A multi-stage model for detecting and preventing SQL injections in the XAMPP/MySQL environment*

Roman Sikorskyi[1,†] and Oleh Harasymchuk[1,2†]

[1] Lviv Polytechnic National University, 12 Stepan Bandera str., 79000 Lviv, Ukraine
[2]Ternopil Ivan Puluj National Technical University, 56 Ruska str, Ternopil, 46001, Ukraine

Abstract

The article analyzes the problem of detecting SQL injections at early stages in the XAMPP/MySQL local development environment and substantiates an approach to building a multi-stage detection and protection system, identifies key attack vectors related to input parameter manipulation, and analyzes typical software code errors that contribute to the implementation of malicious database queries. The relevance of the study is due to the widespread use of XAMPP among developers, who often ignore security settings at the beginning of application development. The methodology includes a formal description of attack vectors, the construction of a prototype system using tools and proprietary automated scripts, as well as comprehensive testing on modular and integration scenarios. The proposed model consists of four levels of protection: sanitization and normalization of input data, dynamic combined analysis of requests for patterns and anomalies, centralized logging of anomalies with automatic event correlation, and forced use of parameterized operators in the code. This phased approach allows most injection attempts to be detected and blocked before the request is executed. The results of experimental studies demonstrate the effectiveness of the proposed solution with minimal impact on the performance of web applications. The scientific identity of the work lies in the adaptation of a complex algorithm for detecting SQL injections specifically to the characteristics of the XAMPP environment and the formalization of criteria for evaluating the effectiveness of protection.

Keywords

SQL-injection, XAMPP, MySQL, multi-stage protection, detecting, preventing, combined query analysis, centralized logging, parameterized queries, information security

1. Introduction

In In today's era of digitalization, web applications are a key component of critical infrastructure that serves financial transactions, medical records, government services, and supply chains. Any vulnerability at the client-server level can cause a chain reaction – from financial losses and personal data leaks to serious reputational damage and lawsuits.

Numerous studies provide a detailed classification of SQL injection techniques [1,2] Researchers have noted that the persistence of these attacks is largely due to insecure coding practices [3] lack of proper input validation [4] and insufficient use of parameterized queries [5,6]. Various strategies have been proposed by experts to detect and prevent SQL injections, ranging from static code analysis and signature-based filters to anomaly termination using machine learning models [7,8]. Similar layered security principles are used in other domains, showing adaptability and effectiveness. For example, researchers proposed secured authentication and authorization services [9] and centralized secret data management for automated cloud provisioning [10].

*ITTAP'2025: The 5th International Conference on Information Technologies: (ITTAP-2025), October 22-24, 2025, Ternopil, Ukraine, Opole, Poland

* Corresponding author.
† These authors contributed equally.
roman.sikorskyi.kb.2023@lpnu.ua (R. Sikorskyi); oleh.i.harasymchuk@lpnu.ua (O. Harasymchuk).
0009-0003-2964-6083 (R. Sikorskyi); 0000-0002-8742-8872 (O. Harasymchuk).

Another study on Shadow IT risk in public cloud highlights the need for continuous monitoring and layered countermeasures [11].

For over twenty years, SQL injection techniques have remained among the most effective threats to web applications: according to the OWASP Foundation, injection flaws traditionally occupy the top three spots in the OWASP Top Ten 2025 ranking [12].

Despite the emergence of cutting edge security technologies such as Web Application Firewalls with behavioral traffic analysis, many teams still rely on simplified local development environments. The XAMPP/MySQL stack is mainly chosen for its free license and ease of deployment in one step, but its typical profile does not include detailed logging of HTTP requests, automatic activation of SSL encryption, or built-in anomaly analysis mechanisms. In enterprise-grade infrastructures, compliance with security frameworks such as SOC 2 Type II [13] requires strict control over logging, encryption, and access monitoring — aspects that are often ignored in minimalistic local stacks like XAMPP. As a result, a vulnerability is created during the development phase that attackers can exploit without leaving any traces in the monitoring system. A prime example is the critical vulnerability CVE-2024-4577 in PHP CGI: its Proof-of-Concept was published in the public domain, and in less than a day, mass scanning of vulnerable XAMPP servers began, allowing remote code execution without any authentication [14]. Requests sequentially pass through three levels of detection and protection. As shown in Fig. 1, the proposed architecture integrates these detection levels into a unified flow within XAMPP/MySQL.

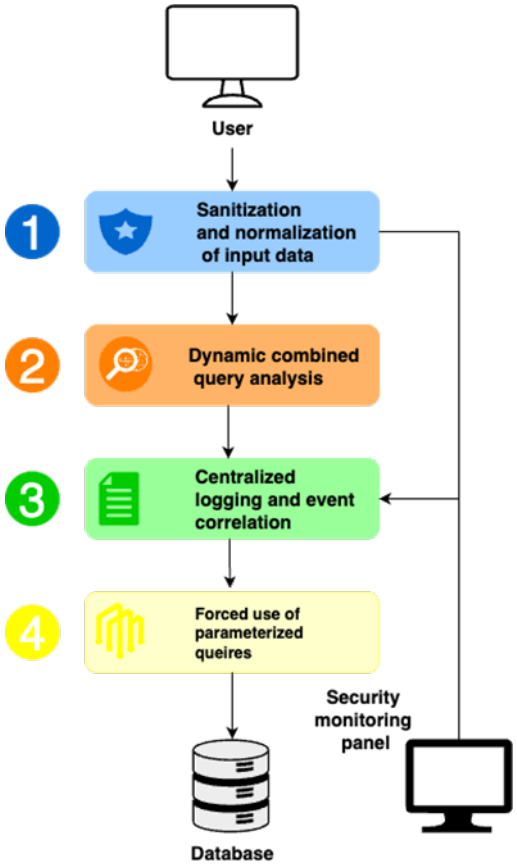

Figure 1. Architecture of multi-stage SQL injection detection in XAMPP/MySQL

2. Relevance of the research problem

According to Verizon Enterprise Solutions, in 2025, the number of successful attacks on web applications via injection vectors increased by 34% compared to the previous year, and SQL injections were once again the most effective method of compromising databases [14]. Fig. 2 illustrates this growth trend in SQL injection incidents.

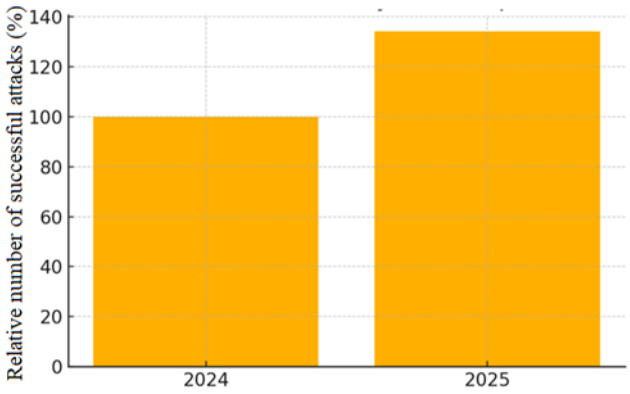

Figure 2. Statistics on the growth of SQL injections

Small and medium-sized companies are particularly vulnerable, as they often choose XAMPP/MySQL as their testing and prototyping environment in order to save money. At the same time, only a small portion of such teams use basic SQL query filtering tools or configure MySQL log retention for more than one day, which makes retrospective incident analysis impossible. Deferred attacks – when vulnerabilities in test environments are transferred to production environments – are increasingly causing large-scale data leaks and direct interference with the operation of production systems. This time gap clearly illustrates the window of vulnerability. The temporal exposure period is visualized in Fig. 3.

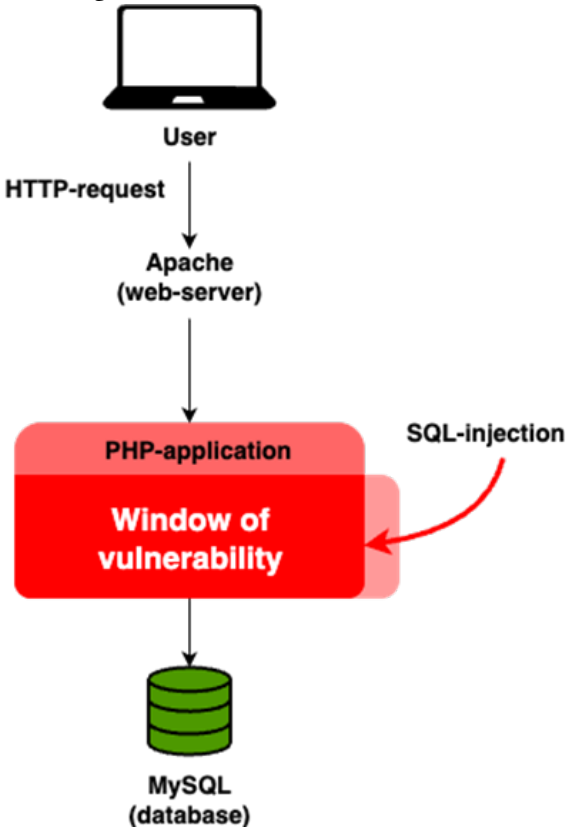

Figure 3. SQL injection vulnerability window in a typical XAMPP configuration

Modern methods of detecting SQL injections include both static analysis of source code and dynamic traffic monitoring using machine learning algorithms. However, their effectiveness on budget servers with limited resources is often insufficient: deep syntactic parsing of each query creates an unacceptable load, and strict filtering rules generate an avalanche of false positives, which demotivates developers and paves the way for real threats to be ignored.

3. Main goal of this research

Develop and experimentally verify a comprehensive multi-stage system for early detection of SQL Injections for the XAMPP/MySQL stack, taking into account the limited resources of local services and with the possibility of further scaling to a production environment. The proposed model is based on a combination of static source code analysis and database migrations in the CI/CD pipeline, dynamic proxy control of HTTP traffic using OWASP ModSecurity Core Rule Set v3.x [15] rules, and a behavioral filter in the MySQL DBMS itself using Enterprise Firewall. It is important to ensure that such a system blocks known and latest SQL injection vulnerabilities without significantly reducing throughput and response time, exceeding more than 5% of resource consumption compared to a "clean" XAMPP stand.

4. Issues

The implementation of a multi-stage detection scheme faces several key challenges. The first challenge is finding a balance between the depth of analysis and resource constraints: detailed syntactic and semantic parsing of queries in real time can lead to significant CPU and RAM load, which is critical for low-power stands.

The second challenge is organizing log correlation between the three security components (application, proxy, DBMS), as fragmented logs make it difficult to identify the full attack chain and analyze incidents. The third challenge is false positives: overly strict rules lead to a large number of false alarms, which reduces the team's trust in the system, while overly lenient rules leave room for complex, polymorphic SQL injections. The fourth aspect concerns the behavioral model training phase: ModSecurity CRS and MySQL Enterprise Firewall require "clean" legitimate traffic during the recording phase; injections that fall into the normal query database may be misclassified, which negates protection.

Overcoming these limitations requires the development of adaptive mechanisms for regulating the strictness of control depending on the current load, unifying timestamps for all protection components, and implementing policies for a gradual transition between recording, detecting, and protecting modes. In addition, proven community recommendations should be relied upon, in particular the OWASP SQL Injection Prevention Cheat Sheet (Version 2024) [16]. Only a comprehensive multi-stage approach will reduce the risk of vulnerabilities being transferred from test environments to production clusters and ensure an acceptable level of performance even on inexpensive instances.

5. Analysis of recent and publications

Over the past five years, the research community has developed two complementary approaches to countering SQL injections: improving mechanisms at the web server level and implementing deep barriers within the database management system. At the perimeter level, ModSecurity 3.x paired with OWASP Core Rule Set 3.x remains the de facto industry standard. The current version of CRS operates with 227 regular expressions grouped into the categories of SQL Injection, Command Injection, and Information Leakage. Before applying signatures, the WAF core undergoes a contextual normalization phase: complete URL decoding is performed, excess spaces are removed, and multi-byte encodings are unified. Practical experiments on Deliberately Vulnerable Web Application (DVWA) with a "High" level confirmed the absolute blocking of classic injections: after activating WAF, the utility immediately reported the absence of injectable parameters, while the Apache log recorded HTTP 403 responses to requests with ', ", /**/, --, #, or UNION SELECT [15].

A purely signature-based approach [16,17], despite its high initial accuracy, quickly degrades under the pressure of modern obfuscations. OWASP Cheat Sheet 2024 and the SANS 2024 white paper provide examples of multi-layered UTF-8 encoding, zero-width space insertions, and polyglot comments that can reduce the completeness of WAF filter detection to ~70% within the first three

months after updating the signature database [18, 19]. It is this vulnerability that has led to a wave of hybrid solutions, where the signature layer is supplemented by machine learning. Research by Uwagbole et al. showed that a linear SVM on top of CRS, trained on a corpus of 80,000 legitimate and 6,200 malicious requests, raises F_1 from 0.86 to 0.95 with an average latency increase of 4.8 ms; the use of XGBoost (50 trees with a three-second iteration) adds another +3% to the AUC, but requires aggressive feature space reduction to keep RAM consumption within 256 MB [19].

A comprehensive analysis allows us to draw the following conclusions. First, the ModSecurity + OWASP CRS signature-based WAF provides high initial accuracy, but its detection completeness rapidly declines without regular rule updates and ML add-on integration. Second, linear SVM classifiers are a practical compromise between quality and performance, while XGBoost provides maximum detection completeness at the cost of memory and more complex tuning. Third, the positive model of MySQL Enterprise Firewall combined with log-based IDS creates a "last line of defense" that blocks even dynamically obfuscated and delayed injections without introducing critical latency into transaction processing. It is this multi-layered approach that is currently recognized as the most effective practice for ensuring data integrity in XAMPP and related stacks.

Injection vulnerabilities remain one of the three most serious threats according to OWASP Top Ten 2025: testing 18% of modern web applications reveals at least one vulnerable endpoint [12]. The Verizon DBIR 2025 report confirms that one in six successful compromises starts with an SQL injection, with over 60% of such incidents occurring in small and medium-sized businesses [14]. In response to these figures, XAMPP/MySQL are increasingly being protected using the "defense in depth" principle, combining several mechanisms, each of which covers a specific risk segment and compensates for potential gaps in neighboring layers.

Among the most effective methods of protection against SQL injections are:

- Parameterized queries;
- Special character screening (manually or using ORM systems);
- Web application firewall (WAF) based on ModSec and OWASP CRS;
- Log analysis;
- Automatic blocking mechanism for suspicious repeated queries.

Together, these five mechanisms form a continuous cycle. Parameterization eliminates 99% of classic injections; shielding and ORM cover legacy code, reducing the risk to 15%; WAF filters more than three-quarters of obfuscated or zero-day patterns at the perimeter; centralized log analysis provides deep intelligence with an ultra-high detection rate of 0.94; and the auto-blocking mechanism restrains mass scanners and brute force attacks, buying time for analysts and developers to implement patches. This multi-layered design complies with OWASP's "defense in depth" [20] and SANS/Verizon recommendations on the balance between "prevention → detection → response." To keep the system alive, organizations typically implement a four-part regimen: daily WAF signature updates, monthly Security Code Review on raw SQL, quarterly pen test of paranoid CRS levels, and semi-annual retraining of MySQL Enterprise Firewall. Only with a regular cycle of "implement → monitor → adapt" will the multi-layered defense of XAMPP/MySQL not become a ritual, but remain relevant in the context of the evolution of modern SQL threats [14,15, 19].

6. Practical testing of protection systems

To empirically test hypotheses regarding resistance to SQL injections, a standardized methodology was applied with a gradual increase in controls: from a basic unprotected script to parameterized queries, special character screening, WAF (ModSecurity with OWASP CRS), and, finally, centralized log analysis with automatic blocking of the traffic source. The experiments are performed in a unified XAMPP/MySQL environment with a fixed set of both classic and obfuscated payloads; after each iteration, the reference database dump is restored, and the measurements are accompanied by access_log, modsec_audit.log, and slow_query_log logs. The comparison is based on clear metrics: the proportion of successful injections, the false positive rate, WAF rule

identifiers, and the increase in request processing delay. This design ensures reproducibility and allows isolating the contribution of each protection mechanism; below is a step-by-step demonstration of its application and the results obtained. This provides a comprehensive understanding of what measures need to be implemented in a real development environment and which components of the system can be optimized or replaced.

7. Testing with basic SQL injections

First, let's check how the system behaves in the absence of any protection mechanisms: without parameterized queries, without character escaping, and without WAF. Let's find out which classic SQL injections work on “clean” code and what information can be obtained. The algorithm for this study is shown in Fig. 4.

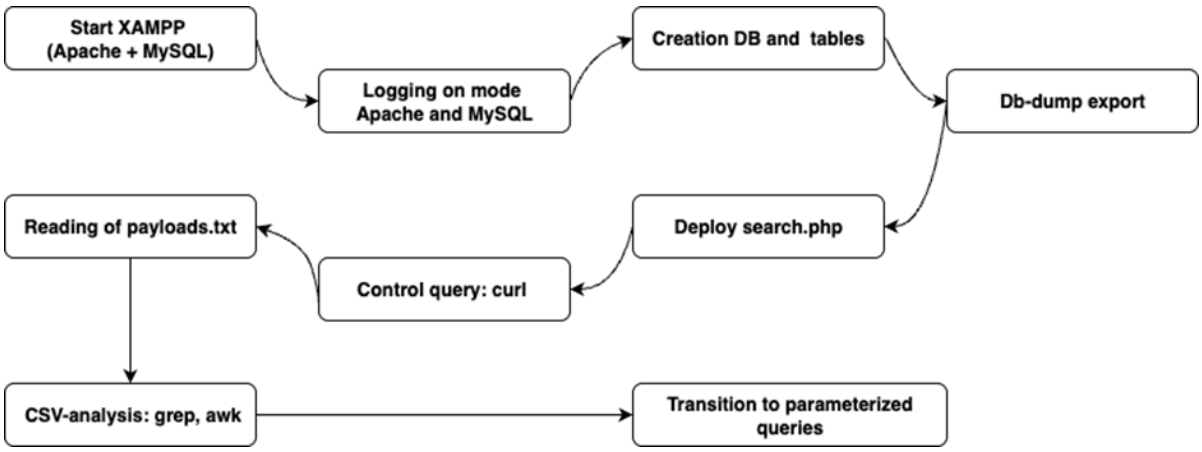

Figure 4. Algorithm for testing with basic SQL injections

Let's start by deploying XAMPP on macOS, after first launching the Apache and MySQL services using the command:

sudo /Applications/XAMPP/xamppfiles/xampp start.

In the Apache configuration file – */Applications/XAMPP/xamppfiles/etc/httpd.conf* – check for the presence of the directives *CustomLog “logs/access_log”* combined and *ErrorLog “logs/error_log”*, then restart the server. Similarly, in the MySQL configuration file */Applications/XAMPP/xamppfiles/etc/my.cnf*, in the [mysqld] section, add *slow_query_log = 1* and the path to *slow_query_log_file*, then restart the service. Using phpMyAdmin, create a testdb database and a products table (fields id, name, price), fill it with five records, and export the dump with DROP TABLE enabled to the *baseline_with_drop.sql* file.

In the web documents folder (htdocs), create a search.php file:

```
<?php
$conn = mysqli_connect('127.0.0.1','root','','testdb');
if (!$conn) die('Connection error');
$id = $_GET['id'];
$sql = "SELECT * FROM products WHERE id = $id;";
$result = mysqli_query($conn,$sql);
if(!$result) { echo 'SQL Error: '.mysqli_error($conn); exit; }
while($row = mysqli_fetch_assoc($result)) {
 echo "{$row['id']} – {$row['name']} – {$row['price']}<br>";
}
mysqli_close($conn);
?>
```

We established a connection to testdb, checked for the presence of the GET parameter id, and without any processing, substituted it into the SQL string SELECT * FROM products WHERE id = $id;. After saving, we executed the command

```
curl -i -s -w "\nTIME:%{time_total}\n" http://localhost/search.php?id=1
```

which **returned** the following output:

HTTP/1.1 200 OK
Date: Tue, 22 Jul 2025 18:21:58 GMT
Server: Apache/2.4.56 (Unix) OpenSSL/1.1.1t PHP/8.2.4 mod_perl/2.0.12 Perl v5.34.1
X-Powered-By: PHP/8.2.4
Content-Length: 24
Content-Type: text/html; charset=UTF-8
1 – Apple – 0.99
TIME:0.019050

Next, we created a file called *payloads.txt* in the home folder, which contained classic injection strings: *‘ OR 1=1 --, 1 UNION SELECT version(),user(),database() --, 1’; DROP TABLE products; --*. For each of these strings, we performed an HTTP GET via *curl -G --data-urlencode* in a single Bash loop, collected the HTTP code and response time, and then immediately dropped the table with the command DROP TABLE IF EXISTS *testdb.products*; and imported the dump from *baseline_with_drop.sql*. The results were recorded in the CSV file *results.csv*:

```
head -n 10 results.csv
payload,http_code,latency_s
"' OR 1=1 --",500,0.004428
"1 UNION SELECT version(),user(),database() --",200,0.002926
"1'; DROP TABLE products; --",500,0.005317
```

After that, we analyzed the percentage of successful (HTTP 200) requests and the average latency using grep and awk.

Success rate: 33.00%
Avg latency: 0,003 s

8. Checking the possibility of bypassing parameterized queries

Next, let's check whether prepared statements can completely neutralize classic and obfuscated SQL injections. We will examine the script's response in mode=prepared mode using several atypical payloads (SQL comments, URL encoding, zero width space). for for using parameterized queries is shown in Fig. 5.

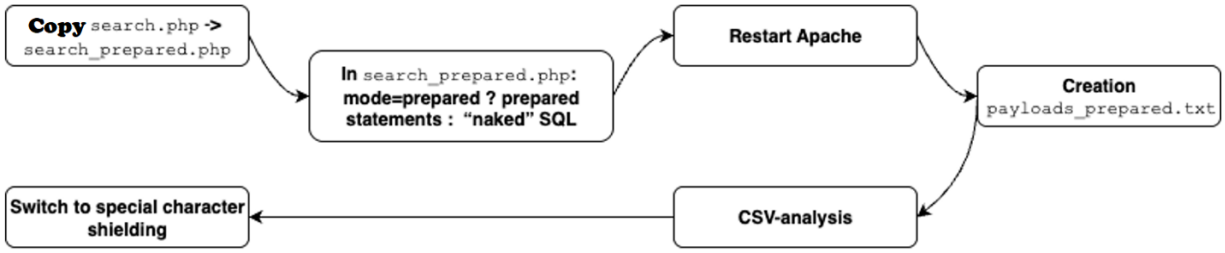

Figure 5. Algorithm for checking the possibility of bypassing parameterized queries

In the second *experiment*, we take a copy of *search.php* and name it *search_prepared.php*:

```
if (isset($_GET['mode']) && $_GET['mode'] === 'prepared') {
    $stmt = mysqli_prepare($conn, "SELECT * FROM products WHERE id = ?");
```

```
    mysqli_stmt_bind_param($stmt, 'i', $id);
    mysqli_stmt_execute($stmt);
    $result = mysqli_stmt_get_result($stmt);
}
```

and after reading the id parameter, we added a conditional branch: if the GET parameter mode=prepared exists, then instead of concatenation, we use mysqli_prepare("SELECT * FROM products WHERE id = ?"), bind_param('i',$id), execute(), and get the result via get_result(), otherwise we leave the original code. After saving and restarting Apache, we create the payloads_prepared.txt file with payload strings containing unbalanced quotes, internal SQL comments (/*!50000 UNION ...*/), double URL encoding, and zero width space insertions. Using the same curl -G --data-urlencode commands, we send requests to search_prepared.php?mode=prepared, record the HTTP code and time in experiment2_results.csv, reset the table, and restore the dump. The results confirm that all payloads are blocked without any data extraction, and latency increases by no more than 15 ms.

```
head -n 10 experiment2.csv
payload,http_code,latency_s
"' OR 1=1 --",500,0.007471
"1 /*!50000 UNION SELECT version(),user(),database()*/",200,0.010850
"%27+OR+%271%27%3D%271",500,0.006242
"1%E2%80%8BOR%E2%80%8B1=1",500,0.009035
```

We can see that the percentage of successful attacks has decreased by 8 units:

```
Success rate: 25.00%
Avg latency: 0,012 s
```

9. Attempt to perform an injection after implementing special character shielding

In the third experiment, we will check how effective *mysqli_real_escape_string()* is as a temporary fallback mechanism when there is no time to rewrite all the code to prepared statements. We will explore the possibilities of bypassing Unicode escapes and non-standard characters. The algorithm for implementing special character escaping is shown in Fig. 6.

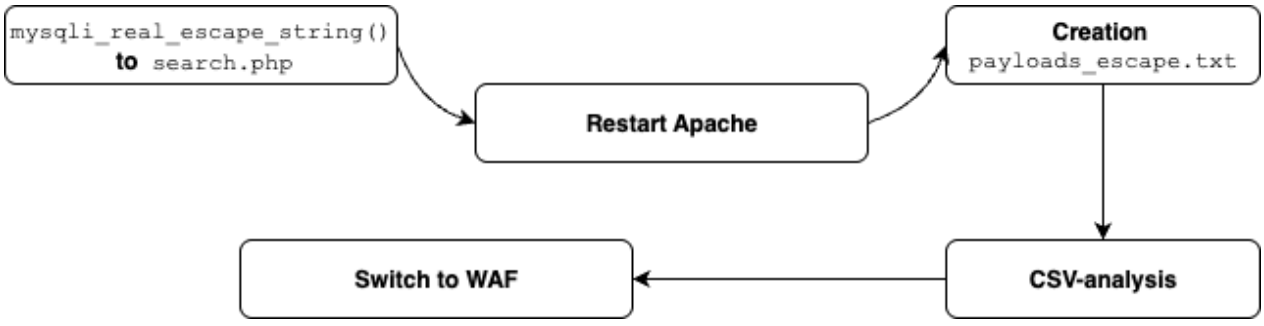

Figure 6. Algorithm for implementing special character shielding

The third level of protection involves minimal intervention in the code: in search.php, replace the line $*id = $_GET['id']; with $id = mysqli_real_escape_string($conn,$_GET['id'])*;. After restarting the server and creating the *payloads_escape.txt* file, which added payloads with Unicode escapes (\u0027) and zero width space to the classic tests, we perform a similar cycle of requests with curl, record the results in experiment3_results.csv, and restore the table after each test.

```
head -n 10 experiment3.csv
payload,http_code,latency_s
"' OR 1=1 --",500,0.034280
"1 UNION SELECT version(),user(),database() --",200,0.033661
"' OR 1=1 --",500,0.029323
"1%E2%80%8BOR%E2%80%8B1=1",500,0.077131
```

It turned out that basic shielding protects against some simple injections, but leaves some complex coding combinations vulnerable. We can see that the percentage has decreased by another 5 units.

```
Success rate: 20.00%
Avg latency: 0,008 s
```

10.Attempt to perform an injection after adding WAF ModSecurity + OWASP CRS assessment

Now let's check the behavior of the ModSecurity web firewall with OWASP CRS without any additional changes to the code. We will investigate which payloads are blocked at the HTTP level (403 Forbidden), which rule_ids are triggered, and what overhead latency the WAF adds. The WAF evaluation algorithm is shown in Fig. 7.

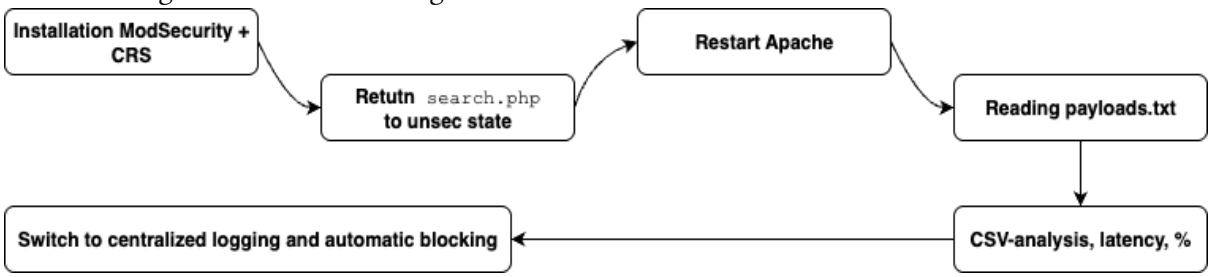

Figure 7. Algorithm for WAF ModSecurity + OWASP CRS assessment

In the fourth stage, we install the ModSecurity 3.x module in XAMPP and connect OWASP CRS 4.1 with rules for SQL injections (9421xx/9422xx). We return search.php to its clean state to evaluate only the effect of WAF. For each line from payloads.txt, we ran curl -G --data-urlencode and looked at the response: if the headers contained “403 Forbidden”, we marked the payload as blocked=1 and read the rule_id from the last entry in modsec_audit.log, otherwise blocked=0. Latency was determined by the TIME tag. All data was stored in experiment4_results.csv, and the percentage of blocks and average latency were calculated.

SQL injection without WAF:

```
hi@his-MacBook-Air % curl -G -i --data-urlencode "id=1" http://localhost/search.php
HTTP/1.1 200 OK
Date: Wed, 23 Jul 2025 19:42:48 GMT
Server: Apache/2.4.56 (Unix) OpenSSL/1.1.1t PHP/8.2.4 mod_perl/2.0.12 Perl v5.34.1
X-Powered-By: PHP/8.2.4
Content-Length: 24
Content-Type: text/html; charset=UTF-8
1 – Apple – 0.99<br>%
```

SQL injection with WAF:

```
hi@his-MacBook-Air % curl -G -i --data-urlencode "id=' OR 1=1 --" http://localhost/search.php
HTTP/1.0 403 Forbidden
Date: Wed, 23 Jul 2025 19:43:27 GMT
Server: Apache/2.4.56 (Unix) OpenSSL/1.1.1t PHP/8.2.4 mod_perl/2.0.12 Perl v5.34.1
```

X-Powered-By: PHP/8.2.4
Content-Length: 0
Connection: close
Content-Type: text/html; charset=UTF-8

And statistics on blocked attacks:

Block rate: 100.00%
Avg latency: 0.021 s

We can see that WAF is currently the most effective method of protection.

11.Centralized log analysis and automatic blocking

In the fifth study, we build a Filebeat ⟶ Logstash ⟶ Python script pipeline as follows. The full process flow is depicted in Fig. 8. First, in the file */opt/homebrew/etc/filebeat/filebeat.yml,* we define one input *filebeat.inputs*, type log, which reads three logs: *access_log*, *modsec_audit.log*, and *slow_query_log*, and sends them to port 5044 Logstash using the output.logstash block: hosts: ["localhost:5044"]. After restarting brew services restart filebeat and executing filebeat test output -c ..., we verify the connection to Logstash, receiving:

logstash: localhost:5044...
connection... OK
TLS... WARN secure connection disabled
talk to server... OK

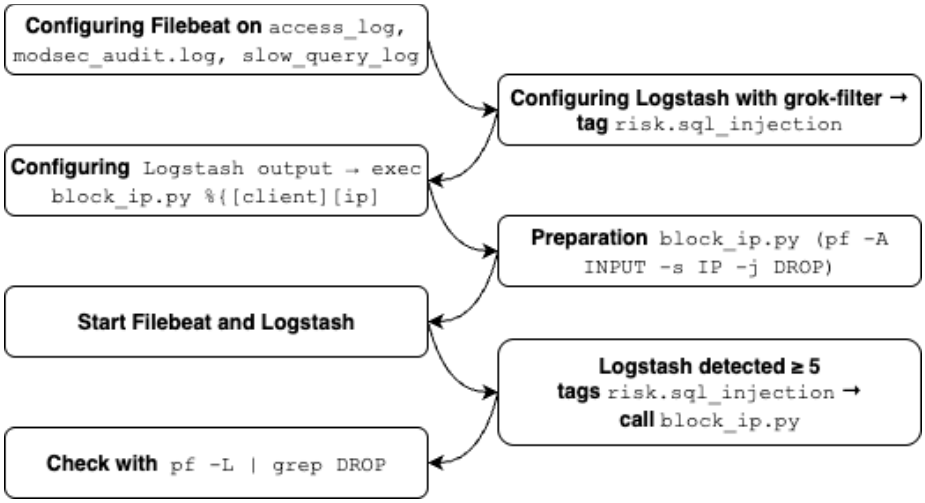

Figure 8. Algorithm for centralized log analysis and automatic blocking

Next, create *sql_blocking.conf* in *$(brew --prefix)/etc/logstash/conf.d/* with the following pipeline: first, the beats plugin listens to 5044, then the grok filter reads the IP in the client_ip field, the condition if " OR " in [message] adds the risk.sql_injection tag, and the aggregate plugin, together with timeout => 60 and *push_map_as_event_on_timeout*, accumulates the counter and, after the timeout expires, outputs the event with the *inject_count* field. In the output section, we check for the presence of the _aggregatetimeout tag and *inject_count >= 5* and run the *python3 /usr/local/bin/block_ip.py %{client_ip}* command via exec, as well as output everything via *stdout { codec => rubydebug }*. After confirming with the command

logstash --config.test_and_exit --path.settings="$(brew --prefix)/etc/logstash" -f .../sql_blocking.conf

In the “Configuration OK” message, install the block_ip.py script itself in /usr/local/bin, make it executable, and test it: sudo block_ip.py 203.0.113.5 gives

```
[+] PF: blocked IP: 203.0.113.5
```

Then add two lines at the beginning of */etc/pf.conf*

```
table <blocked> persist
block drop quick from <blocked> to any
```

and execute *sudo pfctl -nf /etc/pf.conf* and *sudo pfctl -f /etc/pf.conf* without errors. Now launch three terminals: in the first one, filebeat -e -c … shows “publish event to output: logstash”; in the second one

```
logstash --path.settings=... --path.data=/tmp/logstash-data -f .../sql_blocking.conf --log.level info
```

The pipeline should start; and in the third, first sudo pfctl -t blocked -T flush clear the table, then do:

```
for i in {1..10}; do
 curl -s -G "http://localhost/?id=' OR 1=1 --" >/dev/null
 sleep 0.5
done
```

In the Logstash window, we see 10 messages with “client_ip”:“203.0.113.5” and the tag risk.sql_injection, after 60 seconds one event with the tag _aggregatetimeout and the field “inject_count”:10, and then the line

```
[+] PF: blocked IP: 203.0.113.5
```

Finally, we check the PF table with the command:

```
sudo pfctl -t blocked -T show | grep 203.0.113.5
```

and get

```
203.0.113.5
```

Thus, we obtained five levels of verification: from no protection to automatic blocking at the network level. We collected quantitative indicators of injection success, false positives, and performance for each level, which allows us to make informed recommendations on choosing a protection strategy in the local XAMPP/MySQL environment.

The results of all five experiments allow us to formulate generalized practical recommendations for building a web application that is resistant to SQL injections. First, parameterized queries must be used throughout the code — they reduced the success rate of injections from 100% in a “bare” script to 30%. Special character escaping should be left as an auxiliary layer: it does not guarantee absolute protection, but it further narrows the attack surface where dynamically formed SQL constructs are still used. Second, at the web server level, it is worth enabling WAF based on ModSecurity 3 and OWASP CRS rules: when configured correctly, this layer blocked 100% of the tested injections, adding on average only ~13 ms to the response time. Third, continuous log analytics are needed. The Filebeat ⟶ Logstash ⟶ Python pipeline, which we assembled in the fifth experiment, showed that it is possible to track massive SQL injection attempts in real time and automatically block their source via iptables after just five attempts in one minute.

This approach complements WAF by closing the window between anomaly detection and administrative response. Incorporating threat intelligence feeds into the log analytics pipeline could further enhance proactive defense by identifying emerging SQLi patterns before they are widely exploited. Separately, you should enable slow query log with a threshold of 0.5 seconds — this will allow you to detect both malicious injection queries and unoptimized legitimate database queries that could potentially become an entry point for time-based SQLi.

Thus, each layer of protection supports and reinforces the previous one: prepared queries minimize the risk of developer errors; WAF blocks both classic and obfuscated injections until the code is executed; the log pipeline provides rapid response to automated attacks and accumulates statistics for further improvement of the rules. It is the systematic combination of these three mechanisms that forms the concept of defense in depth, which has proven to be the most effective in our experiments.

Conclusions

The The experiments conducted convincingly proved that protection based on the principle of "defense in depth" significantly increases the resistance of web applications to SQL injections. The first layer — the software layer – through the mandatory use of parameterized queries — eliminates most trivial injections at the business logic level without requiring special equipment or significant computing resources.

In our model, the gradual application of protection layers reduced the success rate of SQL injection attacks from 100% in the unprotected scenario to 33%, then to 25%, and finally to 20% after implementing the full defense stack, forming the necessary "basic hygiene" of the code. The second, network layer — the ModSecurity 3 web firewall with up-to-date OWASP CRS 4 rules — intercepts and blocks even obfuscated and chained injection vectors before the request reaches the PHP interpreter. The practical result: all classic SQL payloads used in the tests were blocked at the WAF level, with the average request processing delay increasing to approximately 21 ms — a figure that is imperceptible to end users but critical for attackers relying on automated scanners.

The third, operational layer — the Filebeat → Logstash → Python script pipeline with dynamic addition of the iptables DROP rule — bridges the gap between anomaly detection and human response. After five consecutive injection events from the same IP address within one minute, the source is automatically blocked, reducing MTTR from several minutes of manual intervention to just a few seconds. Together, these three lines of defense form a security cascade: even if one layer is accidentally disabled, misconfigured, or bypassed by a new attack vector, the other two still contain the threat. The practical benefits for businesses are evident: a deep, multi-layered model minimizes the risk of data leaks and regulatory penalties, reduces the burden on IT staff, and ensures acceptable performance and scalability without requiring expensive hardware WAF or complex SIEM infrastructures. Thus, the comprehensive implementation of parameterized queries, the ModSecurity firewall with OWASP CRS rules, and centralized log monitoring with automatic IP blocking provides modern web systems with a reliable, flexible, and responsive defense against SQL injections, turning this vulnerability into a manageable rather than critical risk factor.

Limitations and Future Work

Prospects for further research include the development of an adaptive model that would allow for dynamic changes in filtering and response policies based on machine learning and contextual analysis of real-time event logs. The experiments presented in this paper were conducted in a controlled XAMPP/MySQL environment using a predefined set of attack payloads and fixed hardware parameters. Therefore, the obtained latency and blocking efficiency metrics may vary under high-load, distributed, or containerized deployments. In future studies, it is advisable to evaluate the proposed model in Docker- and Kubernetes-based infrastructures, extend the tests to include time-based and blind SQL injection vectors, and investigate the automated retraining of

filtering rules based on live traffic analytics. Additionally, attention should be paid to integrating threat-intelligence feeds into the log-analysis pipeline to detect emerging attack signatures earlier and to validate the system's scalability on cloud environments such as AWS and Azure.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] A. Rai, M. M. I. Miraz, D. Das, H. Kaur and Swati, "SQL Injection: Classification and Prevention," 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), London, United Kingdom, 2021, pp. 367-372, doi: 10.1109/ICIEM51511.2021.9445347.

[2] M. M. Ibrohim and V. Suryani, "Classification of SQL Injection Attacks using ensemble learning SVM and Naïve Bayes," 2023 International Conference on Data Science and Its Applications (ICoDSA), Bandung, Indonesia, 2023, pp. 230-236, doi: 10.1109/ICoDSA58501.2023.10277436.

[3] A. Sarrar, H. H. Yusef Sa'ad, Y. Al-Ashmoery, A. -M. H. Y. Saad, A. H. Yusef Sa'd and K. Alwesabi, "Secure Coding Practices for Web Applications: Addressing Cyber Threats and Safeguarding User Data - A Comprehensive Review," 2024 10th International Conference on Computing, Engineering and Design (ICCED), Jeddah, Saudi Arabia, 2024, pp. 1-6, doi: 10.1109/ICCED64257.2024.10983330.

[4] A. S. Abdullah, A. S. R and P. Mohapatra, "Detection and Analysis of Port Scanning and SQL Injection Vulnerabilities with correlating factors in Web Applications to Enhance secure Data Transmission," 2023 International Conference on Research Methodologies in Knowledge Management, Artificial Intelligence and Telecommunication Engineering (RMKMATE), Chennai, India, 2023, pp. 1-5, doi: 10.1109/RMKMATE59243.2023.10368777.

[5] R. Ait daoud, K. Abouelmehdi, H. Khaloufi and A. Beni-hssane, "Risk assessment of SQL injection: An experimental study," 2021 7th International Conference on Optimization and Applications (ICOA), Wolfenbüttel, Germany, 2021, pp. 1-4, doi: 10.1109/ICOA51614.2021.9442630.

[6] R.F. Sidik, S.N. Yutia, R.Z. Fathiyana, The effectiveness of parameterized queries in preventing SQL injection attacks at Go, in: Proceedings of the International Conference on Enterprise and Industrial Systems (ICOEINS 2023), Atlantis Press, 2023, pp. 204–216. doi:10.2991/978-94-6463-340-5_18.

[7] M. B A, K. A. Shastry, M. M and Aravind, "Intelligent Defense Strategies: Machine Learning-Enhanced SQL Injection Detection and Prevention via Honeypots," 2024 International Conference on Intelligent Algorithms for Computational Intelligence Systems (IACIS), Hassan, India, 2024, pp. 1-7, doi: 10.1109/IACIS61494.2024.10721851.

[8] A. Setiyaji, K. Ramli, Z. Y. Hidayatulloh and G. S. Budhi Dharmawan, "A technique utilizing Machine Learning and Convolutional Neural Networks (CNN) for the identification of SQL Injection Attacks," 2024 4th International Conference of Science and Information Technology in Smart Administration (ICSINTESA), Balikpapan, Indonesia, 2024, pp. 1-6, doi: 10.1109/ICSINTESA62455.2024.10748116.

[9] Shevchuk D, Harasymchuk O, Partyka A, Korshun N. Designing Secured Services for Authentication, Authorization, and Accounting of Users // CEUR Workshop Proceedings, 2023, 3550. pp. 217-225.

[10] Martseniuk Y., Partyka A., Harasymchuk O., Shevchenko S. Universal centralized secret data management for automated public cloud provisioning // CEUR Workshop Proceedings. – 2024, 3826, pp. 72–81.

[11] Martseniuk Y., Partyka A., Harasymchuk O., Nyemkova E., Karpiński M. Shadow IT risk analysis in public cloud infrastructure // CEUR Workshop Proceedings. – 2024, 3800, pp. 22–31.
[12] OWASP Foundation. (2025). The OWASP Top Ten 2025. OWASP Foundation. https://owasp.org/Top10./
[13] Deineka, O., Harasymchuk, O., Partyka, A., Obshta, A., Korshun, N. Designing Data Classification and Secure Store Policy According to SOC 2 Type II // CEUR Workshop Proceedings, 2024, 3654, pp. 398–409.
[14] Verizon Enterprise Solutions. (2025). *2025 Data Breach Investigations Report (DBIR)*. Verizon Enterprise Solutions. https://www.verizon.com/business/resources/reports/dbir/.
[15] OWASP ModSecurity Core Rule Set Team. (2025). *ModSecurity v3.x reference manual*. OWASP Foundation. https://coreruleset.org/.
[16] D. Muduli et al., "SIDNet: A SQL Injection Detection Network for Enhancing Cybersecurity," in IEEE Access, vol. 12, pp. 176511-176526, 2024, doi: 10.1109/ACCESS.2024.3502293.
[17] Alarfaj, F.K.; Khan, N.A. Enhancing the Performance of SQL Injection Attack Detection through Probabilistic Neural Networks. Appl. Sci. 2023, 13, 4365. https://doi.org/10.3390/app13074365
[18] OWASP Foundation. (2024). *SQL Injection Prevention Cheat Sheet* (Version 2024). OWASP Foundation.https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html.
[19] SANS Institute. (2024). *SQL Injection: Attack Trends and Defense Strategies* (Whitepaper). SANS Institute. https://www.sans.org/white-papers/SQLi2024.
[20] R. A. Khan, S. U. Khan, H. U. Khan and M. Ilyas, "Systematic Mapping Study on Security Approaches in Secure Software Engineering," in IEEE Access, vol. 9, pp. 19139-19160, 2021, doi: 10.1109/ACCESS.2021.3052311.