International Journal Software Engineering and Computer Science (IJSECS)
6 (1), 2026, 231-245
Published Online April 2026 in IJSECS (http://www.journal.lembagakita.org/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v6i1.6809.

RESEARCH ARTICLE Open Access

Design and Development of a Vulnerability Simulation-Based Cybersecurity Training Platform for Secure Programming

Habib Nurfaizal [1], **Afrizal Zein** [2*]

[1,2*] Department of Information Systems, Universitas Pamulang, South Tangerang City, Banten Province, Indonesia.

**Corresponding author*: dosen02807@unpam.ac.id.

Received: March 13, 2026; Accepted: April 10, 2026; Published: April 20, 2026.

Abstract: The increasing number of attacks on web applications necessitates strengthening secure programming competencies among computer science students. However, cybersecurity learning is often constrained by ethical and legal limitations, as direct testing on real-world systems is not permissible. This study designed and implemented a web-based cybersecurity training platform that provides a simulated vulnerability environment for secure programming practice. The methodology covers learning needs analysis, system design, vulnerability module implementation, and integration of defensive coding features. The platform operates as an online virtual laboratory accessible via www.kampuscyber.unaux.com, with modules addressing SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), File Upload Vulnerability, Insecure Direct Object Reference (IDOR), Command Injection, Directory Traversal, Weak Authentication, and Insecure Cookie handling. Each module maps programming errors directly to their security consequences, paired with defensive coding solutions. The evaluation involved 15 students enrolled in a cybersecurity training program. Across 10 modules, students achieved a 79.33% success rate in completing exploitation tasks and 65.33% in providing secure programming solutions — a gap that points to the greater difficulty of defensive over offensive competency. These findings indicate that the platform offers a safe and controlled environment for web vulnerability learning and mitigation practice, and may serve as an ethical alternative for practice-based secure programming education without exposing real-world systems to risk.

Keywords: Secure Programming; Cybersecurity; Vulnerability; Virtual Laboratory; Defensive Coding.

1. Introduction

Digital transformation has fundamentally reshaped how institutions operate, communicate, and deliver services. Across education, government, finance, and public administration, web-based applications have become the dominant infrastructure for data exchange and service delivery (Nelmiawati & Dealova, 2025; Muhammad *et al.*, 2023). The scale of this shift is difficult to overstate — billions of transactions, records, and interactions now pass through web systems daily, many of which were built under conditions that prioritized speed of deployment over security of design. The rapid expansion of web application use has not been matched by a corresponding maturation in secure development practices (Herman *et al.*, 2023), and that gap has consequences. Security, as a non-functional requirement, is routinely deferred to later stages of the

development lifecycle or omitted entirely — despite longstanding consensus that embedding it from the earliest phases of system design is both more effective and less costly than retrofitting it afterward (Sambhus, 2024).

The result is a persistent and well-documented vulnerability landscape. SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), weak authentication, and improper access control remain among the most frequently exploited weaknesses in web applications (Manuel *et al.*, 2025). These are not emerging threats — they have appeared on the OWASP Top Ten list for well over a decade. Yet they continue to be exploited at scale. Recent reports confirm that large-scale JavaScript injection campaigns have compromised more than 150,000 websites globally, a figure that reflects not the novelty of the attack technique but the durability of insecure coding habits among developers. In Indonesia, the problem carries additional local weight: cyber incidents targeting higher education institutions and public service platforms have exposed sensitive data and disrupted critical services, with Hariyadi *et al.* (2021) documenting that many Indonesian university information systems lack even basic security assessment mechanisms. Mahmud and Azim (2023) argue that SQL Injection vulnerabilities persist primarily because developers fail to implement parameterized queries — a countermeasure that is neither technically complex nor resource-intensive. Wiguna *et al.* (2020) reached a parallel conclusion regarding Web Application Firewall deployment, observing that the tools exist but are inconsistently applied due to gaps in developer competency. Taken together, these findings point to a systemic problem: the technical solutions are available, but the knowledge required to apply them is not being adequately developed during formal education.

Cybersecurity education faces a structural constraint that makes this problem difficult to address through conventional means. Direct experimentation on live systems — the most intuitive form of hands-on learning — raises immediate ethical and legal concerns, and is not permissible in educational settings. As a result, instruction tends to remain theoretical: students learn to describe vulnerability classes, trace attack vectors in diagrams, and identify mitigation strategies in the abstract, but rarely work through the full cycle of exploitation and remediation in a practical environment. Abikoye *et al.* (2020) observed that even relatively accessible mitigation techniques, such as string-matching algorithms for detecting injection patterns, are underutilized in practice because developers lack exposure to their implementation under realistic conditions. The gap between knowing that a vulnerability exists and knowing how to write code that prevents it is not trivial — and classroom instruction alone has not been sufficient to close it. What is missing is a controlled, ethically bounded environment in which students can observe the mechanics of exploitation, trace the programming errors that enable it, and practice the defensive coding responses that address it, all within a single structured learning experience.

Several platforms have attempted to fill this space — OWASP Security Shepherd, WebGoat, DVWA, and similar tools — but their coverage tends to be uneven, their pedagogical scaffolding limited, and their emphasis weighted toward exploitation rather than defense. Students can learn to attack; learning to fix is treated as secondary. To address this gap directly, the present study designed and built a web-based cybersecurity training platform that integrates vulnerability simulation with defensive coding practice across ten distinct vulnerability modules. The platform was developed and evaluated with the specific objective of improving students' understanding of web application security and their capacity to implement secure programming solutions in conditions that approximate real-world development contexts. The central proposition guiding this work is that simulation-based training — when it explicitly maps programming errors to their security consequences and pairs exploitation practice with structured mitigation guidance — can produce measurable improvement in both offensive and defensive competencies among undergraduate students, without exposing any real-world system to risk.

2. Related Work

Research on web application security education has grown considerably over the past several years, driven by the recognition that theoretical instruction alone is insufficient for developing practical security competencies. Several studies have proposed simulation-based approaches to address this gap, each contributing meaningfully to the field while leaving certain pedagogical limitations unresolved. A critical reading of these works reveals a consistent pattern: the further a platform moves from pure exploitation toward integrated defensive practice, the less developed its learning scaffolding tends to be. Wibowo and Sulaksono (2021) utilized the OWASP Security Shepherd platform to support hands-on learning of web vulnerabilities, with particular attention to Cross-Site Scripting (XSS) attacks. The platform operates through predefined challenges that expose users to attack mechanics in a structured sequence, and it succeeds in building familiarity with specific exploitation techniques. Its limitations, however, are equally clear. Coverage is concentrated on a narrow set of vulnerability types, and the learning trajectory is oriented almost entirely toward exploitation — secure coding practices receive little to no attention as a formal learning outcome. Students who complete the platform understand how an XSS attack works; whether they can write code that

prevents one is a separate question the platform does not adequately address. Idris *et al.* (2022) developed a web-based learning platform built around attack scenarios aligned with the OWASP API Security Project, targeting vulnerabilities specific to API services. Their approach improves users' ability to recognize and analyze API-level weaknesses through practice-oriented exercises, and the alignment with OWASP standards gives the platform credibility and coverage relevance. What it does not provide is structured guidance for implementing defensive coding responses — recognition and analysis are the terminal outcomes, and the transition from identifying a vulnerability to remediating it in code is left largely to the student. Tryhubets *et al.* (2024) took a different direction entirely, using intentionally vulnerable applications such as OWASP Juice Shop to benchmark the effectiveness of open-source and commercial vulnerability scanning tools. These applications create a realistic environment for tool evaluation, and the study contributes useful comparative data on scanner performance. The pedagogical dimension, however, is incidental rather than central — the connection between vulnerability identification and secure programming practice is not a design objective of the platform, and the learning structure required to support beginner-to-intermediate students is absent.

Taken together, these studies suggest that existing approaches to simulation-based cybersecurity education cluster into three recognizable categories: challenge-based learning platforms, attack scenario-based educational systems, and vulnerable applications designed primarily for security tool testing. Each category has produced genuine contributions, but the gaps across all three are consistent and worth naming directly. Most platforms weight exploitation over defense, treating mitigation as supplementary rather than central. The learning process rarely makes explicit the causal relationship between a specific programming error and its security consequence — students see the attack but not the code that enabled it. And the absence of guided learning paths limits effectiveness for students who are still building foundational competencies, which describes the majority of undergraduate learners in information systems and computer science programs. These are not minor gaps. A student who can exploit a SQL Injection vulnerability but cannot write a parameterized query has learned half the lesson — and arguably the less professionally useful half.

The platform developed in the present study was designed to address these limitations directly. Rather than treating exploitation and defense as separate activities, the platform integrates both within each module: students encounter vulnerable code, observe its exploitation, and then work through the corresponding defensive coding solution within the same learning environment. Three design principles guided this integration — first, multi-vulnerability simulation modules covering the most common web security threats; second, explicit mapping between programming errors and their security impacts; and third, embedded defensive coding guidance within each module rather than appended as an afterthought. The aim was a learning environment that treats secure programming not as a topic adjacent to vulnerability study, but as its necessary and inseparable complement.

3. Methodology

The growing frequency and complexity of attacks targeting web applications point consistently to the same underlying cause: programming errors that fail to account for security from the outset. Addressing these errors requires analysis that is both structured and systematic, moving from vulnerability identification through to the implementation of appropriate remediation measures (Hariyadi & Nastiti, 2021). This study adopted a Research and Development (R&D) approach, selected not because it is methodologically fashionable but because the research objective demanded it — the goal was not merely to generate conceptual findings but to produce a functional artifact in the form of a web-based cybersecurity training platform that functions as a practical learning medium for secure programming. Conceptual insight without a working system would have been insufficient. The developed product is a vulnerability simulation-based cybersecurity training platform, designed as a virtual laboratory that supports safe and controlled practice of web application security in an educational context. System development followed the System Development Life Cycle (SDLC) model, covering requirements analysis, system design, implementation, testing, and evaluation. This model was chosen for the disciplined development framework it provides — each stage builds on the previous one, and the resulting system remains traceable to the requirements that defined it. By combining R&D and SDLC, the study addressed both the technical construction of the platform and its effectiveness as a learning medium, treating these not as separate concerns but as mutually dependent ones.

3.1 Research Stages

The research stages represent a series of systematic activities carried out to achieve the research objectives in a structured and scientifically accountable manner. The development of this cybersecurity training platform proceeded through several main steps, as illustrated in Figure 1.

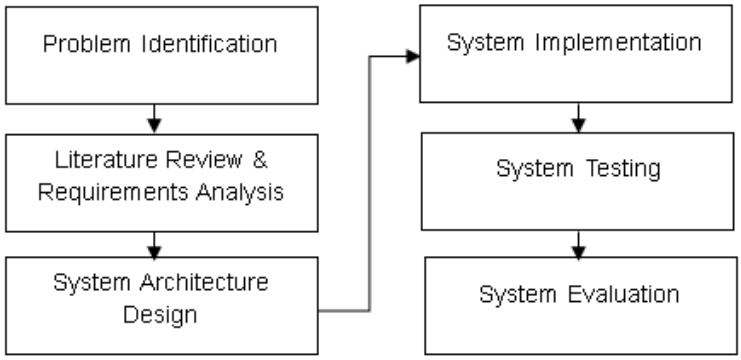


Figure 1. Research Stages

3.2 Problem Identification

The initial phase of this research involved systematically defining the research background, problem context, and urgency. Web-based application growth across education, government, finance, and public services has substantially increased exposure to cyber threats — and many of the security incidents that result are traceable to programming errors and insufficient implementation of secure coding practices during development. Preliminary observations confirmed that students in computer science programs frequently encounter difficulty applying secure coding principles in practical settings. Instruction in cybersecurity courses remains predominantly theoretical, and opportunities for hands-on practice are limited. Without a controlled, ethically bounded environment for experimenting with web vulnerabilities, students cannot engage directly with the mechanics of security failures — they can read about SQL Injection, but they cannot feel the consequence of an unsanitized input field. Current learning approaches also reveal a scarcity of practical media that simulate real-world web vulnerabilities in a structured manner. Practicing exploitation techniques on live systems raises immediate ethical and legal concerns, making such practice unsuitable for educational environments regardless of its instructional value. These conditions make the development of a dedicated practice-based learning environment not merely useful but necessary — one that simulates web application vulnerabilities in isolation from real-world systems, and that supports experiential learning in a structured, interactive, and responsible manner.

3.3 Literature Review and Requirements Analysis

The second stage combined literature review with system requirements analysis, aimed at establishing a theoretical foundation and translating it into concrete development specifications. The review examined scientific sources covering secure programming concepts, simulation-based cybersecurity training models, virtual laboratory environments, and web application vulnerability standards recommended by OWASP (Open Web Application Security Project). Findings from this review informed the overall platform design approach and ensured alignment with established cybersecurity education needs. Requirements analysis was then conducted across two dimensions. Functional requirements included vulnerability simulation modules, user authentication mechanisms, exploitation practice environments, defensive coding learning features, and evaluation mechanisms to support the learning process. Non-functional requirements addressed system security, usability, performance, and simulation environment isolation — ensuring that exploitation activities during learning produce no effect on external systems. Pedagogical considerations were incorporated throughout, so that the platform could integrate vulnerability exploitation and system defense within a single, coherent learning process rather than treating them as separate instructional activities.

3.4 System Architecture Design

The system design stage translated the requirements identified in the previous phase into both conceptual and technical designs. A web-based system architecture was developed, consisting of five main components — the application server, database, vulnerability simulation module, defensive coding module, and user interface — all designed to support integrated cybersecurity learning within a single structured platform. A defining feature of the design is the use of paired learning scenarios, where each module presents both a vulnerable code condition and a remediated code condition side by side. This pairing is not cosmetic. It makes the causal relationship between a programming error and its security consequence explicit and observable, enabling students to understand not only how a vulnerability is exploited but precisely what in the code allowed it to happen. User interface design and learning interaction flow were addressed at this stage to ensure that the platform remained usable and educationally effective, not merely technically functional.

3.5 System Implementation

The implementation stage transformed the system design into a functional, publicly accessible application. The platform was deployed as a web-based virtual laboratory at www.kampuscyber.unaux.com, accessible through a standard web browser without additional installation. Ten vulnerability modules were implemented, covering SQL Injection, Cross-Site Scripting in both Reflected and Stored forms, Cross-Site Request Forgery (CSRF), Insecure Direct Object Reference (IDOR), Command Injection, Directory Traversal, File Upload Vulnerability, Weak Authentication, and Insecure Cookie Handling. Each module allows users to perform controlled exploitation, analyze the impact of the vulnerability in question, and study prevention techniques through defensive coding practice. Internal testing was conducted throughout this phase to confirm that all features operate according to specifications and that no security risks extend beyond the boundaries of the simulation environment.

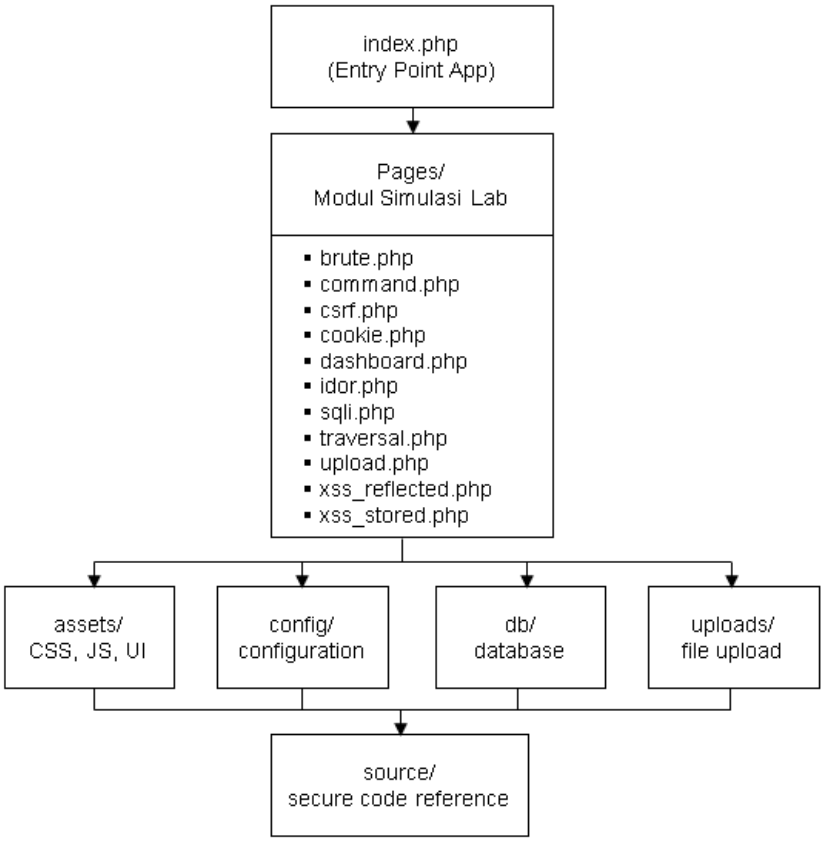


Figure 2. Application System Project Structure

As shown in Figure 2, the platform's implementation structure is organized modularly to separate configuration components, application logic, vulnerability simulation modules, and data storage. This separation improves development organization, facilitates maintenance, and supports the isolation of simulation modules from one another and from external systems. The main directory consists of six key components — assets, config, db, pages, source, and uploads. The assets directory stores interface resources including stylesheets, JavaScript files, and visual components that support the system interface. The config directory holds system configuration files including connection settings and security parameters. The db directory contains database management scripts and configurations. Core vulnerability simulation components reside in the pages directory, which houses modules for SQL Injection, Reflected and Stored XSS, CSRF, IDOR, Command Injection, Directory Traversal, and File Upload Vulnerability, each designed to present vulnerable code alongside learning mechanisms for understanding both exploitation and mitigation. The source directory stores secure code reference implementations for students to examine after working through exploitation scenarios. The uploads directory handles files submitted during simulation exercises such as File Upload Vulnerability testing. All user interactions are routed through index.php, which serves as the application's single entry point.

3.6 System Testing

System testing was conducted to confirm that all platform functions operate in accordance with the requirements and specifications defined in earlier stages. Black Box Testing was employed — a technique that evaluates system functionality based on observed inputs and outputs without examining the internal code structure. Testing covered three main areas, namely the vulnerability simulation modules themselves, user interaction mechanisms including authentication, navigation, and scenario execution, and the stability and

security of the simulation environment as a whole. This approach was appropriate given the platform's educational purpose — what matters for users is whether the system behaves correctly and safely, not how the underlying code achieves that behavior. Testing results confirmed stable operation across all modules, with no functional failures or unintended security exposures observed during the process.

3.7 System Evaluation

The evaluation stage assessed both the technical feasibility of the platform and its effectiveness as a learning medium for secure programming. Functional testing confirmed that all modules operate according to predefined requirements. Learning effectiveness was then evaluated by involving students as system users, using quantitative and qualitative approaches in combination. Data collection included pre-tests and post-tests to measure improvement in students' understanding of web application vulnerability concepts, usability questionnaires to assess the platform's usability level, and observation of practical activities to analyze interaction patterns with the simulation modules. The collected data were analyzed to determine the extent to which the platform improves students' understanding of the relationships between programming errors, vulnerability emergence, and applicable mitigation techniques — providing empirical evidence of the platform's contribution to effective and controlled cybersecurity practical learning.

3.8 Research Objects and Subjects

The object of this study is the web-based cybersecurity training platform developed as a virtual laboratory for secure programming education. The platform functions as a simulation environment enabling users to study web application vulnerabilities in a safe and controlled manner. Research subjects are students from the Information Systems Study Program at Universitas Pamulang who participated in system testing and evaluation activities. These students served as system users, executing vulnerability simulation modules and providing feedback on the platform's effectiveness and usability in supporting the learning process.

3.9 Classification and Relevance of Vulnerability Features

Vulnerability features in this study are organized according to categories of web application security threats that commonly appear in modern software development. The classification ensures that each simulation module represents a specific attack type while maintaining direct pedagogical relevance to secure programming competency development — the modules are not an arbitrary collection of threats but a structured set chosen to cover the most consequential vulnerability classes a developer is likely to encounter. The first category consists of injection-based vulnerabilities, covering SQL Injection and Command Injection. SQL Injection targets web application databases by injecting malicious SQL code to gain unauthorized access to sensitive data (Wiguna *et al.*, 2020). The vulnerability arises when user inputs are not properly validated or sanitized, allowing attackers to manipulate queries within database management systems (Mahmud & Azim, 2023). This is a critical area of study because it directly affects data integrity and confidentiality — attackers can insert malicious commands into login forms or URL parameters to bypass authentication or extract records (Hayati *et al.*, 2024; Natanael *et al.*, 2024). Detection approaches include log analysis, intrusion monitoring systems, and honeypot deployment (Abdullayev & Chauhan, 2023). Command Injection occurs when user input is processed and executed as a server-side system command without adequate restriction, potentially granting an attacker operating system-level access. This module addresses strict input validation, parameterized query use, and least-privilege execution principles as the foundational countermeasures. The second category covers scripting-based vulnerabilities, consisting of Reflected XSS and Stored XSS.

Reflected XSS occurs when user input is directly reflected in server responses without proper filtering, allowing malicious scripts to execute in the client's browser. Stored XSS involves malicious scripts being saved to a database and executed whenever the affected page is accessed by other users — a more persistent and potentially more damaging variant. Both forms provide a practical basis for explaining output encoding, input validation, and the separation of data from executable code in secure web application development. XSS and SQL Injection remain among the most frequently exploited attack types, and one mitigation approach involves comparing user input against stored injection pattern strings using string-matching algorithms (Abikoye *et al.*, 2020). The third category addresses access control vulnerabilities, covering Insecure Direct Object Reference (IDOR) and Weak Authentication. Insecure Direct Object Reference arises when systems provide direct access to objects such as files, user records, or transactions based on user-manipulable parameters without verifying authorization (Putra & Kautsar, 2023). A user who modifies an ID value in a URL can access another user's data — the system never checks whether the request is legitimate. Weak Authentication reflects deficiencies in authentication mechanisms such as simple passwords, absent login attempt limits, and the absence of multi-factor authentication, all of which invite brute-force attacks and credential stuffing. This module builds understanding of least-privilege principles and the necessity of consistent, server-side authorization checks for every access request.

The fourth category covers session and cookie management vulnerabilities, represented by Insecure Cookie Handling. When cookies lack the HttpOnly or Secure attributes, session information becomes accessible to client-side scripts or transmissible over unencrypted connections — conditions that enable session hijacking without requiring the attacker to know the user's credentials. The module addresses HTTPS use, proper attribute configuration through HttpOnly, Secure, and SameSite settings, and session ID regeneration after authentication as the primary countermeasures. The fifth category addresses file and system management vulnerabilities, covering File Upload Vulnerability and Directory Traversal. File Upload Vulnerability arises when systems accept uploaded files without validating type, enforcing size limits, or controlling execution rights, potentially allowing an attacker to upload a server-executable script. Directory Traversal occurs when path parameters are not validated, enabling access to files outside the permitted directory structure (Fazriani *et al.*, 2023; Maizi & Zainal, 2025; Yogi *et al.*, 2019). Both modules stress whitelist-based input validation and strict server-side access restrictions as the foundational mitigations. The final category is request-based vulnerability, represented by Cross-Site Request Forgery (CSRF). CSRF exploits the browser's automatic inclusion of session cookies in requests directed to the same domain, allowing attackers to trigger actions on behalf of authenticated users without their knowledge or consent (Ashari *et al.*, 2022). Mitigation requires anti-CSRF token implementation, verification of origin and referer headers, and SameSite cookie attribute configuration to prevent the execution of unauthorized requests.

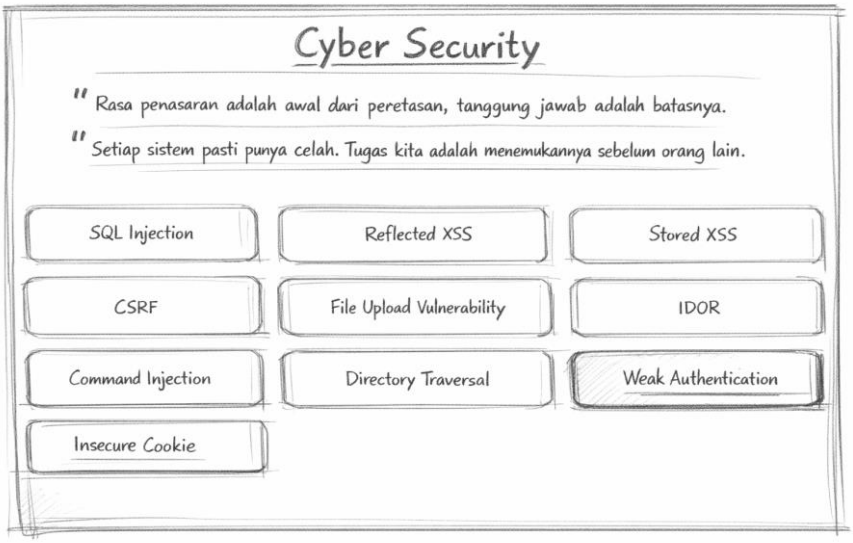


Figure 3. Application Feature Interface Sketch

3.10 Sampling and Participants

This study involved 15 students from the Information Systems Study Program at Universitas Pamulang, selected using purposive sampling based on prior programming experience and foundational knowledge of web development. The sample size, while modest, was appropriate for an evaluative study of this nature — the objective was to assess platform effectiveness in a controlled setting, not to produce population-level generalizations.

3.11 Evaluation Design and Data Analysis

Rather than a conventional pre-test and post-test design, this study evaluated students' performance through module-based assessment across all ten vulnerability simulation modules. Each module required students to complete two distinct tasks — identifying and exploiting the vulnerability to produce a Report score, and proposing an appropriate secure coding solution to produce a Solution score. Each component was scored on a scale of 0 to 10. Data were analyzed quantitatively using percentage-based performance metrics. The Report Score was calculated by dividing the total report score by 10 and multiplying by 100, while the Solution Score was calculated using the same formula applied to the total solution score. The overall average across both components was then determined to identify the general trend of students' competencies and to examine any systematic gap between exploitation ability and defensive coding capability — a distinction that carries direct implications for how cybersecurity curricula should be structured going forward.

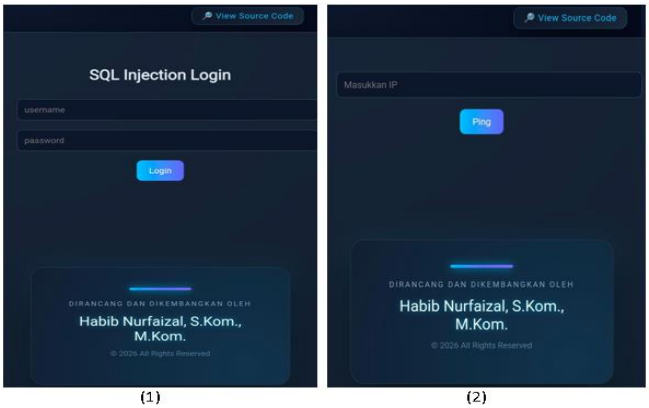
4. Result and Discussion

4.1 Results

Implementation of the web-based cybersecurity training platform was carried out after the requirements analysis and system design stages had been completed. The primary objective of this implementation was to provide an interactive learning medium through which students could understand, identify, and analyze various types of web application security vulnerabilities directly through controlled simulations. The system was designed around a hands-on learning approach, allowing students not only to engage with theoretical concepts but also to observe the impact of exploitation and the corresponding mitigation mechanisms in a single, integrated environment. Technically, the platform was developed using a web-based client-server architecture. The application runs on either a local server or web hosting environment and is accessible through a standard web browser without requiring additional installation. The backend utilizes PHP as the server-side scripting language and MySQL as the database management system to store user data, exploitation results, and learning activity logs. Each user interaction is processed through server-side input validation mechanisms to maintain data integrity and system stability during the simulation process. The main interface, illustrated in Figure 4, displays a homepage titled "Cyber Security," which includes an educational quote intended to promote ethical awareness in cybersecurity practice. On the main page, vulnerability modules are presented as systematically arranged interactive buttons. The main features implemented in the system are SQL Injection to simulate database query manipulation resulting from insufficient input validation, Reflected XSS to demonstrate the reflection of malicious user input within server responses, Stored XSS to simulate the storage of malicious scripts in a database that are executed when accessed by users, CSRF to illustrate the execution of unauthorized requests on behalf of an authenticated user, File Upload Vulnerability to demonstrate the risks associated with uploading files without proper type validation and execution restrictions, IDOR to illustrate weaknesses in access control mechanisms related to specific objects, Command Injection to simulate the execution of system commands through malicious user input, Directory Traversal to demonstrate unauthorized access to restricted system directories, Weak Authentication to illustrate weaknesses in authentication mechanisms such as the absence of login attempt limitations, and Insecure Cookie to demonstrate the risks associated with cookie configurations that lack adequate security attributes.

Figure 4. Dashboard View

4.1.1 Injection-Based Vulnerabilities



(1) (2)

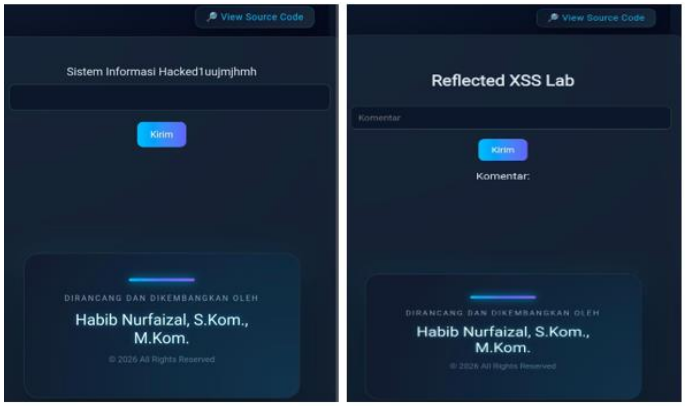
Figure 5. SQL and Command Injection

Figure 5 illustrates the category of injection-based vulnerabilities, which consists of two main types — SQL Injection and Command Injection. SQL Injection is shown as a vulnerability that arises due to the failure to properly validate and sanitize user input before it is processed within a database query. When input is not properly filtered, attackers can insert additional SQL commands to manipulate the structure of the query executed by the system. For example, an attacker may enter the following input into a login form: ' OR '1'='1. As a result, the query becomes:

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

All rows in the table satisfy the condition, allowing the system to bypass the authentication process without verifying the correct password. The impact may include unauthorized access to data, disclosure of sensitive information, modification of database contents, and deletion of data. This module emphasizes the importance of implementing proper input validation, the use of prepared statements or parameterized queries, and the application of secure coding principles when managing database interactions. Command Injection, by contrast, is a vulnerability that occurs when an application executes user input as part of a system command on the server side without adequate restrictions. This condition allows attackers to insert additional commands that the operating system then executes, potentially enabling access to sensitive files, modification of system configurations, or complete control of the server. This module addresses the importance of input validation and restriction, the use of secure functions, and the application of the principle of least privilege in executing commands within the server environment.

4.1.2 Scripting-Based Vulnerabilities



(3) (4)

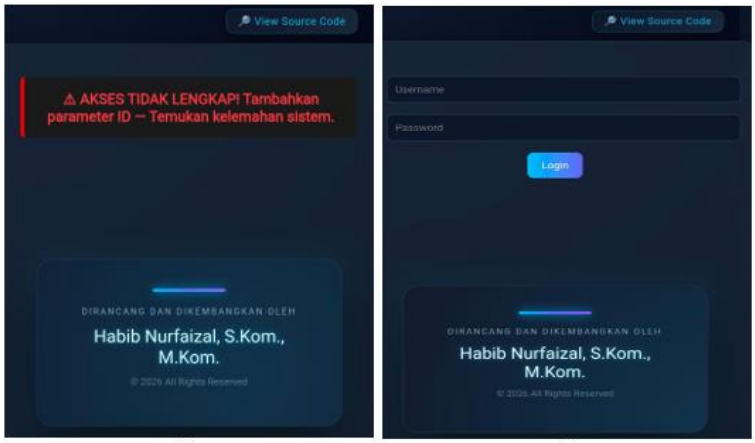
Figure 6. Stored and Reflected XSS

Figure 6 illustrates the category of scripting-based vulnerabilities, consisting of Stored XSS and Reflected XSS as two primary forms of Cross-Site Scripting attacks. Stored XSS occurs when a malicious script submitted by an attacker is permanently stored in the database or application storage media — comment fields, forums, or user profile sections. When other users access a page containing this stored data, the malicious script is automatically executed in the victim's browser. An example payload is as follows:

```
<script>document.location='http://attacker.com/steal.php?cookie=
'+document.cookie;</script>
```

The impact may include session cookie theft, account hijacking, and manipulation of page content. The primary cause of Stored XSS is the failure to perform input sanitization and output encoding properly. Reflected XSS, meanwhile, occurs when malicious user input is directly reflected by the server in the page response without adequate validation or filtering, though it is not permanently stored in the system. Exploitation is typically carried out through a modified URL sent to the victim — when the victim clicks the link, the malicious script executes in their browser. This module addresses the importance of input validation mechanisms, special character filtering, and output encoding to prevent malicious script injection.

4.1.3 Access Control Vulnerabilities



(5) (6)

Figure 7. IDOR and Weak Authentication

Figure 7 illustrates the category of access control vulnerabilities, consisting of IDOR and Weak Authentication as two forms of weaknesses related to system authentication and authorization mechanisms. IDOR occurs when an application provides direct access to an object — such as files, user data, or transactions — based on parameters that users can manipulate, without performing adequate authorization verification. A user may modify the value of an ID parameter in the URL to access data belonging to another user. This condition reflects a failure of the system to ensure that each access request has passed the appropriate access control validation process. The module addresses the importance of session-based authorization control, server-side access validation, and the use of indirect references or consistent access control checks. Weak Authentication describes a vulnerability that arises from deficient authentication mechanisms — simple passwords, absent password complexity policies, and no login attempt limitations. These weaknesses can be exploited through brute-force attacks, credential stuffing, or password guessing to gain unauthorized access. The module addresses the importance of strong authentication security policies, including strict password requirements, multi-factor authentication (MFA), login attempt limitations, and secure session management.
Figure 8. Brute Force Attack

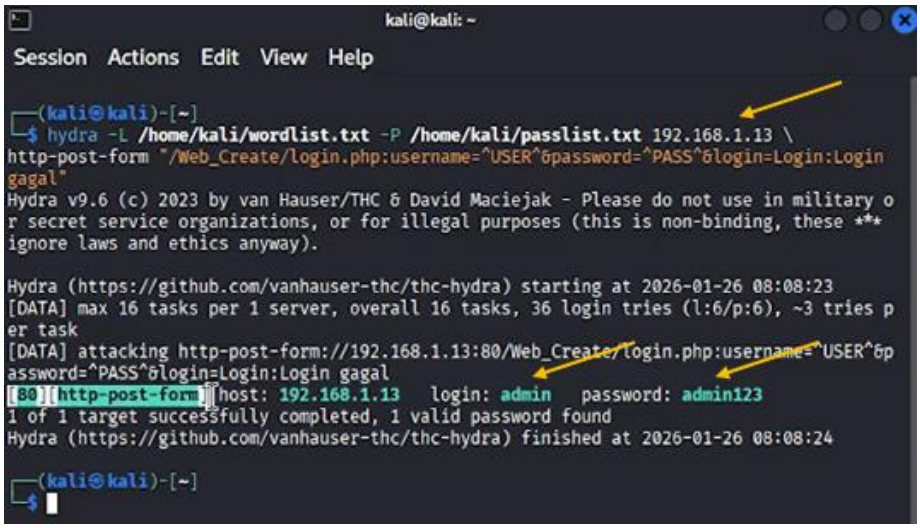


Figure 8 Brute Force Attack

4.1.4 Session and Cookie Management Vulnerabilities

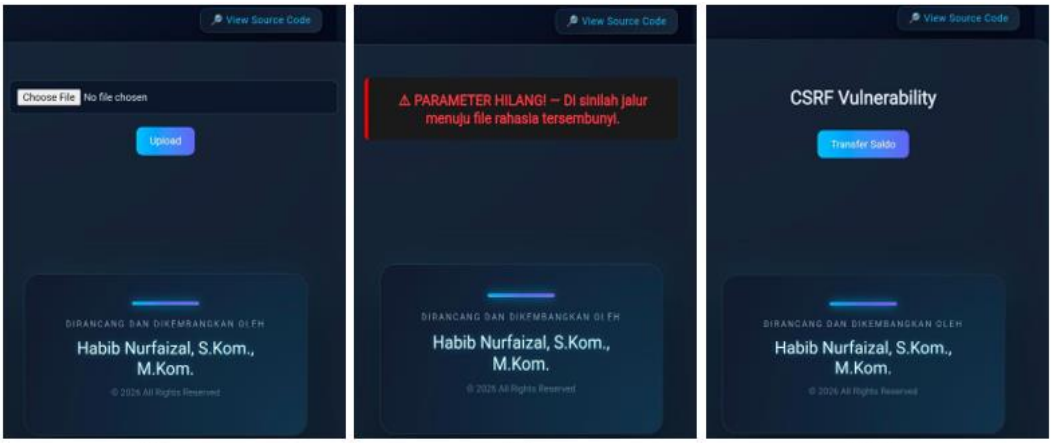
Figure 9 illustrates the category of session and cookie management vulnerabilities, focusing on Insecure Cookie as a weakness in user session management. This vulnerability occurs when a web application fails to configure session cookies with adequate security attributes — specifically, when the HttpOnly and Secure flags are not enabled. Without the HttpOnly attribute, cookies can be accessed through client-side scripts, for example through XSS attacks, thereby increasing the risk of session information theft. Without the Secure attribute, cookies may be transmitted over unencrypted HTTP connections, making them vulnerable to interception through man-in-the-middle attacks. This configuration weakness may lead to session hijacking,

in which an attacker uses stolen cookies to impersonate a legitimate user without needing to know the user's login credentials. The module addresses the importance of implementing secure session management practices — using the HTTPS protocol, configuring HttpOnly, Secure, and SameSite attributes, and applying session ID regeneration mechanisms after successful authentication. Securing cookies is therefore a fundamental component in maintaining the integrity and confidentiality of user sessions in web applications.

(7)

Figure 9. Insecure Cookie

4.1.5 File Management, System, and CSRF Vulnerabilities



(8) (9) (10)

Figure 10. File Management and Request-Based Vulnerabilities

Figure 10 classifies vulnerabilities into two main groups — file and system management vulnerabilities, and request-based and CSRF vulnerabilities — each encompassing critical security flaws in web application development. File Upload Vulnerability addresses the risks arising from applications that provide file upload features without adequate validation mechanisms, including file type verification, size restrictions, and control over access and execution rights. The absence of such controls allows attackers to upload malicious files such as server-side executable scripts, potentially leading to remote code execution, defacement, or system takeover. The module addresses the importance of extension and MIME type validation, storing files outside the public directory, and restricting access and execution rights at the server level. Directory Traversal occurs due to insufficient validation and normalization of path parameters or file names submitted by users. Attackers can manipulate input using special characters or specific patterns to access files or directories beyond the system's permitted scope, resulting in exposure of configuration files, authentication credentials, and other sensitive data. The module addresses whitelist-based access restrictions, path normalization, and strict server-side input validation as the primary countermeasures. CSRF enables unauthorized requests to be sent to the application on behalf of an authenticated user, because browsers automatically include session cookies in requests directed to the same domain. Attackers exploit this behavior to trigger actions without the user's knowledge or consent — data modification or transaction execution, for instance. The module addresses the implementation of anti-CSRF tokens, proper verification of HTTP methods, and SameSite cookie attribute configuration to prevent the execution of unauthorized requests.

4.1.6 System Test Results

System testing was conducted to confirm that all simulation modules functioned according to the design. Black Box Testing was used, evaluating functionality based on observed inputs and outputs without examining the program's internal code structure. Testing focused on module accessibility, the processing of exploit inputs, the system's responses to both valid and invalid inputs, and the stability of the simulation pages.

Table 1. Test Results of Vulnerability Simulation Modules

Module	Test Scenarios	Results	Information
SQL Injection	Input payload query manipulation	The system responds according to the scenario	[√] Success
Reflected XSS	Input script in URL parameters	Reflected script	[√] Success
Stored XSS	Input script in the comment form	Script saved and executed	[√] Success
CSRF	Request execution without token	Request processed	[√] Success
File Upload	Upload files without validation	The file is received by the server	[√] Success
IDOR	ID parameter manipulation	Other data can be accessed	[√] Success
Command Injection	Input system commands	The command is executed	[√] Success
Directory Traversal	Directory path manipulation	Files outside the directory can be accessed	[√] Success
Weak Authentication	Repeated login attempts	No restrictions	[√] Success
Insecure Cookie	Analyze cookie attributes	Cookies without HttpOnly/Secure	[√] Success

All modules operated according to the simulation design scenarios. During normal input testing, the system functioned stably without interruptions. In exploit testing, the system produced responses consistent with the characteristics of each vulnerability. No system errors or functional failures were observed throughout the testing process, and the testing achieved a 100% success rate across all design scenarios.

4.1.7 Student Performance Evaluation (Quantitative Results)

To assess the effectiveness of the developed platform as a learning medium, a performance-based evaluation was conducted involving 15 students. Each student completed 10 vulnerability simulation modules. The evaluation was divided into two components — Report, which measures students' ability to successfully identify and exploit vulnerabilities, and Solution, which assesses their ability to propose appropriate secure coding mitigation strategies. Detailed results are presented in Table 2.

Table 2. Student Performance Results

No	Name	Class	Report	Solution
1	Muhammad Luthfi Nurrahman	05SIFP003	7	7
2	Andini Azaria Zahra	01SIFP010	8	6
3	Muhammad Raihan Ramadhan	03SIFP001	9	6
4	Gazan Wildan Pratama	06SIFP007	7	5
5	Nisrina Difa Lediana	06SIFE004	9	6
6	Devan Haidar Wirya Hidayat	04SIFP016	8	7
7	Rizq Aulia Faisal	06SIFP010	7	5
8	Rijwan Maulana	04SIFE002	10	9
9	Muhammad Fauzan Jindan	01SIFP013	9	9
10	Ihsanudin Abdul Azis	03SIFE001	8	8
11	Sofi Amalia	03SIFE002	7	5
12	Allya Rohali	03SIFE001	7	4
13	Muhamad Sukardi	04SIFM004	6	6
14	Farid Nur Azis	04SIFM004	9	9
15	Kiki Amielia	06SIFM003	8	6

As shown in Table 2, the total Report score reached 119 and the total Solution score reached 98. The average Report score and Solution score were calculated using the following formulas:

$$\text{Report Score (\%)} = \frac{\text{Total Report Score}}{10} \times 100 = \frac{119}{15} \approx 7.93 \rightarrow 79.3\%$$

$$\text{Solution Score (\%)} = \frac{\text{Total Solution Score}}{10} \times 100 = \frac{98}{15} \approx 6.53 \rightarrow 65.3\%$$

The average Report score of 79.3% indicates that most students were able to successfully identify and exploit the vulnerabilities provided across the modules. The average Solution score of 65.3%, by contrast, indicates that students encountered considerably greater difficulty in formulating appropriate mitigation strategies. The gap between these two figures — approximately 14 percentage points — is not incidental. It reflects a pattern that appears consistently in cybersecurity education: students acquire offensive competencies more readily than defensive ones, partly because exploitation produces immediate, visible feedback while secure coding requires a more abstract understanding of what the code must prevent. This finding points to the need for stronger instructional emphasis on defensive programming and mitigation techniques within cybersecurity curricula.

4.2 Discussion

The implementation and testing results confirm that the developed platform provides a structured and interactive learning environment for web application security education. Each vulnerability module represents a real-world case commonly encountered in software development practice, keeping the content directly relevant to the goal of building secure programming competencies. The 100% functional success rate observed during Black Box Testing indicates that the simulation environment operates reliably — a necessary condition for any platform intended to support learning, since system instability would undermine students' ability to draw consistent conclusions from their exploitation exercises (Hariyadi & Nastiti, 2021). The interface design, as reflected in the mockups, keeps users focused on the vulnerability analysis process rather than on navigation overhead. Grouping features into modular buttons supports an experiment-based learning approach and allows students to move between vulnerability categories without losing the thread of their investigation. The ability to compare injection, scripting, access control, session management, and file management vulnerabilities within a single integrated platform is pedagogically significant — it enables students to recognize structural similarities across attack types rather than treating each vulnerability as an isolated phenomenon. This kind of cross-module pattern recognition is difficult to achieve with tools that address only one or two vulnerability classes (Wibowo & Sulaksono, 2021; Idris *et al.*, 2022).

The quantitative results reveal a meaningful gap between students' exploitation performance and their secure coding performance — 79.3% versus 65.3% respectively. This finding is consistent with observations in the broader cybersecurity education literature. Abikoye *et al.* (2020) noted that mitigation techniques are underutilized in practice not because they are technically inaccessible but because developers lack structured exposure to their implementation. The present results suggest a similar dynamic at the student level — exploitation is concrete and produces visible system responses, while defensive coding requires students to reason about what the code must prevent rather than what it currently does. Mahmud and Azim (2023) made a comparable observation regarding SQL Injection, arguing that the persistence of this vulnerability in production systems reflects a gap in developer training rather than a gap in available solutions. The platform's paired module design — presenting vulnerable code alongside its remediated counterpart — directly addresses this gap, though the Solution score of 65.3% suggests that the pairing alone is not sufficient and that more guided instruction on defensive coding reasoning may be warranted.

From a pedagogical standpoint, the controlled simulation approach builds both conceptual and practical understanding simultaneously. Students do not merely observe theoretical constructs — they work through the relationship between a specific programming error and its exploitation consequence, then engage with the corresponding mitigation within the same session. The integration of mitigation discussions within each module reinforces a defensive mindset and supports the principle of being secure by design, which Sambhus (2024) identifies as a foundational orientation for secure software development. What the results also suggest, however, is that the transition from understanding a mitigation to implementing it correctly remains a distinct cognitive step — one that the platform supports but does not fully resolve. Future iterations of the platform would benefit from incorporating more scaffolded guidance at the Solution stage, potentially through automated feedback mechanisms or structured code review prompts that help students identify precisely where their defensive coding falls short.

5. Conclusion and Recommendations

This study successfully designed and implemented a cybersecurity training platform based on vulnerability simulation as a structured and interactive medium for secure programming education. The platform was developed using a web-based client-server architecture and equipped with simulation modules covering SQL Injection, Cross-Site Scripting in both Reflected and Stored forms, Cross-Site Request Forgery (CSRF), File Upload Vulnerability, Insecure Direct Object Reference (IDOR), Command Injection, Directory Traversal, Weak Authentication, and Insecure Cookie. Implementation results indicate that the system provides a stable and responsive learning environment accessible through a standard web browser without additional installation. Black Box Testing confirmed that all modules operated according to the design scenarios, achieving a 100% success rate in functional testing. Each simulation feature presented vulnerability characteristics in a controlled manner, supporting users in the exploration, analysis, and understanding of mitigation mechanisms.

The study employed a quantitative approach using purposive sampling, involving 15 students with prior knowledge of programming and web development. Performance-based assessment across 10 simulation modules measured both vulnerability exploitation capability through the Report component and mitigation capability through the Solution component. Students achieved higher performance in identifying and exploiting vulnerabilities than in formulating secure coding solutions, indicating a gap between offensive and defensive competencies that warrants further attention in cybersecurity curriculum design. The main academic contribution of this study lies in the integration of hands-on simulation with a structured learning model that connects exploitation and prevention within a single platform. This approach deepens students' understanding of software security principles and builds a defensive mindset — the orientation of being secure by design — during application development. The platform therefore demonstrates feasibility as an educational tool for web application security instruction in higher education. Future development should consider incorporating automated assessment features, integrating learning analytics, and conducting broader evaluations to measure students' skill improvement with greater empirical precision.

References

Abdullayev, V., & Chauhan, A. S. (2023). SQL injection attack: Quick view. *Mesopotamian journal of Cybersecurity*, *2023*, 30-34. https://doi.org/10.58496/MJCS/2023/006.

Abikoye, O. C., Abubakar, A., Dokoro, A. H., & Akande, O. N. (2020). A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm. *EURASIP Journal on Information Security*. https://doi.org/10.1186/s13635-020-00112-z

Ashari, I. F., Oktariana, V., Sadewo, R. G., & Damanhuri, S. (2022). Analysis of cross site request forgery (CSRF) attacks on West Lampung Regency websites using OWASP ZAP tools. *Jurnal Informatika*, *11*, 276–281. https://doi.org/10.32736/sisfokom.v11i2.1393

Fazriani, N. I. S., Cut, B., & Sanusi. (2023). Uji keamanan website terhadap serangan path traversal (studi kasus website pendapatan warga). *Jurnal Ristech (Jurnal Riset, Sains dan Teknologi)*, *4*(1), 60–66.

Hariyadi, D., & Nastiti, F. E. (2021). Analisis keamanan sistem informasi menggunakan Sudomy dan OWASP ZAP di Universitas Duta Bangsa Surakarta. *Jurnal Informatika dan Rekayasa Perangkat Lunak*, *5*(1), 35–42. https://doi.org/10.31603/komtika.v5i1.5134

Hayati, F., Nizar, M., Bana, S., Anugrah, T., & Huda, M. Q. (2024). Penetration testing keamanan website STIE Samarinda menggunakan teknik SQL injection dan XSS. *Jurnal Ilmu Komputer dan Sistem Informasi*, *12*(1), 618–624. https://doi.org/10.23960/jitet.v12i1.3882

Herman, Riadi, I., Kurniawan, Y., & Rafiq, I. A. (2023). Analisis keamanan website menggunakan Information System Security Assessment Framework (ISSAF). *Jurnal Keamanan Informasi*, *9*(1), 126–136.

Idris, M., Syarif, I., & Winarno, I. (2022). Web application security education platform based on OWASP API Security Project. *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, *10*(2), 246–261. https://doi.org/10.24003/emitter.v10i2.705

Maizi, Z., & Zainal, Z. (2025). Analisis log akses server web untuk mendeteksi anomali dan serangan siber menggunakan metode kuantitatif dan kualitatif. *Jurnal Teknologi dan Sistem Informasi*, *8*(3), 1741–1747.

Mahmud, S. M. S., & Azim, M. A. (2023). SQL injection attack vulnerabilities of web application and detection. *International Journal of Computer Applications*, *185*(38), 41–48. https://doi.org/10.5120/ijca2023922829

Manuel, J. K., & Kurniati, R. (2025). Analisis keamanan website E-Pinter terhadap serangan SQL injection dan XSS. *Jurnal Keamanan Siber*, *4*(September), 46–60.

Muhammad, H. H., Hadiana, A. I., & Ashaury, H. (2023). Pengamanan aplikasi web dari serangan SQL injection dan cross site scripting menggunakan web application firewall. *Jurnal Teknik Informatika*, *7*(5), 3265–3273. https://doi.org/10.36040/jati.v7i5.7320

Natanael, Y., Felicia, R., Malays, E., & Sakti, S. (2024). Analisis keamanan informasi bagi pengguna menggunakan Kali Linux melalui teknik SQL injection website. *TEKINFO*, *25*(1), 123–132.

Nelmiawati, & Dealova, K. (2025). Analysis of polyglot obfuscation techniques against ModSecurity in preventing cross-site scripting (XSS) and SQL injection attacks with experimental method. *Jurnal Teknik Informatika (JUTIF)*, *6*(4), 2540–2549. https://doi.org/10.52436/1.jutif.2025.6.4.5000

Putra, R. A., & Kautsar, I. A. (2023). Detection and prevention of insecure direct object references (IDOR) in website-based applications. *Procedia of Engineering and Life Science*, *4*(June). https://doi.org/10.21070/pels.v4i0.1388

Sambhus, K. (2024). Automating SQL injection and cross-site scripting vulnerability remediation in code. *Software*, *3*(1), 28–46. https://doi.org/10.3390/software3010002

Tryhubets, B., Tryhubets, M., & Zagorodna, N. (2024). Analysis of the efficiency of open source and commercial vulnerability scanners for e-commerce web application. *Scientific Journal of the Ternopil National Technical University*, *4*(116), 23–30.

Wibowo, R. M., & Sulaksono, A. (2021). Web vulnerability through Cross Site Scripting (XSS) detection with OWASP security shepherd. *Indonesian Journal of Information Systems*, *3*(2), 149-159. https://doi.org/10.24002/ijis.v3i2.4192.

Wiguna, B., Prabowo, W. A., & Ananda, R. (2020). Implementasi web application firewall dalam mencegah serangan SQL injection pada website. *Jurnal Teknologi Informasi & Komunikasi*, *11*(2), 245–256.

Yogi, Ruslianto, I., & Bahri, S. (2019). Analisa log web server untuk mengetahui pola perilaku website menggunakan teknik regular expressions. *Jurnal Komputer dan Aplikasi*, *7*(1), 120–130. https://doi.org/10.26418/coding.v7i01.32692.