLAN Not.* 2006, vol. 41, no. 1, pp. 372–382. ISSN 0362-1340. Available from DOI: 10.1145/1111320.1111070.

29. MITRE. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*. 2024. Available also from: https://cwe.mitre.org/data/definitions/89.html. Accessed: 2025-05-13.

30. MITRE. *CWE-863: Incorrect Authorization*. 2024. Available also from: https://cwe.mitre.org/data/definitions/863.html. Accessed: 2025-05-13.

31. MITRE. *CWE-862: Missing Authorization*. 2024. Available also from: https://cwe.mitre.org/data/definitions/862.html. Accessed: 2025-05-13.

32. CONTRIBUTORS, Podman. *Podman Documentation* [online]. 2025. [visited on 2025-05-14]. Available from: https://docs.podman.io/en/latest/.

33. W3SCHOOLS. *PHP MySQL Prepared Statements*. 2025. Available also from: https://www.w3schools.com/php/php_mysql_prepared_statements.asp. Accessed: 2025-05-14.

34. SQUARCINA, Marco. *A/D Service Development Guidelines* [online]. 2023. [visited on 2025-05-04]. Available from: https://github.com/lavish/CTF-Doc/blob/main/SERVICE.md.

35. SQLITE CONSORTIUM. *SQLite Documentation: Literal Values And Constants* [online]. 2025. [visited on 2025-05-04]. Available from: https://www.sqlite.org/lang_expr.html#literal_values_constants_.

36. POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL Documentation: Lexical Structure - String Constants* [online]. 2025. [visited on 2025-05-04]. Available from: https://www.postgresql.org/docs/current/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS-ESCAPE.

37. ORACLE CORPORATION. *MySQL Reference Manual: String Literals* [online]. 2025. [visited on 2025-05-04]. Available from: https://dev.mysql.com/doc/refman/8.4/en/string-literals.html.

38. MOZILLA FOUNDATION. *Array.toString() JavaScript reference* [online]. 2025. [visited on 2025-05-04]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toString.

39. THE POSTMAN TEAM. *What Is a REST API? Examples, Uses, and Challenges*. 2020. Available also from: https://blog.postman.com/rest-api-examples/. Accessed: 2025-05-14.

40. FACULTY OF INFORMATICS, MASARYK UNIVERSITY. *PV297 Cybertraining Course* [online]. 2025. [visited on 2025-05-04]. Available from: https://is.muni.cz/predmet/fi/PV297.