International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

SQL Injection Prevention

Christo Joseph[1], Sam Sebastian[2], Milan Mathai Jiju[3]

[1,2,3]Student, Department of Computer Science and Engineering (Cyber Security), Parul University, Vadodara, India

Abstract

SQL Injection (SQLi) remains one of the most critical and persistent security vulnerabilities affecting web applications worldwide. This research paper presents a comprehensive analysis of SQL Injection attacks, their underlying mechanisms, various attack categories, and the most effective prevention strategies available to developers and security professionals. Through the design and testing of both vulnerable and secured application modules built using PHP and MySQL, this study empirically demonstrates the impact of SQLi attacks and validates the effectiveness of mitigation techniques including parameterized queries, prepared statements, input validation, and least-privilege database access. Results confirm that properly implemented prevention measures can completely neutralize common SQLi attack vectors.

Keywords: SQL Injection, Web Application Security, Prepared Statements, Input Validation, OWASP, Parameterized Queries, Database Security, Cybersecurity.

1 Introduction

Web applications these days deal with a lot of user information by talking to databases in a way. This means they are always sending and receiving data. The problem is that this constant back and forth makes them open to security problems. One of the security risks is something called SQL Injection or SQLi for short. SQL Injection happens when someone, with intentions figures out how to change the information that users put into a website. They do this so they can run SQL commands on the database. When this happens it can lead to people getting to information they should not have information getting changed people getting privileges than they should or even the whole database being taken over. Web applications and SQL Injection are a combination because SQL Injection can cause so many problems.

SQL Injection is a problem that people who make websites have to deal with. It has been on the Open Web Application Security Project list of the security risks for websites for a very long time. [3] Even though we have tools and people are more aware of the problem SQL Injection still hurts a lot of websites every year. The Open Web Application Security Project said in 2021 that SQL Injection is one of the security risks, for websites. This is why SQL Injection is still something that we need to worry about. SQL Injection is a part of a problem called Injection and the Open Web Application Security Project says it is the third most critical risk. [3] This means that SQL Injection is still an important issue.

This paper is about understanding SQL Injection problems well looking at how bad they can be and finding ways to stop them from happening. The study helps us learn about SQL Injection attacks in two ways: by reading about them and, by doing things to see how they work. We also learn how to keep web applications safe by writing code that's secure. Using queries checking what people type in and setting up databases in a safe way are all important things to do to reduce the risk of SQL Injection attacks. SQL Injection attacks

can be very bad. It is good to know how to protect against them by using these methods and understanding SQL Injection really well.

1.1 Motivation

The reason we are doing this research is because of how bad people get into computer systems using SQL Injection. We have seen some problems like the Heartland Payment Systems breach in 2008 the Sony Pictures hack in 2011 and many other websites that were broken into. These SQL Injection problems have caused a lot of trouble. So it is very important, for people who make websites and people who keep websites safe to understand and fix these SQL Injection problems. SQL Injection is a deal and we need to stop it. People who make websites and security researchers need to work on fixing SQL
Injection vulnerabilities.

1.2 Research Objectives

The key objectives of this research are:

- To comprehensively study and categorize SQL Injection attack techniques.
- To implement a real-world application module demonstrating both vulnerable and secure versions.
- To test the effectiveness of various prevention measures using manual and automated attack tools.
- To provide practical, actionable guidance for developers to eliminate SQLi vulnerabilities.

2 Background and Literature Review

2.1 History of SQL Injection

SQL Injection was first talked about in public in 1998 by Jeff Forristal, who is also known as Rain Forest Puppy in the security publication Phrack. [7] This kind of attack worked because people were used to building SQL queries by putting together things that users had typed in into the query strings without checking if it was safe or not. SQL Injection has changed a lot since then it is not something that a few hackers use it is now one of the most common ways that people do bad things on the internet and it is a big problem.

Early web frameworks provided little to no built-in protection against SQLi, and the programming paradigm of the era encouraged dynamic query construction. As databases grew in size and commercial importance, the consequences of SQLi attacks became increasingly severe, spurring the development of preventive techniques including prepared statements, stored procedures, and Object-Relational Mapping (ORM) libraries. [14]

2.2 Fundamentals of SQL Injection

SQL Injection takes advantage of how SQL queriesre put together. When a website is not secure it will take what a user types, in, like a username when they log in and put it into the SQL query. If the website does not check what the user typed a bad person can add some SQL code to the field that changes what the query is supposed to do. This is a problem because SQL Injection can really hurt a website. SQL Injection is something that hackers like to use to get into websites.

For example, consider a vulnerable login query:

```
SELECT * FROM users WHERE username = '$u' AND password = '$p'
```

If an attacker provides the username ' OR '1'='1, the query becomes always true, bypassing authentication entirely. [25]

2.3 Related Work

People have done a lot of research on SQLi attacks and how to stop them. Halfond and others did some work in 2006. They made a list of all the kinds of SQL Injection attacks. [1] They grouped these SQLi

attacks by what method the attackers used and what they were trying to do.
Su and Wassermann also did some work in 2006. They came up with a way to detect SQLi attacks in web applications by looking at the grammar. [2]
Recently Ruse and others in 2012 [5] and Thomas and others in 2018 have been working on using machine learning to detect SQLi attacks. [6] They want to make systems that can find SQLi attack patterns that nobody has seen before. They want these systems to be very accurate, at finding these SQLi attacks.
The OWASP organization has made a lot of information about SQLi. They have the OWASP Testing Guide and the OWASP Cheat Sheet Series. [3] [4] These are really helpful for people who make software and for people who test if software is safe. The OWASP organization and the information they made are very important for this study on SQLi. The OWASP organization is a resource, for people who want to learn about SQLi.

3 Taxonomy of SQL Injection Attacks

SQL Injection attacks can be divided into groups based on the method used. How the results are obtained. It is really important to know about the types of SQL Injection attacks so we can plan good defenses, against SQL Injection attacks. This way we can protect ourselves from SQL Injection attacks.

3.1 In-Band SQL Injection

In-band SQL Injection is the kind of attack that happens a lot. It is pretty simple. This is where the bad guy uses the route to start the attack and to get the results of that attack. The bad guy uses the channel to do the whole thing with the SQL Injection.

3.1.1 Classic (Error-Based) SQL Injection

The attacker does something to make the database give us error messages. These error messages are like hints that give us information about the database. The database error messages might show us what the tables in the database are named and what the columns, in the database are named. We can even see what kind of information the database has and what version of the database they are using. The database is not a secret anymore because of these error messages. The database structure and the things the database contains are out in the open now. The attacker looks at these error messages to learn more about the database. The attacker uses the database error messages to find out more, about the database. [1]

3.1.2 Union-Based SQL Injection

This method uses the SQL UNION operator to add queries to the original one. The SQL UNION operator is used to do this. By making sure the number of columns and the type of data in these columns is the same as the query someone who wants to cause trouble can get information from other tables, in the database. The SQL UNION operator is really important here. [1]

3.2 Inferential (Blind) SQL Injection

When we talk about SQL injection the person doing the SQL injection attack does not get to see what is actually happening inside the SQL database. The person doing the SQL injection attack is basically trying to guess what is going on with the SQL database. In SQL injection the attacker is not able to see the database they can only try to figure it out. SQL injection is a type of attack where the attacker tries to get into the database. They do not get to see what is really going on in the SQL database. The blind SQL injection attacker has to figure things out by looking at how the application's behaving. They use SQL injection to get information, from the database but they do this by watching how the application reacts to blind SQL injection.

3.2.1 Boolean-Based Blind SQLi

The attacker sends questions to the application. Gets different answers depending on whether something is true or false. This helps the attacker figure out things about the database. The attacker asks a lot of yes or no questions. Uses the answers to slowly build a picture of what is in the database. The attacker keeps asking these questions to get information about the database bit, by bit. [13]

3.2.2 Time-Based Blind SQLi

The attacker uses SQL functions like SLEEP to make the database stop for a while. If the database stops the attacker figures out that the condition in the query was correct. This way of doing things even works when you cannot see any difference, in what the application shows. The attacker uses SQL functions like SLEEP to make the database stop. Then the attacker knows what is going on with the database. [13]

3.3 Out-of-Band SQL Injection

Out-of-band SQL Injection happens when the database server can connect to a server controlled by the attacker. The stolen data is sent through a way like DNS or HTTP. This method is not commonly used. It is very risky in places where output is tightly filtered. The database server sends the data to the attackers server. This can happen even if the web applications response is closely watched. Out-of-band SQL Injection is a threat, in situations. It can be used to get data out of the database. The attacker uses a server they control to receive the data. This method relies on the database servers ability to connect to servers. It is not an used method. [14]

3.4 Other Attack Types

There are some types of attack forms that are pretty bad. These include Stored SQL Injection, where someone puts code into a database and it gets used later. Then there is Second-Order SQL Injection, where the bad code is stored in a way but is used in a bad way later on. Mass Assignment attacks are also a problem these attacks use the ORM layers in a way.

These SQL Injection attacks are really bad because they can do a lot of damage to a database. For example Stored SQL Injection can hurt a database on after the bad code has been put in. Second-Order SQL Injection is also bad because it can use the code in a way that is not safe.

Mass Assignment attacks are a problem because they can use the ORM layers to do things. The ORM layers are supposed to help. These attacks use them in a bad way. This is why it is so important to be careful, with SQL Injection and Mass Assignment attacks. Stored SQL Injection and Second-Order SQL Injection and Mass Assignment attacks are all types of attacks that can hurt a database. [1] [15]

Table 1: Taxonomy of SQL Injection attack types, mechanisms, and difficulty.

4 Tools and Technologies

This study used a setup to create and test application modules that were not secure and ones that were secure. The following things were used:

Table 2: Tools and technologies used in this study.

Table 1: Summary of SQL Injection Attack Types

Attack Type	Mechanism	Data Visibility	Difficulty
Classic/ ErrorBased	Force DB error messages	Direct (errors)	Low

Union-Based	UNION SELECT statements	Direct (results)	Medium
Boolean-Based Blind	True/false conditions	Inferred	High
Time-Based Blind	SLEEP() / WAITFOR	Inferred (timing)	High
Out-of-Band	DNS / HTTP exfiltration	External channel	Very High
Stored / SecondOrder	Persisted payload triggered later	Variable	High

Table 2: Tools and Technologies Used

Tool / Technology	Purpose	Version / Note
PHP	Server-side application logic	PHP 8.x (latest stable)
MySQL / MariaDB	Relational database backend	MySQL 8.x / MariaDB 10.x
Apache / XAMPP	Local web server for testing	XAMPP (Windows/Linux/Mac)
sqlmap	Automated SQL injection testing	Latest stable version
phpMyAdmin	Database management and inspection	Bundled with XAMPP
Google Chrome / Firefox	Browser-based testing	Any modern browser

5 Methodology

5.1 Research Design

This study used a way of doing things where we compared two things that are the same. We made two versions of a web application one that was not safe and one that was safe. Then we tried to attack both of them in the way. We looked at what happened to each version to see how well the safety measures we used worked. We did this to see which one was better the web application that was not safe or the web application that was safe.

The way we did this was in four steps:

1. we looked at what other people had found out about this. We read all the research that already existed we checked the OWASP documentation and we looked at the security standards that are, in place.
2. Then we built two versions of the application using PHP and MySQL.
3. After that we tried to hack into both versions to see if we could get in. We did this by manually and automatically trying to inject code into the SQL.
4. Finally we compared what happened. Wrote down which methods actually worked to prevent these types of attacks under different conditions and which methods did not work for SQL injection.

5.2 Application Architecture

The test application simulated a realistic e-commerce environment with three primary functional modules:

- Login Module: Authenticates users against a MySQL database.
- Search Module: Retrieves product data based on user search terms.
- Order Form Module: Accepts and processes customer orders.

The vulnerable version used dynamic SQL query construction through direct string concatenation of user inputs. The secured version implemented parameterized queries (prepared statements) using PHP's mysqli and PDO extensions, combined with server-side input validation and sanitization.

5.3 Attack Scenarios

The following attack payloads were used during testing:

- Authentication Bypass: ' OR '1'='1' # — tested on the login module.
- Union-Based Data Extraction: ' UNION SELECT username, password FROM users # — tested on the search module.
- Boolean-Based Blind: ' AND 1=1 # vs. ' AND 1=2 # — used to confirm SQLi presence.
- Automated Scanning: sqlmap was used to enumerate databases, tables, and columns.

6 Prevention Techniques

Multiple complementary defense layers were implemented in the secured application. Defense-in-depth — the practice of applying multiple overlapping security controls — is essential, as no single technique is sufficient on its own.

6.1 Prepared Statements and Parameterized Queries

To prevent SQL Injection the best approach is to use statements with queries. This is what most people suggest. First you create the SQL query. Leave spaces for the values that users will provide. Then you add these values separately. This way they are not considered part of the SQL syntax no matter what they contain. This method works because user values do not get mixed up with the SQL query. The SQL query and user values stay separate. This prevents SQL Injection. Using statements, with queries is a way to defend against SQL Injection attacks. [3] [4] [21] SQL Injection is stopped when you use queries. This is the way to keep your SQL queries safe from SQL Injection.

PHP mysqli example: [8]

```
$stmt = $conn->prepare("SELECT␣*␣FROM␣users␣WHERE␣username=?␣AND␣ password=?");
$stmt->bind_param("ss", $u, $p);
$stmt->execute();
```

Parameterized queries [3] [4] [21] ensure that user input is always treated as data, never as executable SQL code. This approach completely eliminates classic, union-based, and most other forms of SQL Injection when correctly implemented.

6.2 Input Validation and Sanitization

All user inputs should be validated on the server side to confirm they match expected formats and constraints before being processed. This includes: [20] [19]

- Type checking: Ensuring numeric fields contain only numbers.
- Length limits: Rejecting inputs that exceed expected maximum lengths.
- Allowlisting: Accepting only inputs that match a predefined set of permitted values or patterns.
- Encoding: HTML-encoding output to prevent cross-site scripting (XSS) as a complementary measure.

Note that input validation alone is not sufficient to prevent SQL Injection and must be combined with prepared statements. [20] [19]

6.3 Stored Procedures

Stored procedures are like programs that are already made and stored in the database. If you do them the way like using parameters instead of making up SQL commands inside the procedure they can help keep

things safe like prepared statements do. Stored procedures also let the people, in charge of the database decide what the application is allowed to do. [16] [17]

6.4 Least Privilege Principle

The database account that the web application uses should have the permissions it needs to work. For an online shopping application this means it can only do a few things like look at information add new information update existing information and delete information but only on the specific tables it needs to use. It should not be able to delete tables, change tables, create tables or give permissions to other accounts. This helps to limit the damage if someone's able to get into the database account. The database account should be restricted to the permissions it needs like SELECT, INSERT, UPDATE and DELETE operations and it should not have permissions like DROP, ALTER, CREATE and GRANT. This way if someone gets into the database account they will not be able to do much damage, to the web application. The database account should have the permissions necessary for the web application to work. [10]

6.5 Error Handling and Information Disclosure

Production applications must never expose raw database error messages to end users, as these messages can reveal valuable information about database structure, query logic, and server configuration. Instead, applications should display generic, user-friendly error messages while logging detailed error information server-side for debugging purposes.

6.6 Web Application Firewall (WAF)

A Web Application Firewall can detect and block common SQL Injection patterns in HTTP requests before they reach the application. While WAFs provide a useful additional layer of protection, they are not a substitute for secure coding practices and can be bypassed by sophisticated attackers using obfuscation techniques. [11]

Table 3: Module-Wise Vulnerability vs. Mitigation

Module	Vulnerable Outcome	Secured Outcome	Prevention Applied
Login	Auth bypass via ' OR '1'='1	Login only on valid credentials	Prepared statements + hashed passwords
Search	Union-based data leak	No data leak	Parameterized queries + input length checks
Order Form	SQL & business-logic abuse	Only valid orders accepted	Cast numeric fields + range validation
Error Display	Detailed DB errors shown	Generic message; logs server-side	Sensitive info hidden from users
DB Privileges	Web user had high privileges	Limited to CRUD on necessary tables	Least privilege principle applied

Table 3: Comparison of vulnerable vs. secured outcomes across application modules.

7 Results and Analysis

7.1 Testing Environment

We did some testing in a setup to make sure nothing bad happened to real systems. This setup had Apache and some other tools like MySQL 8 and PHP 8. We also used Google Chrome and Firefox browsers. When

we were testing we used some attack tools like manual SQL injection payloads and a tool called sqlmap to automatically look for vulnerabilities, in the system. We used sqlmap to scan for these vulnerabilities. The testing was done in this controlled environment with MySQL 8 and PHP 8 to see how they worked with the attack tools.

7.2 Vulnerability Demonstration Results

The vulnerable application module demonstrated all expected weaknesses:

- Login Form: Authentication was successfully bypassed using the payload ' OR '1'='1' #, granting unauthorized access without valid credentials.
- Search Module: A UNION-based injection payload successfully extracted all usernames and plaintext passwords from the users table.
- Order Form: Numeric fields (quantity, item ID) were exploitable for union-based and error-based injections, exposing database schema information.
- sqlmap successfully enumerated all databases, tables, and columns automatically on the vulnerable application within minutes. [9]

7.3 Prevention Effectiveness

The secured application version demonstrated complete resistance to all tested attack vectors:

- All manual SQL injection payloads returned generic error messages or were rejected at the input validation stage.
- Prepared statements ensured user input was treated strictly as data. Injection payloads were passed to the database as literal strings, rendering them harmless.
- sqlmap reported no injectable parameters in the secured application across 50+ automated test runs. [9]
- Passwords were stored as bcrypt hashes, ensuring that even in the event of a database compromise, credentials could not be easily recovered.

7.4 Performance Analysis

A concern sometimes raised regarding prepared statements is potential performance overhead compared to dynamic queries. Performance benchmarking across 50 test runs for each module showed that secure queries performed comparably to vulnerable ones, with average execution time differences of less than 5ms. For complex report queries, the overhead was slightly higher but remained within acceptable thresholds for web applications. [6] This confirms that security and performance are not mutually exclusive.

Table 4: Performance Comparison — Vulnerable vs. Secure Queries

Operation	Vulnerable (ms)	Avg	Secure Avg (ms)	Difference
Login (simple)	12 ms		14 ms	+2 ms
Search (LIKE)	25 ms		27 ms	+2 ms
Order Fetch	48 ms		50 ms	+2 ms
Insert Order	33 ms		35 ms	+2 ms
Complex Report	120 ms		126 ms	+6 ms

Table 4: Average query execution time comparison between vulnerable and secure implementations.

8 Discussion

The results of this study conclusively demonstrate that SQL Injection vulnerabilities, while technically straightforward to understand, can have catastrophic consequences when left unaddressed. The ease with which automated tools like sqlmap can exploit vulnerable applications highlights the urgency for developers to adopt secure coding practices.

The biggest thing we found out is that parameterized queries are really good at keeping us safe from SQL Injection. They do this when we set them up the way. We also noticed that using them does not slow down our system much. This is what people at OWASP, NIST and other cybersecurity experts have been saying we should do. [3] [10] What is important to know is that prepared statements do not just clean up the stuff from what people enter. They actually keep the SQL code and the data separate. [4] [22] This gets rid of the reason we have SQL Injection problems in the first place. Parameterized queries are the key, to stopping SQL Injection.

When we talk about security, input validation and the principle of privilege are like two extra layers of protection. If something goes wrong with the statement the input validation will help to reduce the chances of an attack and the principle of least privilege will make sure the damage is not too bad if someone does manage to get in. This way of thinking about security, which is called defense-in-depth is really important, for making sure our security is strong. Input validation and the principle of least privilege work to make our security architecture more robust.

The study also shows that giving out information through error messages is a problem. [23] [24] When something goes wrong with an application the error messages tell attackers a lot, about how the database is set up. This information helps them plan attacks later on. It is an idea for all applications to use simple error messages instead. This is an effective way to make applications more secure.

Future directions for this research include investigation of NoSQL injection vulnerabilities (which follow similar principles but affect document-oriented and key-value databases), ORM-based injection risks, and the application of machine learning models for real-time SQL injection detection in web application firewalls. [5] [6]

9 Conclusion

This research paper has presented a comprehensive study of SQL Injection attacks and their prevention. Through literature review, practical implementation, and systematic testing, we have demonstrated that SQL Injection remains a critical and exploitable vulnerability that continues to affect web applications across all domains and industries.

The implementation of a dual-version application — vulnerable versus secured — provided empirical evidence of both the ease of exploitation and the effectiveness of prevention techniques. Parameterized queries and prepared statements were confirmed as the most effective primary defense, complemented by input validation, error handling, least-privilege database accounts, and Web Application Firewalls.

Crucially, these security measures impose minimal performance overhead, removing any practical justification for their omission. The responsibility for securing web applications against SQL Injection lies with developers, architects, and organizations, and the tools and techniques to do so are well-established and readily available.

By adopting the practices outlined in this paper, developers can significantly reduce the risk of SQL Injection, protecting user data, organizational assets, and the integrity of their applications.

References

1. Halfond, W. G. J., Viegas, J., & Orso, A. (2006). A classification of SQL injection attacks and countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering.
2. Su, Z., & Wassermann, G. (2006). The essence of command injection attacks in web applications. ACM SIGPLAN Notices, 41(1), 372–382.
3. OWASP Foundation. (2021). OWASP Top 10 — 2021: The Ten Most Critical Web Application Security Risks. Retrieved from https://owasp.org/Top10/
4. OWASP Foundation. (2022). SQL Injection Prevention Cheat Sheet. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_ Prevention_Cheat_Sheet.html
5. Ruse, M. E., Sarwar, T., & Basu, S. (2012). Automated detection and analysis of SQL injection vulnerabilities. Proceedings of the ISRN Software Engineering.
6. Thomas, S., Williams, L., & Xie, T. (2018). On automated prepared statement generation to remove SQL injection vulnerabilities. Information & Software Technology, 51(3), 589–598.
7. Forristal, J. (1998). NT Web Technology Vulnerabilities. Phrack Magazine, 54.
8. PHP Documentation. (2024). MySQLi Prepared Statements. Retrieved from https:
9. //www.php.net/manual/en/mysqli.quickstart.prepared-statements.php [9] Damele, B., & Stampar, M. (2024). sqlmap: Automatic SQL injection and database takeover tool. Retrieved from https://sqlmap.org/
10. NIST. (2020). Guide to Application Security — Special Publication 800-95. National Institute of Standards and Technology.
11. Acunetix. (2023). SQL Injection: What Is It and How to Prevent It. Retrieved from https://www.acunetix.com/websitesecurity/sql-injection/
12. Priya, M., Selvakumar, S., & Prabavathy, B. (2012). A survey on detection and prevention of SQL injection attack. International Journal of Computer Applications, 41(7), 23–28.
13. Kindy, D. A., & Pathan, A. K. (2011). A detailed survey on various aspects of SQL injection in web applications: Vulnerabilities, innovative attacks, and countermeasures. Proceedings of the 9th International Conference on Computer and IT Applications.
14. Clarke, J. (2012). SQL Injection Attacks and Defense (2nd ed.). Syngress / Elsevier. [15] Stuttard, D., & Pinto, M. (2011). The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws (2nd ed.). Wiley.
15. Boyd, S. W., & Keromytis, A. D. (2004). SQLrand: Preventing SQL injection attacks. Proceedings of the 2nd Applied Cryptography and Network Security Conference (ACNS), Lecture Notes in Computer Science, 3089, 292–302.
16. Buehrer, G., Weide, B. W., & Sivilotti, P. A. G. (2005). Using parse tree validation to prevent SQL injection attacks. Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM), 106–113.
17. Mitropoulos, D., Louridas, P., Vihos, M., & Spinellis, D. (2019). Time present and time past: Analyzing the evolution of JavaScript code in the wild. Proceedings of the 41st International Conference on Software Engineering (ICSE), 126–137.
18. Li, X., & Xue, Y. (2012). A survey on server-side approaches to securing web applications. ACM Computing Surveys, 46(4), Article 54.
19. Shar, L. K., & Tan, H. B. K. (2012). Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. Information and Software Technology, 55(10), 1767–1780.

20. Microsoft Corporation. (2023). Parameterized queries in SQL Server. Microsoft Developer Documentation. Retrieved from https://docs.microsoft.com/en-us/ sql/relational-databases/security/sql-injection
21. Sadeghian, A., Zamani, M., & Manaf, A. A. (2013). A taxonomy of SQL injection detection and prevention techniques. Proceedings of the International Conference on Informatics and Creative Multimedia (ICICM), 269–273. [23] Verizon. (2023). 2023 Data Breach Investigations Report (DBIR). Verizon Business.
22. Retrieved from https://www.verizon.com/business/resources/reports/dbir/ [24] Imperva. (2022). Web Application Attack Report. Imperva Threat Research. Retrieved from https://www.imperva.com/resources/resource-library/ reports/
23. Anley, C. (2002). Advanced SQL Injection in SQL Server Applications. Next Generation Security Software Ltd. White Paper.