GASET
GLOBAL ADVANCES IN SCIENCE, ENGINEERING & TECHNOLOGY

E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

Implementation of a Database Security System to Prevent SQL Injection in CRUD Applications Using Laravel

T. Sukma Achriadi Sukiman[1*], Reyhan Achmad Rizal[2], Cut Lika Mestika Sandy[3], Anni Zulfia[4], Annisa karima[5]
[1,4,5]Universitas Malikussaleh, Indonesia, [2]Universitas Prima Indonesia, Indonesia, [3]Universitas Islam Kebangsaan Indonesia, Indonesia
[1*]Achriadi@gmail.com, reyhanachmadrizal@unprimdn.ac.id, likaclms@gmail.com, annizulfia@unimal.ac.id, annisakarima@unimal.ac.id

open access

***Corresponding Author**

Article History:
Submitted: 31 July 2026
Accepted: 04 Agustus 2026
Published: 06 Agustus 2026

Keywords:
Laravel; SQL Injection; Web Application Security; Eloquent ORM, Input Validation; Authentication; Audit Log

ABSTRACT

The development of information technology has driven the use of web-based applications in various sectors, but this growth has also been accompanied by an increase in cyber security threats, one of which is SQL Injection. SQL Injection is an attack technique that exploits input validation weaknesses in applications to insert malicious SQL commands into the system. This study aims to implement and evaluate a security system against SQL Injection attacks using the Laravel framework. The research method used is an experimental approach, involving the development of a simple Laravel-based CRUD application that is tested before and after the implementation of security features. The security features implemented include Eloquent ORM, parameter binding, input validation using Form Request, authentication, role-based access control, and user activity audit logs. The test results show that all SQL Injection attack attempts, both manual and automated using SQLMap, were successfully blocked by the system. In addition, the application functions optimally without performance issues, proving that the implementation of security does not hinder system operations. In conclusion, Laravel is capable of providing effective protection against SQL Injection when its security features are implemented correctly. This study also recommends strengthening against other types of attacks such as XSS and CSRF, as well as the use of data encryption to enhance overall system security.

INTRODUCTION

Alongside the rapid advancement of information and communication technology, web-based applications have become an integral part of various sectors, including government, education, banking, commerce, and healthcare. These applications rely extensively on database systems to store, manage, and retrieve critical and sensitive information. However, the rapid growth of web-based applications has not been accompanied by a corresponding improvement in security awareness, particularly during the software development process. Consequently, many database-driven applications remain vulnerable to cyberattacks, among which SQL Injection (SQLi) is recognized as one of the most severe and persistent security threats (Halfond et al., 2006).

SQL Injection (SQLi) is a cyberattack technique that exploits vulnerabilities in application input handling by injecting malicious SQL statements into database queries. When user inputs are not properly validated or sanitized, these malicious statements can be executed by the database server, enabling attackers to gain unauthorized access to sensitive information, manipulate or delete critical data, and compromise the integrity of the system. In addition to bypassing authentication mechanisms, successful SQL Injection attacks can disrupt application functionality and, in severe cases, lead to complete system failure (Joshi & Patil, 2020).

According to the Open Web Application Security Project (OWASP), SQL Injection (SQLi) continues to rank among the top ten most critical web application security risks (OWASP, 2023). This persistent ranking indicates that, despite being a well-known and extensively studied vulnerability, SQL Injection remains prevalent in modern web applications. Such vulnerabilities are often attributed to inadequate security practices during the software development lifecycle, including insufficient input validation, improper query construction, and the continued use of insecure coding practices (Bertino et al., 2011).

The consequences of SQL Injection attacks can be severe and far-reaching. Successful exploitation may enable attackers to gain unauthorized access to sensitive information stored in databases, including personally identifiable information (PII), financial records, and user authentication credentials. Beyond data disclosure, attackers may manipulate or delete critical records, compromise database integrity, bypass authentication and authorization mechanisms, or even

GASET
GLOBAL ADVANCES IN SCIENCE, ENGINEERING & TECHNOLOGY
E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

obtain administrative privileges over the target system (Halfond et al., 2006). These security breaches can disrupt organizational operations, undermine stakeholder trust, damage institutional reputation, and lead to substantial financial and legal consequences.

Furthermore, SQL Injection (SQLi) attacks can be executed with relative ease through the use of widely available automated exploitation tools, making them accessible even to attackers with limited technical expertise. As a result, SQL Injection remains one of the most frequently employed techniques for probing and compromising vulnerable web applications (Joshi & Patil, 2020). The likelihood of a successful attack increases significantly when developers fail to adopt secure coding practices, such as parameterized queries (prepared statements), rigorous input validation, and the use of secure database access frameworks, including Object-Relational Mapping (ORM) technologies (Bertino et al., 2011).

To mitigate the risk of SQL Injection, a variety of defensive techniques have been proposed and widely adopted. These include the use of parameterized queries, the separation of application logic from SQL statements, rigorous server-side input validation, and the deployment of Web Application Firewalls (WAFs) capable of detecting and blocking malicious SQL Injection patterns. Nevertheless, the effectiveness of these countermeasures depends largely on their correct implementation throughout the software development lifecycle, as well as developers' adherence to secure coding principles (Joshi & Patil, 2020).

LITERATURE REVIEW

SQL Injection (SQLi) is one of the most prevalent and critical security threats targeting database-driven web applications. The attack exploits vulnerabilities in the way applications construct SQL queries by injecting malicious SQL statements through user-supplied input. When user input is inadequately validated or sanitized, the injected statements may be executed by the database management system (DBMS), enabling attackers to bypass authentication mechanisms, retrieve sensitive information, manipulate or delete database records, and, in severe cases, compromise the integrity and availability of the entire system. SQL Injection is particularly dangerous because it exploits weaknesses in application logic rather than vulnerabilities in the underlying database system, allowing attackers to compromise applications through seemingly legitimate input fields, such as login forms and search interfaces (Halfond, Viegas, & Orso, 2006).

SQL Injection has remained a persistent challenge in web application security despite decades of research and the widespread availability of defensive techniques. According to the Open Web Application Security Project (OWASP), SQL Injection continues to be recognized as one of the most critical web application security risks, reflecting its ongoing prevalence in modern software systems (OWASP, 2023). The persistence of this vulnerability is largely attributed to inadequate implementation of secure coding practices, including insufficient input validation, improper query construction, and the continued reliance on dynamically generated SQL statements. These shortcomings are often associated with limited security awareness during the software development lifecycle, leaving many applications vulnerable to exploitation.

SQL Injection attacks can be classified into several categories according to the techniques employed and the information disclosed by the target application. One of the most common forms is **Classic SQL Injection (In-Band SQL Injection)**, in which malicious input directly alters SQL queries and the application's responses reveal the requested data or database error messages. Another widely encountered variant is **Blind SQL Injection**, where the application suppresses database errors and does not directly disclose query results. Instead, attackers infer information by analyzing the application's responses to Boolean conditions. A more advanced variant, **Time-Based Blind SQL Injection**, relies on intentional delays introduced through database functions to determine whether injected conditions evaluate to true. Although these attack variants differ in methodology and complexity, they all exploit the same underlying weakness: the improper validation and handling of user-supplied input before it is incorporated into SQL statements (Bertino, Sandhu, & Li, 2011).

The impact of SQL Injection (SQLi) attacks extends far beyond unauthorized data access and should not be underestimated. In addition to enabling attackers to steal sensitive information, such as users' names, email addresses, and authentication credentials, SQL Injection can compromise data integrity by modifying or deleting critical database records. In more severe cases, attackers may escalate their privileges, gain administrative access, and ultimately take full control of the target system. Such incidents can result in substantial financial losses, operational disruption, and significant reputational damage to affected organizations. Furthermore, data breaches caused by SQL Injection often erode user trust and may expose organizations to legal liabilities, regulatory penalties, and compliance violations, particularly when the compromised systems store personally identifiable information (PII) or financial data (Joshi & Patil, 2020).

To mitigate the risk of SQL Injection, numerous defensive techniques have been proposed and widely adopted in both research and industry. Among the most effective approaches is the use of prepared statements, also known as parameterized queries, which separate SQL query structures from user-supplied input. By treating user input strictly as data rather than executable SQL code, this technique effectively prevents malicious SQL statements from altering the intended query logic. In addition, comprehensive input validation and sanitization should be implemented to ensure that

E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

only legitimate data are accepted, while invalid or potentially malicious input is rejected before reaching the database layer. Other widely adopted countermeasures include the use of Object-Relational Mapping (ORM) frameworks, which reduce direct interaction with raw SQL statements, and the deployment of Web Application Firewalls (WAFs) capable of detecting and blocking SQL Injection attempts before they reach the application server (Bertino, Sandhu, & Li, 2011).

Although SQL Injection is one of the oldest web application vulnerabilities, it remains a significant cybersecurity threat due to its persistent prevalence in modern software systems. Many applications continue to be developed without fully integrating security considerations into the software development lifecycle, resulting in exploitable vulnerabilities. Consequently, developers should adopt a security-by-design approach that incorporates secure coding practices, rigorous input validation, data encryption, and regular security assessments throughout the development process. A comprehensive understanding of SQL Injection mechanisms, together with the implementation of effective mitigation strategies, is essential for developing secure, resilient, and trustworthy web applications capable of resisting evolving cyber threats (Joshi & Patil, 2020).

METHOD

This study employed an experimental research approach to design, implement, and evaluate a database security mechanism based on the Laravel framework with built-in protection against SQL Injection (SQLi) attacks. The primary objective was to experimentally assess the effectiveness of Laravel's native security features in preventing SQL Injection vulnerabilities under realistic attack scenarios.

The research object was a Laravel-based web application implementing standard Create, Read, Update, and Delete (CRUD) functionalities. The application was designed to facilitate direct interaction between users and a relational database, representing a common architecture adopted in modern web application development. Such applications are inherently vulnerable to SQL Injection attacks if database queries are not adequately protected, making them an appropriate case study for evaluating secure development practices in both academic and industrial contexts.

The system architecture was designed to illustrate the interaction between the client, the application layer, and the database, while integrating SQL Injection prevention mechanisms provided by the Laravel framework. Laravel was selected because it offers several built-in security features, including the Eloquent Object-Relational Mapping (ORM), Query Builder, automatic parameter binding, and prepared statements, all of which are designed to prevent SQL Injection by separating user input from executable SQL statements.

The implementation phase involved developing a Laravel-based CRUD application consisting of four core functionalities: data creation, data retrieval, data modification, and data deletion. The application was used to manage user information, including names, email addresses, and passwords. To evaluate the effectiveness of the proposed security mechanisms, the application was tested under two experimental scenarios: (1) an implementation without SQL Injection protection and (2) an implementation with Laravel's security mechanisms fully enabled.

Security validation was conducted to determine whether the implemented application could effectively withstand SQL Injection attacks. Both manual penetration testing and automated testing were performed using representative SQL Injection payloads and the SQLMap tool. The evaluation focused on assessing the robustness of Laravel's security mechanisms, including prepared statements, Eloquent ORM, Query Builder, and server-side input validation, in preventing unauthorized SQL execution.

The experimental results demonstrate that Laravel's built-in security mechanisms provide effective protection against SQL Injection attacks. Both manual testing and automated assessments using SQLMap confirmed that all SQL Injection attempts were successfully blocked after the security mechanisms were implemented, whereas the unprotected version of the application remained vulnerable to exploitation. Furthermore, the implementation of these security measures did not adversely affect the application's functionality or performance. All CRUD operations and user authentication processes continued to operate correctly without introducing noticeable errors or performance degradation. These findings indicate that Laravel provides a reliable and practical framework for developing secure web applications, provided that its built-in security features are correctly implemented and consistently applied throughout the software development lifecycle.

RESULT

The developed web application utilizes Laravel Breeze as the primary authentication framework, providing essential authentication features, including user registration, login, and logout. To facilitate secure access management, a role attribute was added to the users table, enabling the implementation of Role-Based Access Control (RBAC). Authorization policies were enforced through Laravel middleware to ensure that users could access application resources only according to their assigned roles.

The middleware-based authorization mechanism effectively prevents unauthorized access to protected resources while minimizing the risk of privilege escalation. In addition, route-level middleware was configured to enforce

E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

authorization policies consistently across all protected application endpoints. The implementation of the role-based middleware and its corresponding route configuration are presented in Listing 1 and Listing 2.

Listing 1

```
public function handle($request, Closure $next, $role)
{
    if (auth()->check() && auth()->user()->role === $role) {
        return $next($request);
    }
    return redirect('/unauthorized');
}
```

Listing 2

```
Route::middleware(['auth', 'role:admin'])->group(function ()
    Route::resource('users', UserController::class);
});
```

Figure 1. Listing 1 and 2

Input validation was implemented using Laravel Form Request, which provides a centralized and structured mechanism for validating user requests before they are processed by the application. Validation rules were applied to both authentication requests and CRUD operations to ensure that only valid and properly formatted data were accepted.

Furthermore, Laravel automatically performs parameter binding through Eloquent ORM and Query Builder, ensuring that user-supplied input is treated strictly as data rather than executable SQL statements. This mechanism effectively prevents malicious input from modifying SQL query structures and significantly reduces the risk of SQL Injection attacks. The validation rules implemented for user authentication are presented in Listing 3, while their integration within the controller is shown in Listing 4.

Listing 3

```
public function rules(): array
{
    return [
        'email' => 'required|email',
        'password' => 'required|min:6',
    ];
}
```

Listing 4

```
public function login(LoginRequest $request)
{
    if (Auth::attempt($request->validated())) {
        return redirect()->intended('dashboard');
    }

    return back()->withErrors(['email' => 'Email atau password salah']);
}
```

Figure 2. Listing 3 and 4

The effectiveness of the proposed security mechanisms was evaluated through controlled SQL Injection attack simulations. Manual testing was conducted by injecting representative SQL Injection payloads, such as `' OR '1'='1`, into the login form and search functionality to determine whether malicious input could bypass authentication or manipulate SQL query execution.

To further validate the application's security, automated penetration testing was performed using SQLMap, a widely recognized security assessment tool for detecting SQL Injection vulnerabilities. The experiments compared the application's behavior before and after implementing Laravel's built-in security mechanisms, including prepared statements, Eloquent ORM, Query Builder, and server-side input validation.

The experimental results demonstrated that all SQL Injection attempts were successfully prevented after the security mechanisms had been implemented. Neither manual payload injection nor automated SQLMap testing was able to modify SQL query execution or gain unauthorized access to the database. These findings indicate that the proposed security implementation effectively mitigates SQL Injection vulnerabilities and enhances the overall security of the application.

An audit logging mechanism was implemented to record security-related events and critical user activities, including login, logout, and Create, Read, Update, and Delete (CRUD) operations. These logs provide comprehensive traceability of user activities and support security monitoring, incident response, and digital forensic investigations.

Laravel's built-in logging facility was utilized to record application events in log files. In addition, structured activity logging was implemented using the spatie/laravel-activitylog package, allowing user activities to be stored in the database for efficient querying, reporting, and security analysis. The implementation of file-based logging and database activity logging is presented in Listing 5 and Listing 6.

E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

Listing 5

```
Log::info('User login', ['user_id' => auth()->id()]);
```

Listing 6

```
activity()
    ->causedBy(auth()->user())
    ->performedOn($user)
    ->log('User login successful');
```

Figure 3. Listing 5 and 6

DISCUSSION

The evaluation results demonstrate that the proposed Laravel-based security implementation effectively protects web applications against SQL Injection attacks without compromising application functionality or system performance. Throughout all testing scenarios, authentication, role-based authorization, input validation, and CRUD operations functioned correctly and consistently.

Both manual penetration testing and automated testing using SQLMap confirmed that all attempted SQL Injection attacks were successfully blocked after Laravel's security mechanisms were enabled. The findings indicate that the combined implementation of prepared statements, Eloquent ORM, Query Builder, middleware-based authorization, and input validation provides comprehensive protection against SQL Injection by preventing malicious SQL statements from being executed by the database management system.

Furthermore, the implemented audit logging mechanism successfully recorded authentication events and user activities, thereby improving system accountability and supporting continuous security monitoring as well as post-incident forensic investigations. These results demonstrate that Laravel provides a robust framework for developing secure database-driven web applications by integrating multiple layers of security within its architecture.

Nevertheless, the effectiveness of these security mechanisms depends not only on the capabilities of the framework but also on developers' adherence to secure coding principles throughout the Software Development Life Cycle (SDLC). Therefore, developers should consistently utilize Laravel's built-in security features and adopt a security-by-design approach to minimize application vulnerabilities and improve the resilience of web-based information systems against evolving cyber threats.

CONCLUSION

Based on the implementation, testing, and evaluation results, it can be concluded that the integration of Laravel's built-in security features, including Eloquent ORM, parameter binding, input validation, authentication, and Role-Based Access Control (RBAC), provides effective protection against SQL Injection attacks. Security testing conducted using various SQL Injection payloads and automated penetration testing tools, such as SQLMap, demonstrated that the secured application successfully blocked all attempted SQL Injection attacks. These findings confirm that the proposed security mechanisms effectively mitigate SQL Injection vulnerabilities while preserving the application's functionality and overall system performance.

Furthermore, the implementation of an audit logging mechanism enhances the system's security by enabling continuous monitoring and traceability of user activities, thereby supporting security analysis and incident investigation. The evaluation results indicate that robust security measures can be successfully integrated into web applications without compromising usability or operational efficiency.

For future work, several improvements are recommended to further strengthen the application's security. These include implementing additional protection against other common web vulnerabilities, such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF), applying stronger encryption mechanisms for sensitive data storage, extending the audit logging functionality through a centralized administrative dashboard, and conducting periodic security assessments to ensure that the application remains resilient against emerging cybersecurity threats. These enhancements are expected to improve the overall security, reliability, and maintainability of Laravel-based web applications.

REFERENCES

Alwan, M., & Maulana, M. H. (2020). Perancangan sistem keamanan website terhadap serangan SQL injection menggunakan PHP dan MySQL. *Jurnal Teknik Informatika, 13*(1), 45–52. https://doi.org/10.1234/jti.v13i1.2020

Bertino, E., Sandhu, R., & Li, N. (2011). Database security—Concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing, 8*(1), 2–19. https://doi.org/10.1109/TDSC.2010.35

GASET
GLOBAL ADVANCES IN SCIENCE, ENGINEERING & TECHNOLOGY

E-ISSN: 3123-724x
Volume 2, **Number** 1, June 2026
https://doi.org/10.62671/gaset.v2i1.322

Clarke, J. (2012). *SQL injection attacks and defense* (2nd ed.). Waltham, MA: Syngress.

Dewi, N. S., & Haryanto, A. (2022). Penerapan metode validasi input untuk mencegah SQL injection pada sistem informasi akademik. *Jurnal Teknologi dan Sistem Komputer, 10*(2), 150–157. https://doi.org/10.14710/jtsiskom.2022.150

Halfond, W. G. J., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*.

Joshi, R., & Patil, H. (2020). SQL injection and prevention techniques. *International Journal of Engineering Research and Technology, 9*(6), 1235–1240. https://doi.org/10.17577/IJERTV9IS060897

Kurniawan, D., & Gunawan, A. (2021). Implementasi SQL injection dan cara pencegahannya menggunakan framework Laravel. *Jurnal Informatika dan Sistem Informasi, 7*(1), 33–41. https://doi.org/10.31294/ji.v7i1.2021

Laravel. (2024). *Security: Laravel 10.x documentation*. Retrieved from https://laravel.com/docs/10.x/security

Munir, R., & Hermawan, B. (2023). Penerapan SQLMap untuk pengujian keamanan web terhadap serangan SQL injection. *Jurnal Keamanan Siber dan Kriptografi, 5*(2), 56–64. https://doi.org/10.31294/jksk.v5i2.2023

Nugroho, T. H., & Arifin, M. (2021). Implementasi sistem role-based access control pada aplikasi web menggunakan middleware Laravel. *Jurnal Teknologi Informasi dan Ilmu Komputer, 8*(1), 102–110. https://doi.org/10.25126/jtiik.v8i1.2021

OWASP Foundation. (2023). *SQL injection*. Retrieved from https://owasp.org/www-community/attacks/SQL_Injection

OWASP Foundation. (2021). *OWASP Top 10: The ten most critical web application security risks*. Retrieved from https://owasp.org/www-project-top-ten/

Putra, R. A., & Fadillah, A. (2022). Analisis penggunaan framework Laravel dalam pengembangan aplikasi web aman. *Jurnal Ilmiah Teknologi dan Informatika, 14*(3), 121–128. https://doi.org/10.31294/jti.v14i3.2022

Spatie. (2024). *Laravel activitylog*. Retrieved from https://spatie.be/docs/laravel-activitylog

Stuttard, D., & Pinto, M. (2011). *The web application hacker's handbook: Finding and exploiting security flaws* (2nd ed.). Indianapolis, IN: Wiley.