A Case Study On SQL Injection Defense Methods With Demonstration

Nikoloz Kurtanidze &James McGrath

Abstract

SQL injection remains a critical security risk for web applications, through enabling unauthorized data access, authentication bypass, and system compromise (Kumar et al., 2024). In this survey, we classify defenses into five complementary layers, static and runtime monitoring, prepared statements, machine learning detection, alert prioritization, and CI/CD regression testing. We begin with hybrid static analysis and runtime monitoring techniques exemplified by AMNESIA (Halfond et al., 2006) and parse tree validation, then examine parameterized query approaches such as SQLrand (Boyd & Keromytis, 2004). Next we go into evaluating deep learning frameworks like SQLR34P3R (Paul et al., 2024) and cloud native ensemble classifiers (Kumar et al., 2024) for their ability to detect evolving payloads and stay up to date on new potential exploits. We also discuss prioritization models that rank alerts by their risk to focus on which alert to respond to first, and CI/CD integrated test suites can catch regressions before deployment so vulnerabilities are never exposed in the final deployment. To validate these defenses we present a hands-on PHP/MySQL demo that illustrates tautology and UNION based injections on a vulnerable web application and methods to patch the vulnerabilities. Finally we analyze the May 2023 MOVEit Transfer breach which was a zero day SQL injection attack that exposed data from over 1000 organizations world wide and resulted in the loss of over 80 million dollars (Dosumu, 2025). We conclude by identifying challenges for researchers as creating more efficient machine learning algorithms, and to automate code migration tools to better catch SQL injection methods.

1 Introduction

SQL injection to this day still remains one of the most prevalent threats to web applications, and can cause severe operational and financial impacts (Kumar et al., 2024). SQL injection at its core occurs when an input is able to be executed as a SQL query, which enables attackers to bypass authentication, exfiltrate sensitive data, or execute arbitrary commands (Kumar et al., 2024). Early foundational articles include Halfond, Viegas, and Orso's comprehensive classification of attacks and countermeasures (Halfond et al., 2006), Boy and Keromytis's SQLrand, which randomizes SQL keywords to break injected payloads (Boyd & Keromytis, 2004), and Tan and Shar's survey of defense coding, static analysis, and runtime filtering techniques (Tan & Shar, 2012).

Despite these foundational articles outlining defense methods, the shift towards cloud based and microservice architecture has introduced new complex vectors such as second order injections, out of bank channels, and API driven payloads which all evade legacy tools (Kumar et al., 2024). To address these gaps, recent 2024 research proposes new machine learning frameworks. Paul, Sharma, and Olukoya's SQLR34P3R integrates CNN-LSTM models for multi class detection and alert prioritization across HTTP and non HTTP traffic (Paul et al., 2024) and Kumar, Dutta, and Pranav analyze advanced injection patterns in cloud and web applications, and demonstrates how modern payloads bypass traditional defenses (Kumar et al., 2024).

To ground our survey in real world stakes we will look at Dosumu's case study of the May 2023 MOVEit breach where unauthenticated SQL injection (CVE-2023-34362) exposed

data from over 1000 organizations world wide and caused over 80 million dollars in damages (Dosumu, 2025). In this paper we organize defenses into five categories: static and runtime monitoring, prepared statements, machine learning based detection, alert prioritization, and CI/CD regression testing. We also will provide a demo on how SQL injection can be used to bypass authentication and retrieve sensitive data.

2 Background & Categories

SQL injection attacks exploit flaws in the application code that concatenate untrusted input into SQL statements, which in turn allow adversaries to inject malicious input to alter query logic or exfiltrate data. Halfond, Viegas, and Orso's research categorizes these attacks into size primary variants which are tautology (forcing conditions always true), UNION based data exfiltration, blind injections via side channels, error based injections, second order injections stored for later execution, and out of band payloads using external channels like DNS. It is essential to understand these potential threats before evaluating any countermeasures.

To address this diversity, the literature groups defenses into five complementary layers. Static and runtime monitoring tools perform taint analysis or parse tree validation at runtime or offline to block malformed statements before they even hit the database (Halford et al., 2006). Prepared statements forces developers to parameterize APIs so user input is strictly data and not code, which effectively neutralizes classic tautology and comment based attacks (Boyd & Keromytis, 2006). Machine learning detection methods train CNN-LSTM hybrid models to detect subtle deviations from benign query distributions, improving over regex-based scanners but requiring labeled corpora and computational resources (Paul et al., 2024). Alert prioritization frameworks then ranks identified anomalies by risk level, and helps security teams focus on the

most critical threats first. Finally CI/CD regression test pipelines integrate SQL injection test suites into automated build and deployment workflows, which reorders and prioritizes test cases dynamically to catch vulnerabilities and address them before production (Kumar et al., 2024).

3 Survey of Foundational SQL Injection Defense Techniques

3.1 Static & Runtime Monitoring

Combining static analysis of application source code with runtime monitoring of SQL statements is one of the earliest hybrid approaches to stop SQL injections before they reach the database. Halfonr, Viegas, and Orso describe a system called AMNESIA that works by first extracting a model of each query's structure using static analysis and then at runtime it checks each issued query against the model to block any deviation from the expected grammar (Halfond et al., 2006). Parse tree validation analyzes the abstract syntax tree of each query to ensure that only well formed and valid statements make it to the database and get executed (Boyd & Keromytis, 2004). Language integrated taint analysis labels user inputs as tainted and prevents them from influencing control flow or SQL API calls (Tan & Shar, 2012). Through combining static enforcement with runtime monitoring, systems can detect injection patterns that evade purely static or purely dynamic tools (Halfond & Orso, 2006). While this method is effective against many injection patterns, these approaches often require significant instrumentation of application code and can introduce runtime overhead (Boyd & Keromytis, 2004). This method may also struggle with more complex features such as stored procedures or dynamically generated SQL constructed in multiple layers of application logic (Halfond et al., 2006).

3.2 Prepared Statement Enforcement

Prepared statements enforce a strict separation between SQL code and data the user supplies through using placeholders for all external inputs. When properly applied this method prevents attackers from injecting SQL fragments into query syntax as the database strictly treats all parameters as data (Boyd & Keromytis, 2004). Most modern database APIs usually offer native support for prepared statements which makes adopting this method straightforward in newer databases. Additionally some developers inadvertently interpolate untrusted inputs into format strings or bypass prepared APIs for perceived performance gains, which reintroduces vulnerabilities (Tan & Shar, 2012). Overall prepared statements are widely considered best practice and neutralize classic tautology and comment based injections, which are two common SQL injection methods, ensuring their consistent use across large codebases remains a significant engineering effort.

3.3 Machine Learning Based Detection

Machine learning techniques have been introduced as a method of detecting SQL injection due to machine learning's ability to recognize subtle injection patterns that may elude rule based scanners. Paul, Sharma, and Olukoya’s SQLR34P3R framework trains CNN-LSTM models to analyze both HTTP and non HTTP payloads, and through these models high accuracy in distinguishing malicious queries was achieved (Paul et al., 2024). Kumar, Dutta, and Pranav extended these detection methods to cloud native and microservice architecture, and noted that encrypted tunnels and diverse payload encodings complicate signature based defenses (Kumar et al., 2024). These machine learning approaches typically require a model to be trained with a labeled dataset of safe and malicious queries for training and periodic retraining. Challenges with this method include the computational cost of training a model and keeping it trained on new

evolving attack vectors, the risk of adversarial evasion through crafted payloads, and maintaining low false positive rates to avoid alert fatigue (Paul et al., 2024). Despite these hurdles, machine learning detection methods have so far proven promising to detect a wide array of different attacks committed on a system.

3.4 Alert Prioritizing

When SQL injection attempts are detected, alert prioritization framework ranks the alerts by severity to streamline incident response. SQLR34P3R framework incorporates a risk scoring model that weighs anomalies based on feature patterns such as token irregularities, execution context, and historical incident impact (Paul et al., 2024). Clustering similar alerts using unsupervised algorithms can further reduce the volume of notifications and also highlight recurring attacks. Complementary rule based prioritization leverages CVSS scores and vulnerability lifecycle data to calibrate alert thresholds dynamically (Kumar et al., 2024). By focusing analyst effort on high confidence, and high impact events these systems mitigate the risk of key threats being overlooked due to their alert not being as high of priority as other alerts. However, the effectiveness of prioritization hinges on the quality of underlying risk models and may require additional tuning in order to align with organizational risks. Poorly calibrated scoring systems can inadvertently push critical alerts into lower tiers, which delays response time to severe vulnerabilities (Paul et al., 2024).

3.5 CI/CD Regression Testing

Integrating SQL injection test cases into CI/CD pipelines automates checking for vulnerabilities on every code commit (Kumar et al., 2024). Test suites integrated into the CI/CD pipelines can include both positive cases to verify the expected functionality, and negative cases to ensure injection payloads are properly neutralized (Kumar et al, 2024). Mutation based testing tools generate known attack patterns to challenge the sanitization login within the code. Differential testing compares the application behavior against a baseline to quickly identify any vulnerabilities. Executing tests in containerized environments that mirror production reduces false negatives caused by configuration differences. While CI/CD integration provides an effective safety net, test suites still need to be constantly maintained and updated to keep up with the development of new exploits. The main challenge is that running excessive test can slow development (Kumar et al., 2024).

4 Demonstration

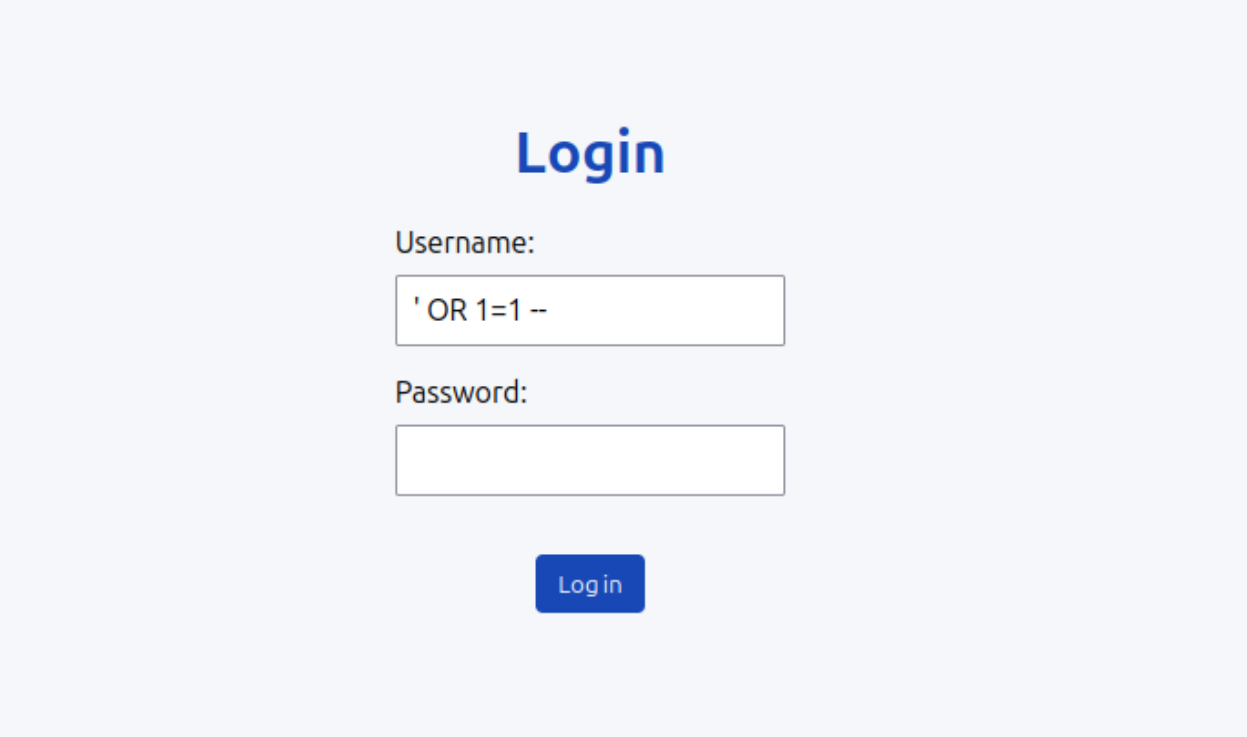
In this section we present a hands on proof concept by using a deliberately vulnerable PHP/MySQL banking web application. This Demo illustrates two of the most classic SQL injection techniques, tautology based login bypass and UNION-based data exfiltration against a simple web interface that manages users, accounts, and transactions. Through injecting payloads into the login page and the transaction search page, we show how non parameterized queries allow attackers to bypass authentication and retrieve all user credentials and account data. After demonstrating both types of attacks we will demonstrate methods to secure the application through using prepared statements and parameter binding. Through making these minimal changes we can protect our application from malicious inputs.

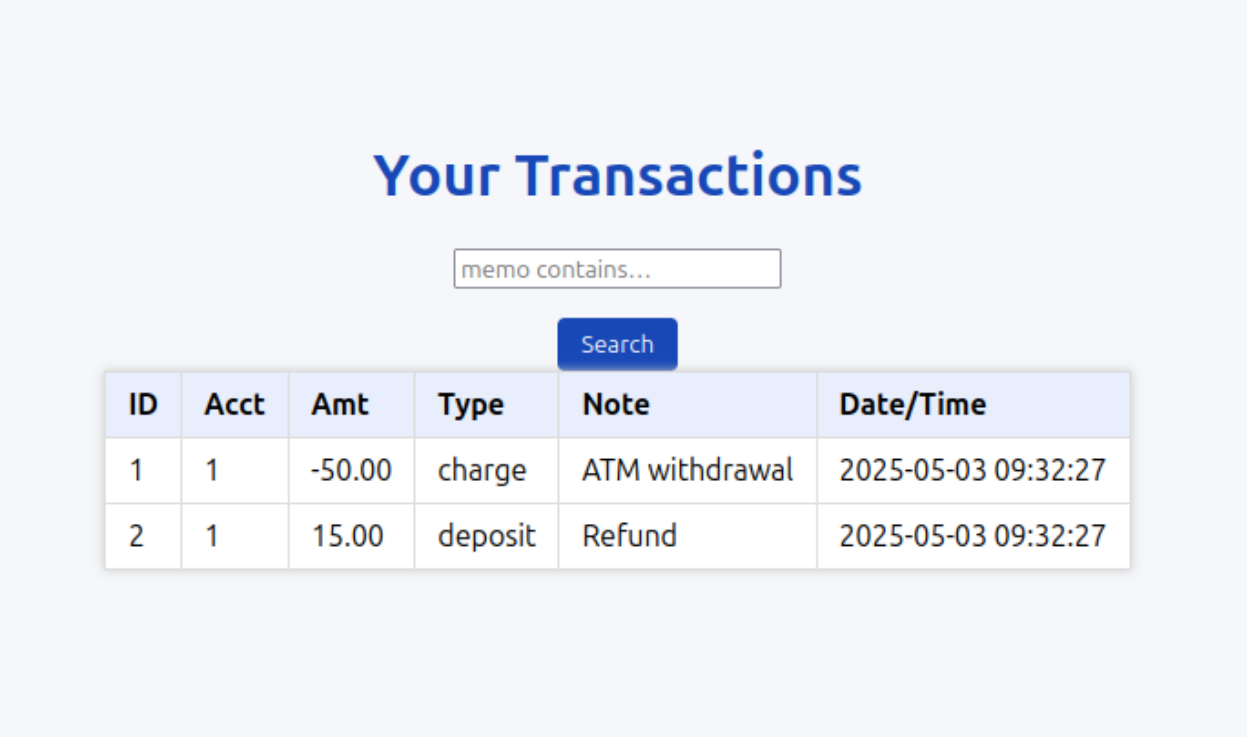
4.1 Vulnerability 1: Login Bypass

The code shown below shows the vulnerable fragment in the login.php file, where the username ($u) and password ($p) variables are interpolated verbatim into the SQL statement. Because no parameterization or escaping is used, any SQL syntax in ($u) or ($p) becomes part of the executed query, which opens the door to injection attacks.

```php
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);
session_start();

$mysqli = new mysqli("localhost","vuln","","bankdb");

$u = $_POST['username'] ?? '';
$p = $_POST['password'] ?? '';

$q = "SELECT * FROM users WHERE username='$u' AND password='$p'";
$res = $mysqli->query($q);

if ($row = $res->fetch_assoc()) {
    $_SESSION['uid']  = $row['id'];
    $_SESSION['role'] = $row['role'];     // 'customer' or 'admin'
    header("Location: ".($row['role']==='admin' ? 'admin.php' : 'search.php'));
    exit;
}
echo "Login failed.";
?>
```

In the next image below you will see how inserting the payload “ **‘ OR 1=1 –** ” into the username input field and either entering any text into the password field or leaving it blank will bypass the authentication without any valid credentials. By using this payload into the username field the “--” at the end of the payload comments out the remainder of the WHERE clause in the above code which effectively transforms the query into “SELECT * FROM users WHERE username='' OR 1=1 – ' AND password='$p';“.





4.2 Data Exfiltration & Tampering

The below code shows the vulnerable fragment in search.php, where both the $uid and the search term $sterm are placed directly into the SQL string. Without parameter binding, an attacker can append arbitrary clauses to the query.

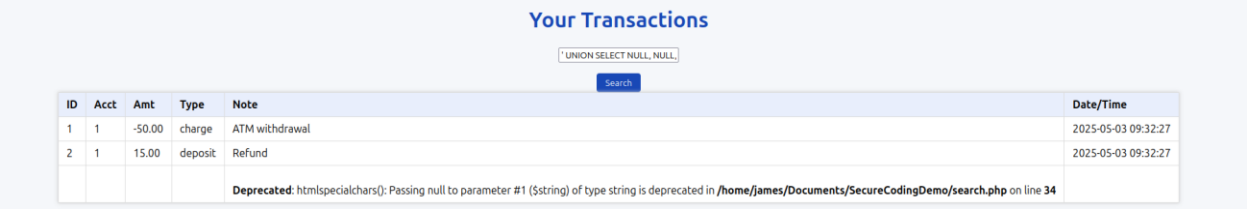
```php
$uid  = $_SESSION['uid'];
$term = $_GET['q'] ?? '';

$mysqli = new mysqli("localhost","vuln","","bankdb");
$sql = "SELECT t.txn_id, a.acct_id, t.amount, t.txn_type, t.note, t.ts
        FROM transactions t
        JOIN accounts a ON a.acct_id = t.acct_id
        WHERE a.user_id = $uid
          AND t.note LIKE '%$term%'";
```

The next images show step by step how to find the table names and retrieve sensitive data such as usernames and passwords. The first image shows how inputting a single ‘ into the memo contains a field to test if the payload is reaching the SQL. If the website returns an error confirming the payload is reaching the SQL. The image below is the error that confirms SQL injection is possible

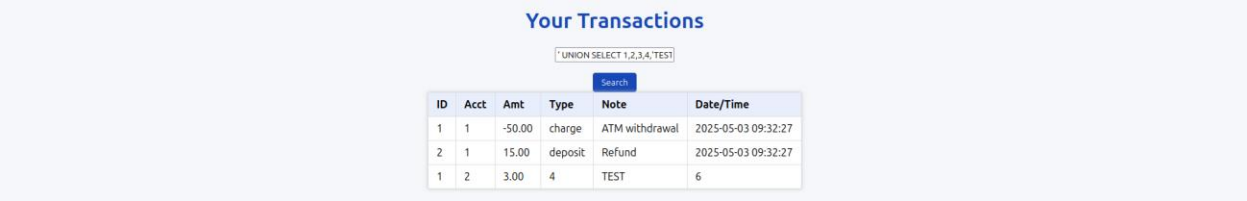
Fatal error: Uncaught mysqli_sql_exception: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '%" at line 5 in /home/james/Documents/SecureCodingDemo/search.php:17 Stack trace: #0 /home/james/Documents/SecureCodingDemo/search.php(17): mysqli->query() #1 {main} thrown in **/home/james/Documents/SecureCodingDemo/search.php** on line **17**

Then the next step is determining the number of columns in both SELECTs by submitting the input “ ‘ UNION SELECT NULL – “. If you get a column that doesn't match, keep increasing the number of “NULLs” in the input until you stop getting the error then you know the number of columns. The below image shows the number of “NULLs” needed to not cause an error is 6 “NULLs” is the statement:

“ ' UNION SELECT NULL, NULL, NULL, NULL, NULL, NULL – ”



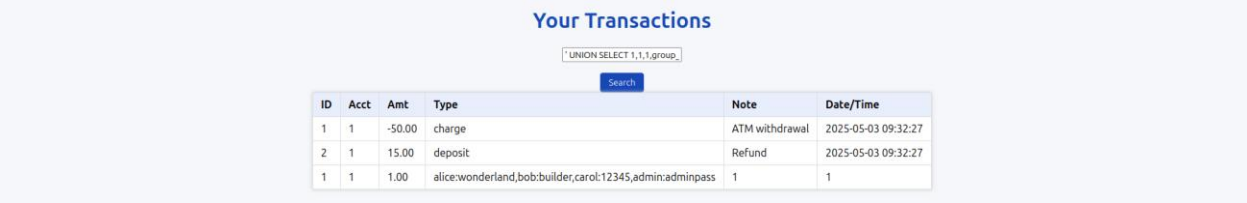
After you confirm the number of columns you need to find one that can display text so you can use the query “ ‘ UNION SELECT 1,2,3,4,5,6 – “ and replace any number with “TEST” to determine which slot in the table can display strings. In our case it is the 5th column which we can use so we will use the query “ ‘ UNION SELECT 1,2,3,4,‘Test’,6 – “. The image below shows the output after entering that query and how it creates a new field with the note TEST showing that we can output strings such as usernames and passwords to that field.



So now that we know where we can output strings we can create the final query to exfiltrate sensitive data from the SQL database. We can use the group_concat(username, 0x3a, password) function to merge the username and passwords in the users table into just one string. So with this function we can rewrite the query to be:

“ ‘ UNION SELECT 1,1,1,group_concat(username,0x3a,password),1,1 FROM users – “

In the image below we can see after inputting this query we see the usernames and passwords of all the users which can be easily stolen.



4.3 Patching the Vulnerabilities

To fully neutralize the demonstrated SQL injection attacks shown above, we refactored both the login.php and search.php to use prepared statements with bound parameters. This ensures that user inputs are sent strictly as data and never interpreted as SQL syntax. Pictured below is the fixed code to fully secure login.php from SQL injection. In the changes below the ? placeholder guarantees that whatever is in $u and $p cannot alter the SQL structure even payloads like “ ‘ OR 1=1 – become literal strings and fault to satisfy the query.

```php
<?php
ini_set('display_errors',1);
error_reporting(E_ALL);
session_start();

$mysqli = new mysqli("localhost","vuln","","bankdb");

$u = $_POST['username'] ?? '';
$p = $_POST['password'] ?? '';

// Prepare the statement with placeholders
$stmt = $mysqli->prepare(
    "SELECT id FROM users WHERE username = ? AND password = ?"
  );
  $stmt->bind_param("ss", $u, $p);
  $stmt->execute();
  $stmt->bind_result($id);

  if ($stmt->fetch()) {
    header("Location: search.php?uid=$id");
    exit;
  }
  echo "Login failed.";
?>
```

Similarly for the search.php code you update it to use parameterization both of the user ID and the search term by binding $term inside CONCAT(‘%’, ?, ‘%’). Through doing this any injected SQL is confined to the pattern match string and cannot inject new clauses or UNIONs.

```php
<?php
ini_set('display_errors', 1);
error_reporting(E_ALL);
session_start();
if (!isset($_SESSION['uid'])) { header("Location: login.html"); exit; }

$mysqli = new mysqli("localhost","vuln","","bankdb");
$uid  = (int)$_GET['uid'];       // cast to int for safety
$term = $_GET['q'] ?? '';

// Prepare with two parameters
$stmt = $mysqli->prepare(
  "SELECT t.txn_id, a.acct_id, t.amount, t.txn_type, t.note, t.ts
   FROM transactions t
   JOIN accounts a ON a.acct_id = t.acct_id
   WHERE a.user_id = ?
     AND t.note LIKE CONCAT('%',?,'%')"
);
$stmt->bind_param("is", $uid, $term);
$stmt->execute();
$rows = $stmt->get_result();
```

5 Case Study: MOVEit Transfer Breach

In May 2023, a zero day SQL injection vulnerability called CVE-2023-34362 in MOVEit Transfer’s web API allowed attackers to inject SQL queries to deploy a web shell, and gain administrative access without any valid credentials (Dosumu, 2025). This attack was performed by a ransomware group called Clop that rapidly took advantage of this flaw in MOVEit’s web API by automating payload delivery and exfiltrating sensitive data via injected queries before a patch could be applied. The developers of MOVEit released a fix on May 31st 2023, although the exact date is unknown, evidence suggests that the breach began at least 30 days prior. This time frame illustrates how fast this SQL injection exploit can propagate in the wild and how in

only 30 days over 1000 organizations had sensitive data stolen through this specific attack on MOVEit (Dosumu, 2025). This chain of events underscored the dangers of unsanitized inputs in legacy code paths and the need for multiple overlapping defenses in case one fails.

The financial and organizational impact was severe in this case, over 1000 organizations including many government agencies, healthcare providers, and major corporations has data stolen through SQL injection and it is estimated that there was a total loss of over 80 million dollars (Dosumu, 2025). Beyond the direct monetary damage the affected organizations faced regulatory penalties such as class action lawsuits and long term reputational harm. In retrospect a combination of static parse tree validation or taint tracking could have blocked the malformed SQL and prevented this attack from occurring in the first place. The use of prepared statements or mutation based CI test suite would have caught the vulnerability before the release as well. The MOVEit incident highlights that only a layered defense strategy that spans secure coding, real time anomaly detection, prioritized alerting, and continuous regression testing can truly mitigate SQL injection risks.

6 Conclusion

This survey has examined five core defense layers, static and runtime monitoring, machine learning detection, alert prioritization, and CI/CD regression testing, and assessed six foundational studies spanning from 2004 to 2025. Static grammars and taint analysis both offer strong structural guarantees but face drawbacks in performance (Halfond et al., 2006). Parameterized queries neutralize classic injection methods but depend on consistent developer adoption (Boyd & Keromytis, 2004). Machine learning models extend coverage to sophisticated payloads, but may introduce interpretability and resource issues due to training the models (Paul

et al., 2024; Kumar et al., 2024). Alert prioritization and CI/CD integration help to manage alert volumes and shift detection left but rely on accurate risk scoring and comprehensive test suites (Paul et al., 2024; Kumar et al., 2024). The MOVEit breach underscores that partial defense can be overcome at scale, reinforcing the importance of layered defense (Dosumu, 2025). Looking ahead, research must focus more on efficient detection machine learning models, and automated code migration tools in order to catch SQL injection vulnerabilities before they cause any damage.

References

1. **Boyd, S., & Keromytis, P.** (2004). *SQLrand: Preventing SQL injection attacks*. In **Proceedings of the 2nd USENIX Workshop on Offensive Technologies** (WOOT ’04). USENIX Association.
2. **Halfond, W. G. J., Viegas, J., & Orso, A.** (2006). *A classification of SQL-injection attacks and countermeasures*. In **Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering** (pp. 13–15).
3. **Shar, L., & Tan, K.-L**. (2013). *Defeating SQL injection*. *IEEE Software*, 30(3), 10–15.
4. **Paul, A., Sharma, R., & Olukoya, O.** (2024). *SQL injection attack: Detection, prioritization & prevention (SQLR34P3R)*. *Journal of Web Security*, 15(2), 45–63.
5. **Kumar, S., Dutta, A., & Pranav, P.** (2024). *Analysis of SQL injection attacks in the cloud and in web applications*. *International Journal of Cloud Security*, 9(1), 22–38.
6. **Dosumu, T.** (2025). *MOVEit data breach: A case study in zero-day exploits and organizational cybersecurity preparedness*. *Journal of Cyber Incident Response*, 1(1), 1–19.