FETT: Fault Injection as an Educational and Training Tool in Cybersecurity

Anaé De Baets*, Guillaume Nguyen*, Xavier Devroey*, and Fabian Gilson†
*NADI, Computer Science Faculty, University of Namur, Namur, Belgium.
Emails: anae.debaets@student.unamur.be, guillaume.nguyen@unamur.be, xavier.devroey@unamur.be
†Computer Science and Software Engineering, University of Canterbury, New Zealand.
Email: fabian.gilson@canterbury.ac.nz

***Abstract*—In this paper, we present FETT, a fault injection tool for educational and training purposes addressed to educators and students in cybersecurity. Our tool aims to analyze and inject vulnerabilities into existing Django web applications for educational purposes. Indeed, security education often relies on either abstract theoretical instruction or overly simplistic examples. This tool bridges the gap between theory and practice by modifying real web applications in a targeted, reproducible way. With its user-friendly interface and modular vulnerability injection, instructors can create challenges tailored to specific learning goals, while students engage directly with code that simulates production-level vulnerabilities. We evaluated FETT based on five publicly available GitHub projects and six student projects from the last three academic years (2022-2024). We successfully managed to efficiently inject vulnerabilities inspired by the OWASP top 10:2021 while keeping the core functionalities of the target application operational. The project is publicly available at https://gitlab.com/fabgilson/django-vulnerability-injector. A demonstration is available at https://youtu.be/ZYQs2vLzbyE.**

***Index Terms*—cybersecurity, education, OWASP, vulnerability injection, Django, web application security**

I. Introduction

There is a growing need for realistic, hands-on environments where learners can safely explore web vulnerabilities for cybersecurity education and training [1]. Manually creating these environments is often resource-intensive and requires significant technical expertise. To address this, our tool introduces a vulnerable web application configurator: a system designed to modify an existing web application by injecting known vulnerabilities based on user-selected criteria.

Rather than generating an application from scratch, FETT takes a pre-existing, functioning web application that it systematically alters to include specific vulnerabilities from the Open Worldwide Application Security Project (OWASP) top 10:2021 [2]. The OWASP publishes a regularly updated list of the ten most critical web application security risks. This approach ensures that the resulting application maintains realistic structure and behavior while incorporating security flaws that are both realistic and educationally meaningful.

FETT seeks to replace the manual process of creating a new vulnerable web application for each new training, to ensure that the training material remains fresh, relevant, and effective for each cohort. This process is not only time-consuming but also difficult to scale and maintain consistently. Our goal is to streamline and automate the generation of training environments by modifying an existing web application and injecting vulnerabilities into it. This significantly reduces the instructor's workload while introducing flexibility and modularity into how vulnerabilities are introduced and managed.

By allowing users to select specific OWASP vulnerabilities and automatically apply them to a known codebase, FETT enables faster setup of new training scenarios without compromising on educational value. It also allows for consistent quality and structure across course iterations, while maintaining the ability to introduce variation from cohort to cohort.

Ultimately, FETT supports the broader objective of making cybersecurity education more efficient, scalable, and engaging, benefiting both instructors and students.

The paper is structured as follows: Section II describes the background and motivation for the development of FETT. Then, in section III, we lay out the architecture of our tool, and in section IV, we evaluate the performance of our tool. Last, we discuss the limitations of our tool and conclude.

II. Background & Motivation

In this section, we discuss the requirements of FETT following the more thorough analysis presented by De Baets [3] in reviewing the current literature on existing practices for integrating automated security simulation and automated vulnerability injection for educational programs. We report the key findings in four main categories.

a) Automation and Dynamic Adaptation: This is the most explored automation technique in cybersecurity exercises, particularly using automated vulnerability injection tools and dynamic code adaptation. These systems were shown to

This research was supported by the CyberExcellence by DigitalWallonia project (No. 2110186), funded by the Public Service of Wallonia (SPW Recherche).

This is the authors' version. The final version is accepted for publication in the *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025*, Seoul, South Korea.

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, including reprinting/republishing this material for advertising or promotional purposes, collecting new collected works for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

significantly improve the flexibility and scalability of cybersecurity training environments [4].

b) Code Generation for Security Simulation: Papers that addressed code generation emphasized the importance of real-time code adaptation in training simulations. Tools designed for generating dynamic security configurations allowed for more personalized and responsive training experiences [5].

c) Security Testing Parametrization: A subset of papers focused on generating and parametrizing security testing scenarios. These tools allowed for the customization of test conditions, making it easier to tailor cybersecurity exercises to various skill levels and security challenges [4]–[8].

d) Cybersecurity Training Platforms: Several papers discussed designing and implementing comprehensive training platforms that integrate multiple security scenarios, from simple attacks to complex multi-layered threats. These platforms were highly valued for their ability to provide immersive, hands-on experiences [6], [7]. FETT's objective is to offer the same learning experience for cybersecurity assessment material.

e) Summary: Existing research highlights the rapid evolution of cybersecurity training, particularly in automation, scenario generation, vulnerability injection, and adaptive learning. These advancements provide both insights and demonstrate interest in developing a system that dynamically generates vulnerable web apps for training purposes.

f) Our solution: While configuration-based automation ensures repeatability and ease of deployment, our system extends beyond static templates by integrating dynamic scenario generation, leveraging techniques similar to those in CyExec* [9] and SecGen [8], to inject randomized yet controlled vulnerabilities. This approach aligns with the trend toward variability and individualization, addressing limitations of rigidity seen in configuration-only systems.

Furthermore, incorporating elements from model-driven and DSL-based systems, such as those discussed in CRACK [10] and ASL [11], informs the structured backend of our generator, ensuring consistency in how vulnerabilities are described and deployed. However, unlike these formalized systems, our goal is to abstract away the complexity, offering an intuitive interface for non-experts to create unique training instances.

Ultimately, the system we are building synthesizes key strengths from existing methods: **automation**, **dynamic generation**, and **customization**. We aim at overcoming their limitations by prioritizing **usability** and **flexibility** to contribute to the next generation of cybersecurity training tools, empowering educators and learners with accessible, on-demand, realistic training environments.

III. Architecture

In this section, we lay out the user interaction flow, the key components of the system implementation, the details of the environment setup, the configuration interface, the code parsing and transformation process, and an example use case to demonstrate the system's capabilities. Fig. 1 shows the high-level architecture.

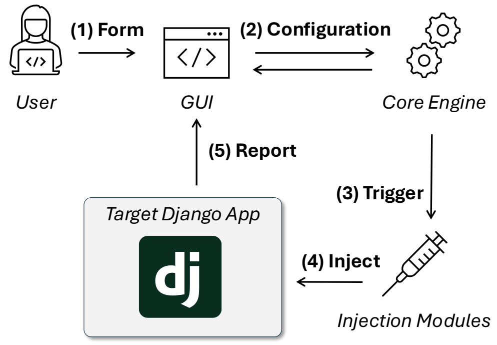


Fig. 1. The high-level architecture of FETT.

Targeting Django applications for vulnerability injection is driven by the framework's popularity, well-defined structure, and everyday use in educational and industry contexts. Nevertheless, several key technological constraints shaped the tool's design, such as the framework rigidity, preserving the code integrity (e.g., formatting, comments, syntactic correctness), and the fact that FETT assumes a conventional Django layout. Projects with custom structures may require manual adaptation. Despite these constraints, focusing on Django allows the tool to automate a previously manual and repetitive task while ensuring the resulting applications remain pedagogically valuable and functionally stable.

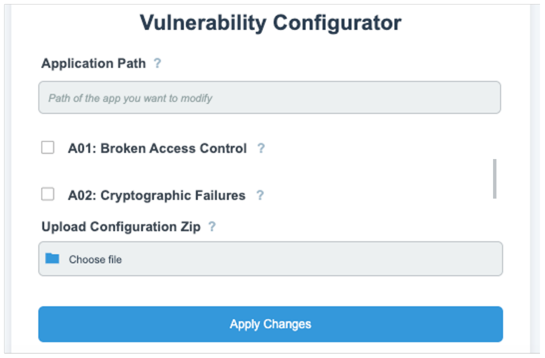
To structure these injections meaningfully and consistently, the tool is based on the *OWASP top 10:2021* vulnerability list, a widely recognized list of the most critical categories of web application security risks, such as SQL injection, authentication failures, and logging failures. Users interact with the tool via a graphical interface, where they can selectively introduce vulnerabilities corresponding to OWASP categories. By simulating real-world flaws in a systematic and replicable manner, the tool supports hands-on learning in a safe, contained environment.

A. Graphical User Interface

The configuration interface, item (1) in Fig. 1, offers a user-friendly platform for customizing vulnerability injection as shown in Fig. 2. The user can select any category and configure the vulnerability to inject. The user can also load a configuration file from prior injections.

B. Input Handling

We use Python `3.12.3` for its flexibility in text parsing, file handling, and web and testing frameworks integration. For the back-end development, (2) in Fig. 1, form submissions management, server-side logic, and file processing, we use Django `5.2.1`. Upon submission, all inputs are validated on the server side. Missing or invalid entries, such as unsupported file types or empty application paths, trigger descriptive error messages. The form supports multiple vulnerabilities and transformation options. For specific vulnerabilities like Server-Side Request Forgery (SSRF), the system restricts the



Vulnerability Configurator

Application Path ?

Path of the app you want to modify

A01: Broken Access Control ?

A02: Cryptographic Failures ?

Upload Configuration Zip ?

Choose file

Apply Changes

Fig. 2. A screenshot of the user interface of FETT.

Before:
```
user = auth.authenticate(
        username=username,
        password=password)
```
After:
```
connection = sqlite3.connect('db.sqlite3')
cursor = connection.cursor()
cursor.execute(
        f"SELECT *
        FROM auth_user
        WHERE username = '{username}'
        AND password = '{password}'")
user_login = cursor.fetchall()
user_login = User.objects.get(username = username)
```

Fig. 3. An example of vulnerability injection (A03 Code Injection)

upload to a valid Python file to ensure compatibility with the overridden injection.

C. Code Parsing and Injection with `LibCST`

The core functionality of the system relies on `LibCST 1.7.0`, a Concrete Syntax Tree (CST) parser, enabling precise parsing and transformation of Python source code while preserving formatting, structure, and syntactical correctness. This ensures that vulnerability injection, (3) and (4) in Fig. 1, does not compromise code integrity and preserves the code structure, including comments. Tab. I lists the different vulnerabilities supported by FETT. Each vulnerability A01 to A10 can be injected in various ways. To support this, FETT defines (up to) three different variants for the same vulnerability.

Example: In Fig. 3, FETT looks for a call to the secure *authenticate()* method and replaces it with an unsecured call using a formatted string (`f String`) vulnerable to SQL injection. This example is *A03 Code Injection* vulnerability and shows the capability of one of the three variants. The two other variants inject code vulnerable to cross-site scripting (XSS) attacks by using unescaped HTML code, or unsafe usages of Django's `render()` function. The detailed list and descriptions of injectors are available in [3].

D. Report, Resulting App and Feedback

The interface streamlines the configuration process, making it accessible even to non-technical users. By abstracting away the complexity of code transformation, the system enables safe, customizable vulnerability injection with minimal effort. After processing, a report is generated, see item (5) in Fig. 1, In this report, the system provides the following feedback: (i) a success page with a summary of the vulnerabilities chosen, their parameters, and warning messages about unsuccessful applications, if any; (ii) a download link to a ZIP archive containing a `vulnerability_config.json` file detailing the applied modifications and, if applicable, the python file containing the overridden function (for SSRF injection). All changes are applied to the supplied code in place. In case of errors (e.g., invalid path), a contextual error page is shown to help users correct the issues.

IV. Evaluation

We performed a preliminary evaluation of FETT using Django-based applications with varied sizes and complexity. We focus on *effectiveness* by looking at injections that could be applied, and *efficiency* by looking at duration and system resource consumption. We applied FETT on five open-source projects from GitHub, and six student submissions from previous coursework at the University of Canterbury.

a) Benchmark Selection: Candidate open-source repositories were identified using GitHub's search and filter functionality, prioritizing the following criteria: (i) more than 100 stars used as a proxy for code quality; (ii) last updated in 2020 or later; (iii) a modest repository size, i.e. $\leq$ 25 files and $\leq$ 100 average lines of code per file, to allow manual testing of existing functionality post-injection within reasonable time and resource constraints.

The selected open-source projects are **Django React Ecommerce**,[1] **Django Web-App**,[2] **Django Social Media Website**,[3] **Versity**,[4] and **Dwitter**.[5] Despite having only a single GitHub star at the time of selection, we included the **Versity** project due to its clear structure and development activity. The remaining applications were drawn from a university-level cybersecurity course, where students were tasked with identifying and fixing vulnerabilities in instructor-provided Django applications, which purposefully contained OWASP vulnerabilities. Two of the highest-graded fixed submissions were selected from each of the last three academic years (2022-2024). This ensured the inclusion of purpose-built and secured codebases, matching the typical size of assessments targeted by FETT. For each project, the number of files ranged from 13 to 21, the average number of lines per file ranged from 28.5 to 87.3, and all presented sufficient architectural complexity for educational apps.

b) Effectiveness: We manually inspected the different projects to analyze FETT's effectiveness. Tab. I reports the number of successful injections for all projects.

[1]https://github.com/justdjango/django-react-ecommerce
[2]https://github.com/smahesh29/Django-WebApp
[3]https://github.com/tomitokko/django-social-media-website
[4]https://github.com/emhash/Versity-Class-Management-System
[5]https://github.com/lionleaf/dwitter

TABLE I
SUCCESSFULLY INJECTED VULNERABILITIES.

OWASP Category	Vulnerability	#
A01 Broken access control	no check admin page	0
A01 Broken access control	remove `login_required`	10
A02 Cryptographic failure	replace secret key	11
A02 Cryptographic failure	plain text passwords	6
A03 Injection	replace with raw SQL	9
A03 Injection	unescaped HTML - XSS	6
A03 Injection	unsafe `render()`	2
A04 Insecure design	unchecked password change	4
A04 Insecure design	unsafe `get_object()`	10
A04 Insecure design	unsafe `ObjectDoesNotExist`	1
A05 Security misconfig.	enable debug mode	7
A05 Security misconfig.	allow all hosts	7
A05 Security misconfig.	default `admin` console	3
A06 Vulnerable components	downgrade libraries	11
A07 Authentication failure	remove decorators	5
A07 Authentication failure	remove password validation	5
A08 Data integrity failure	remove upload validation	0
A08 Data integrity failure	remove form validation	9
A09 Logging failure	remove log statements	6
A09 Logging failure	make logs public	6
A09 Logging failure	log leak	9
A10 SSRF	inject custom script	11

A02.replace-secret-key, A06.downgrade-libraries, and A10.-SSRF could be injected into all 11 projects. Typical programming mistakes, such as A03.SQL-injection, A03.XSS, A08.form-validation, and A09.log-leak could be injected into most projects. Only A01.admin-page and A08.upload-validation could not be injected into any of the projects. During our manual analysis, we observed that unfeasible injections were primarily due to architectural incompatibilities (e.g., non-standard access control, no file upload) or missing prerequisites in the codebase (e.g., no admin panel, no access controls). Despite Django's structural constraints, we noticed that open-source projects do not always follow the prescribed architecture style, leading to more non-applicable cases: 45 injections could be applied to the five open-source projects (9 per project on average, 41%). In contrast, 93 could be applied to the students' projects (15.5 per project on average, 70%).

Last, we manually tested the applications after injection from manual test cases gathered pre-injection. No critical failures preventing the regular usage of the app were observed post-injection. This confirms that FETT preserves the applications' integrity while introducing vulnerabilities, fulfilling its role as an instructor-facing tool for preparing stable, deliberately insecure applications for student assessments.

c) Efficiency: FETT's performance was measured in terms of total and average processing time per file, which ranged from 0.47s to 1.61s per file and from 8.45s to 24.20s per project. The hardware conditions under which the measurements were taken include a Z2 Tower G9 workstation equipped with an Intel Core i7-13700 CPU, 32GB of RAM, 1TB NVMe SSD, and an NVIDIA RTX 4070 GPU with 12GB of VRAM. Resource usage metrics, including CPU time and memory footprint, were monitored throughout the injection process. With a resident set size staying under 71MB and CPU usage not exceeding 2.94s, our results demonstrate that FETT remained lightweight and did not incur significant computational overhead.

V. Conclusion and Future Work

We developed FETT with a clear educational objective: to facilitate hands-on cybersecurity training by lowering the barrier to creating realistic, vulnerable web applications. Traditional training methods often involve synthetic examples that lack realism or highly customized applications that are costly to produce and maintain. Our tool balances realism and ease-of-use by modifying functional Django applications to include targeted vulnerabilities, selected by the educator. Our preliminary evaluation demonstrates that the tool performs efficiently on projects of varied complexity, with high coverage of many OWASP vulnerabilities without any impact on the application's stability.

Future work includes improving the FETT to extend the support to non-standard Django applications, extending the injection to non-Python files, and automating the post-injection vulnerability validation (now done manually). We also intend to improve the configurability of the injectors to add a probability for applications. This enables the rapid generation of several vulnerable versions from the same Django application for a classroom. We also plan to run a full-scale evaluation of FETT, both on its vulnerability injection ability to precisely identify the current limitations of the static analyses used in the injectors and its impact on cybersecurity education. For the former, we intend to extend our current benchmark to larger Django applications and follow a similar evaluation method. For the latter, we will devise a longitudinal study with educators to understand the impact of FETT on their pedagogical approach and students' learning.

References

[1] M. Bishop, D. Burley, S. Buck, J. J. Ekstrom, L. Futcher, D. Gibson, E. K. Hawthorne, S. Kaza, Y. Levy, H. Mattord, and A. Parrish, *Cybersecurity Curricular Guidelines*. Springer International Publishing, 2017, p. 3–13.

[2] OWASP, "OWASP Top 10:2021," https://owasp.org/Top10/, (last visit 08/07/2025).

[3] A. De Baets, "Automated vulnerability injection in web applications: A tool to support cybersecurity education," Master's thesis, University of Namur, Namur, Belgium, 2025.

[4] M. Malone, Y. Wang, and F. Monrose, "An online gamified learning platform for teaching cybersecurity and more," in *Proceedings of the 22nd Annual Conference on Information Technology Education*. Association for Computing Machinery, 2021, pp. 29–34.

[5] M. Benzi, G. Lagorio, and M. Ribaudo, "Automatic challenge generation for hands-on cybersecurity training," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 496–503.

[6] W. H. Ang, H. Guo, and E. G. Chekole, "Vulngen: Vulnerable virtual machine generator," in *2023 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, 2023, pp. 1–8.

[7] C. Justice and R. Vyas, "Cybersecurity education: Runlabs rapidly create virtualized labs based on a simple configuration file," in *ASEE Annual Conference and Exposition, Conference Proceedings*, 06 2017.

[8] Z. C. Schreuders, T. Shaw, M. Shan-A-Khuda, G. Ravichandran, J. Keighley, and M. Ordean, "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)*, 2017.

[9] R. Nakata and A. Otsuka, “Cyexec*: A high-performance container-based cyber range with scenario randomization,” *IEEE Access*, vol. 9, pp. 109 095–109 114, 2021.

[10] E. Russo, G. Costa, and A. Armando, “Building next generation cyber ranges with crack,” *Computers & Security*, vol. 95, p. 101837, 2020.

[11] S. Arshad, M. Alam, S. Al-Kuwari, and M. H. A. Khan, “Attack specification language: Domain specific language for dynamic training in cyber range,” in *2021 IEEE Global Engineering Education Conference (EDUCON)*, 2021, pp. 873–879.