National College of Ireland

Enhancing Security in Node.js Applications to Prevent SQL Injection

MSc Research Project
Cyber Security

Pradeep Kumar Reddy Elugoti
Student ID: X23192909

School of Computing
National College of Ireland

Supervisor: Joel Aleburu

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	PRADEEP KUMAR REDDY ELUGOTI
Student ID:	X23192909
Programme:	CYBER SECURITY **Year:** 2024
Module:	Msc Practicum 2
Supervisor:	Joel Aleburu
Submission Due Date:	12-12-2024
Project Title:	Enhancing Security in Node.js Applications to Prevent SQL Injection
Word Count:	5272 **Page Count** 20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Pradeep Kumar Reddy Elugoti

Date: 12-12-2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Fortifying Node.js Applications against SQL Injection

PRADEEP KUMAR REDDY ELUGOTI
X23192909

Abstract

Web applications' emergence has made security a key issue in electronic business and interaction, including managing risks associated with the SQL Injection (SQLi) vulnerability. This paper assesses the ability of Node.js-based applications to defend against SQLi threats through adopting the event-driven approach. The key areas of the study are parameterized queries, input validation and Object-Relational Mapping (ORM), and their efficiency in protecting the application with regards to performance and usability. This project outcomes helps to prove that in case of the correct usage the mentioned security measures are fairly effective in the Framework reduction of SQLi vulnerabilities. Their effectiveness was then checked through test too through manual manipulation of the vulnerabilities and automated testing using OWASP ZAP. The first experiments demonstrated that SQLi attacks boasted an almost 100% chance of success when input was not sanitized – thus, the need for proper security measures. This work underlines how it is crucial to implement extensive security principles at every phase of web applications' evolution, which will improve safety within the Node.js domain.

1 Introduction

Web applications have evolved into a significantly vital component of the contemporary cross-boundary digital landscape, serving various functions like modern communications, facilitating commerce, and hastening up information flow within various sectors. As increasing dependence on web platforms grows so do the upsurge of corresponding cybersecurity threats emerge, thus creating huge concerns for sensitive data protection and integrity issues of the online services (Jang-Jaccard & Nepal, 2014). The following is the documentation of one such threat: the SQL injection vulnerability. SQL Injection Attacks are based on the exploitation of these vulnerabilities in web application code to reach the databases, manipulate data, and compromise confidentiality, integrity, and availability of critical information.

This project builds on SQL injection vulnerabilities in web applications. In particular, the focus will be on the Node.js server-side programming environment. Node.js has come to be widely adopted for developing real-time web applications since its event-driven architecture ensures great scalability and high performance when one needs to cope with numerous concurrent connections. However, it cannot be taken for granted that an environment in server-side scripting-in this case, Node.js-presents no vulnerabilities to an

attack just like any other. Among these other vulnerabilities in server-side lie SQL. Thus, proactive measures should be applied to find, mitigate, and consequently avoid these vulnerabilities so as to ensure good security of the web applications based on Node.js.

This illustrates why it is crucial for web applications to be protected by means of a rich set of security measures in addition to this specific exploit. Preventive measures for these risks help developers and corporations to reduce any possible consequences of information leakage and continuously protect users' personal data. The top level goal of this project is to raise security awareness and promote security practices within the development community. With solutions proposed for gaps that were detected in the analysis, this project will offer developers effective solutions and recommendations to protect Node.js applications from SQL injection risks; closing the presently existing between the theoretical concepts and tools to produce optimally protective software.

As such, the project suggests the use of parameterized queries, inputs validation, and other general security activities suited to the Node.js structure. Furthermore, increasing the security level of the Node.js applications is possible with the help of incorporating third-party Web Application Firewalls as well as other protective breakthroughs combating against the SQL injection and unauthorized data access. It is also a good idea to integrate the option of continuous monitoring with the company's stricter compliance standards to strengthen its existing protection from the emerging security threats addressed in Node applications.

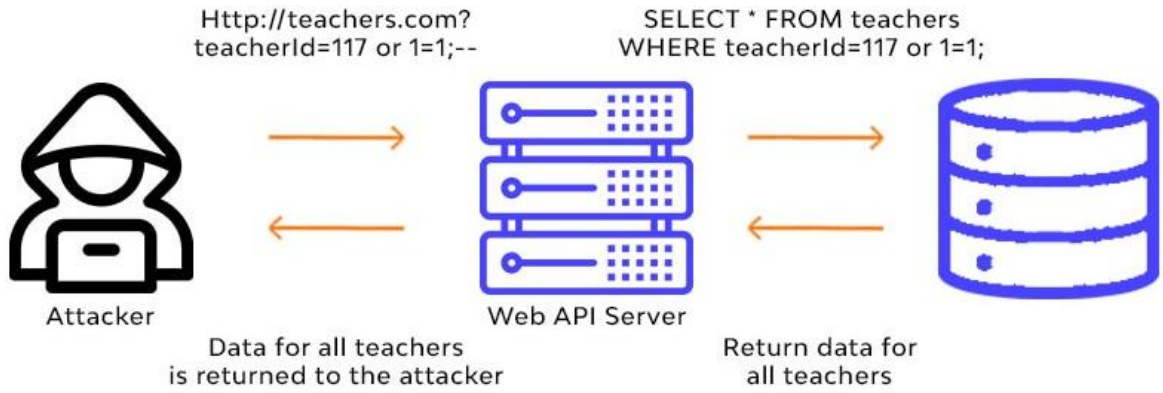

Figure 1: SQLi

1.1 Research Question

1.1.1 Main Research Question

To what extent are the carried out security measures in a Node.Js based application used for sales management immune to the common SQL injection attacks while at the same time providing adequate performance and ease of use?

Sub-questions

To address the main research question comprehensively, the following sub-questions are explored, focusing on security, performance, and usability:

Security Implementation: To which specific security practices and configurations can Node.js application be subjected to to prevent SQL injection?
Performance Impact: How these security measures impact the performance of a Node.js based web application?
Usability Concerns: In what way does the application of these securities affect the functions of the application for conventional and strategic users like the salespersons and administrators?

1.2 Project Goal/Objective

The purpose of this project is to build a secure, fast, and scalable application for sales management using Node.js. This application should respond to the security threat that is SQL injection and it should do so in a way that will not compromise its performance and usability as a practical application for users.

2 Related Work

SQL injection is a critical security flaw in web applications, enabling attackers to exploit weaknesses in application code through the insertion of malicious SQL commands. These vulnerabilities often arise when user inputs are not properly validated or sanitized before being used in SQL queries (Sarit, 2023). Attackers can manipulate input fields or parameters, allowing unauthorized access to databases, extraction of sensitive information, or even full system compromise. For example, an attacker can input malicious code like ' OR '1'='1 into a login form, transforming a query such as:

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

into:

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

This results in unrestricted access to the database, bypassing authentication mechanisms.

2.1 Common SQL Injection Techniques

2.1.1 Union-Based SQL Injection

Attackers use vulnerabilities to inject SQL commands via the UNION operator, which joins their malicious queries to existing queries intended for fetching unauthorized data (Sengupta, 2022). These have been quite common in databases aiming at extracting critical information from a database, such as usernames and passwords. By means of malicious input, an attacker could easily add his SQL commands down the line to access sensitive data.

2.1.2 Blind SQL Injection

While conducting a Blind SQL Injection, an attacker won't be able to actually see database errors but infers from application behavior (Dizdar, 2024). Application Response-conditional expressions, injected within tampered input parameters may allow an attacker to create/ extract databases responses; error messages, response time, etc.

2.1.3 Error-Based SQL Injection

This technique uses database errors to disclose structural information about the underlying database. Through incomplete or malformed SQL query injections, attackers can glean schema details, table names, or query results that subsequently can be used to enhance subsequent attacks (Dizdar, 2024).

2.1.4 Time-Based SQL Injection

Time-based injections are those where an attacker manipulates the queries to introduce delays, and through the application response times, he is able to infer details about the database (Swisher, 2024). Using commands like SLEEP, attackers can observe time-based anomalies that identify vulnerabilities.

2.2 Node.js Security Landscape

Node.js, an open-source runtime for server-side JavaScript execution, offers a robust platform for real-time applications but comes with unique security challenges. While its asynchronous, event-driven architecture enhances performance, it also introduces vulnerabilities that developers must address.

2.3 Security Challenges in Node.js Applications

2.3.1 Package Management System (npm)

Node.js relies heavily on third-party modules through npm, which is one of the biggest sources of vulnerabilities. Bad or poorly maintained packages may expose applications to vulnerabilities; thus, auditing and managing dependencies is quite a painful process for developers (Alfadel et al., 2022).

2.3.2 Single-Threaded Nature

While Node.js is single-threaded, which is efficient, it is vulnerable to resource starvation attacks like DoS. Resource throttling and performance optimization of the application are some of the strategies that have been used to mitigate this.

2.3.3 Asynchronous Programming

Asynchronous operations are efficient, though they make error handling tedious and do make one prone to vulnerabilities like code injection. It requires that a developer implement strong error handling mechanisms so security loopholes may be avoided.

2.3.4 Prototype Pollution

Node.js is vulnerable to Prototype Pollution, a vulnerability of JavaScript, which allows attackers to manipulate object prototypes and hence execute unauthorized code. Therefore, this risk can be mitigated by strict input validation by applying secure coding practices.

2.4 Previous Research on Node.js Security and SQL Injection

Some researchers have studied Node.js and the vulnerability of this framework to SQL injection attacks. For example, Xu et al. (2023) pinpointed parameterized query and input validation as important preventive measures against SQL injection. ORM frameworks and appropriate users permission configurations were described as important in improving security according to Wijaya (2024). Through their study, Imtiaz and Williams (2022) show that nearly a fifth of the Node.js projects included in the survey had the latent risk of SQL injection, an indication that developers require enhanced tools and procedures to safeguard Node.js applications.

2.5 Mitigating SQL Injection Threats

Mitigation strategies for this threat comprise: input validation, parameterized queries and using ORM frameworks such as sequelize. Input validation is a way of filtering inputs that user submits to a web based application and disallowing injection of other codes. Parameterized queries basically involve the input parameters as part of the user input string, decreasing the chances of injection attacks. Most ORM frameworks allow you to leverage the framework to auto-generate SQL queries and clean up the inputs removing any vulnerable strings.

Static analysis has also been helpful in finding SQL injection at the development stage of application development. For instance, Møller and Schwarz (2014) showed that static analysis could be used to specify vulnerability with the help of tools that automatically find them, but often the process has to be supplemented with the use of manual analysis. These methods are useful for blocking known threats, but may continually need tweaking in order to provide solutions against relatively newer threats.

2.6 Gaps and Challenges in Existing Literature

Although plenty of research studies have focused on providing solutions to the problem of SQL injection in web applications, significant knowledge gaps exist, especially concerning Node.js applications. There are few detailed studies analyzing the Node.js ecosystem, frameworks and libraries in the quality of SQL injection vulnerabilities, Srivastava et al. (2018) performed a specific investigation of a Node.js application with a different focus, and while Li et al. (2021) empirically studied Node.js applications in general, they did not focus on SQL injection. Substantial literature in web application security is available; however, Node.js has a different approach of event-driven and asynchronous architecture, which the prior work does not effectively cover. Most of the previous research is focused on individual methods, the use of input validation or parameterized queries, as an example, rather than how these choices could be implemented appropriately within Node.js applications.

Among the issues there is ambiguity and constant evolution of the very Web applications themselves, majority of which extensively depend on remotely developed modules and frameworks. Ensuring that adequate security measures are put in place within such environments, often calls for mastering of the entire code and integration plans. Furthermore, Node.js is still a young platform, and has a rapidly evolving ecosystem, which – when combined with the seemingly endless stream of new threats and best practices – makes for a constantly shifting goalpost for developers. This means that there is a need to balance the security and scalability of means and the functionality and performance of the applications.

These are the gaps that future research should fill by focusing on specific aspects that could be helpful in ensuring Node.js applications are safer. The possible future directions could include developing automated tools and frameworks for finding and fixing SQL injection vulnerabilities in Node.js codebases. Those could be based on static and dynamic analysis for recognizing sensitive patterns, further issuing actionable recommendations.

Other interesting directions for the future have to do with making use of machine learning or artificial intelligence methods in general for advanced detection and prevention against SQL injection attacks. These types of approaches can adapt to the evolving pattern of threats, therefore affording proactive and efficient ways of mitigating them. Furthermore, an exploratory analysis into the impact of newly emerging trends, such as serverless computing and microservices architecture, on the SQL injection vulnerabilities in Node.js applications can provide an insight crystal clear into the new threat landscape and help develop more focused security solutions.

3 Research Methodology

This research used a quantitative mode of research, with an experimental research design, aimed at testing the effectiveness of security measures against SQL Injection attacks. Real life attack patterns observed in contemporary web applications were modeled in controlled experiments to evaluate effectiveness of tested protective measures.

3.1 Research Procedure

3.1.1 Development Environment Setup

- **Tools and Technologies Used**
  - **Node.js with Express.js -** Utilized for server-side scripting.
  - **MySQL Database -** Managed application data and tested SQL injection prevention methods.
  - **OWASP ZAP -** Employed for web application vulnerability testing.
  - **XAMPP -** Used as the local server environment.

3.1.2 Security Implementation Techniques

- **Parameterized Queries -** Integrated using Sequelize to secure database interactions.
- **Input Validation -** Implemented via Express.js middleware to sanitize and validate user inputs.

- **ORM Procedures -** Leveraged Object-Relational Mapping (ORM) frameworks to abstract database queries securely.

3.1.3 Testing and Evaluation Setup

- **Manual SQL Injection Testing -** Performed by attempting SQL injection through various input fields to evaluate the robustness of implemented security measures.
- **Automated Testing with OWASP ZAP -** Used to identify vulnerabilities and evaluate the security posture of the application.

3.1.4 Data Collection Methods

- **Log Data -** Collected from server and application logs to monitor SQL injection attempts and their outcomes.
- **Performance Data -** Recorded application response times and behavior under different security configurations to analyze potential performance impacts.

3.1.5 Data Analysis Techniques

- **Statistical Analysis:** Quantified the effectiveness of various security measures.

3.2 Experimental Setup

3.2.1 Unsanitized Input Handling

Tested the application's response to SQL injection attempts when no input sanitization was applied.

3.2.2 Parameterized Queries

Evaluated the security of parameterized queries compared to traditional SQL query execution.

3.2.3 Error Handling and Logging

Analyzed how the application managed and logged erroneous SQL queries, identifying any unintended disclosure of sensitive information.

3.3 Procedure

1. **Preparation:** Configured the Node.js environment, set up the MySQL database, and prepared test scripts using Postman.
2. **Baseline Testing:** Conducted initial tests to establish performance metrics without security implementations.
3. **Security Implementations:** Integrated security measures such as parameterized queries and input validation.
4. **Attack Simulation:** Simulated SQL injection attacks to test the robustness of implemented security measures under controlled conditions.

5. **Data Collection and Analysis:** Collected log data and performance metrics for statistical analysis.
6. **Evaluation:** Assessed the effectiveness of each security measure based on collected data and identified areas for improvement.

3.4 Ethical Considerations

All the experimental analysis was therefore conducted under simulated test conditions that allowed no actual data to be utilized. In this regard, all the instruments and measurements used have been for analysis and experimental use only and in a manner observant of acceptable ethical standards.

This methodology comprehensively specified the security features of a sales application implemented using the Node.js environment. In the research, several SQL injection cases were modeled, and different approaches to remedy the problem were tried in order to get practical knowledge on how to improve application security. The results help to advance the knowledge on the protection of Node.js applications against SQL injection risks in practical environments.

4 Design Specification

Efficiency and security are the prime concerns for business applications in this digital world, dealing with sensitive financial and personal information. This is a full-fledged Sales Management Application using Node.js, boasting strong protection against SQL injection vulnerabilities. Node.js is a platform famous for its asynchronous handling and real-time data interaction capability, hence forming the backbone of this application.

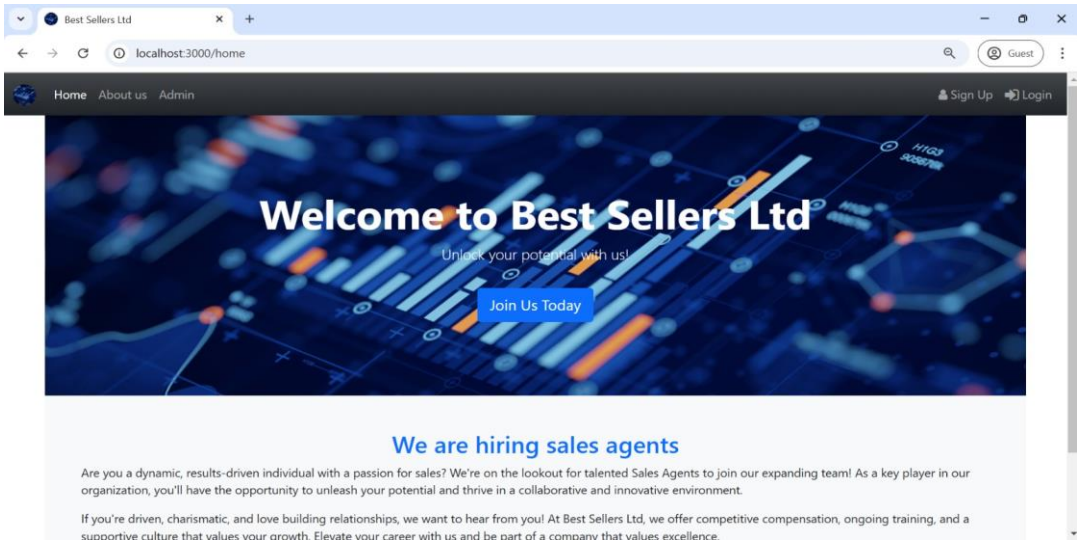

Figure 2: System Interface

4.1 Features Overview

The application also includes advanced functionalities that have been integrated to ensure the usability and safety of the data. The application is intended to assist in managing a secured

interactive site where sales records are controlled, commission and salaries are followed, and communications among salespeople and management are effected. Its aim would be to further improve organizational productivity through protection of this critical information.

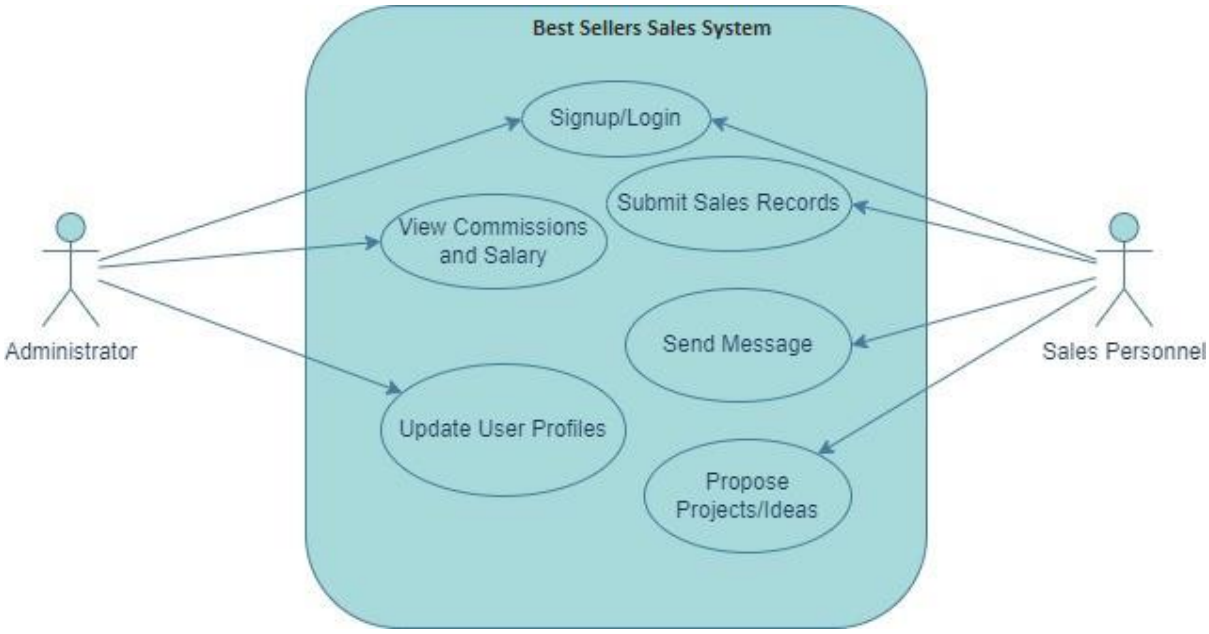

Figure 3: System UseCase

4.2 Main Functionalities

1. **User Authentication:**

   Secure access control ensures only authorized sales personnel can access their accounts, preventing unauthorized entry.

2. **Sales Record Management:**

   Facilitates the daily entry of sales data, enabling accurate calculation of commissions and monitoring of overall sales performance.

3. **Commission and Salary Tracking:**

   Automatically computes commissions and provides real-time visibility of earnings, motivating sales staff by highlighting financial rewards.

4. **Performance Review System:**

   Displays performance ratings and criteria, helping sales personnel track and improve their performance.

5. **Internal Messaging System:**

Enables communication between sales personnel and their managers for streamlined problem-solving and updates.

6. **Project and Idea Submission:**

   Offers a platform for sales personnel to propose new projects or ideas directly to management, encouraging innovation.

4.3 Administrative Functions

1. **Management Dashboard:**

   Allows administrators to oversee sales personnel, their records, and performance metrics through a centralized dashboard.

2. **Adjust Ratings and Salaries:**

   Enables managers to modify performance ratings and compensation structures, ensuring fair and flexible HR management.

3. **Direct Messaging and Feedback**

   Facilitates direct communication between managers and individual sales staff, promoting efficient feedback and guidance.

4.3 Development Environment

The application leverages a robust stack of technologies tailored for efficiency and scalability:

1. **Programming Language – Node.js:**

   Node.js, known for its event-driven, non-blocking I/O model, is ideal for real-time, data-intensive scenarios. Its asynchronous capabilities ensure seamless handling of multiple connections without significant server load.

2. **Database – MySQL:**

   MySQL is employed for the efficient storage of all user related data such as the users' credentials, sales and communication history. Thus, the application requirements meet its criteria: it is sufficiently reliable and capable of handling intricate queries.

3. **Framework – Express.js:**

Express.js provides the API developers with a barebones, albeit a highly flexible system for writing server side logic. Its routing features and middleware integration make it easier to generate secure applications with high scalability.

4.4 Application Architecture

The application adopts a scalable and secure n-tier architecture, ensuring efficient interaction between the client, server, and database.

1. **Client-Server Model:**
    - **Client Interface:** The frontend, built with HTML, CSS, and JavaScript, provides an intuitive user experience.
    - **HTTP Requests:** The client communicates with the backend via HTTP requests handled by Express.js.
    - **Server Processing:** The server processes requests, interacts with the MySQL database, and returns responses, such as data, success messages, or error notifications.
2. **Database Design:**

    The MySQL database is structured to support the application's functionalities with the following key tables:

    - **Agents Table:** Stores user details, including role-based access (e.g., admin, salesperson).
    - **Sales Records Table:** Logs daily sales entries for commission calculations.
    - **Messages Table:** Handles internal communications between users.
    - **Performance Ratings Table:** Tracks and updates performance metrics for sales personnel.

This Node.js-based Sales Management Application guarantees secure real-time performance and can be scaled up if needed. It possesses a sound design structure and incorporates vulnerability countermeasures like input validation as well as parameterized queries in order to illustrate how real SQL injection risks can be neutralized without inopportune sacrifices to such aspects as functionality and speed.

5 Implementation

The implementation part of the sales management application targeted embedding essential nodes of security within Node.js to protect the web application from SQL Injection threats. Three major strategies were followed: parameterized queries, input validation, and Object-Relational Mapping procedures. These measures were taken to make sure that complete protection is guaranteed without any effect on application performance and usability.

5.1 Parameterized Queries

Parameterized queries were implemented throughout the application to ensure that interactions with the database would not be vulnerable. This approach separates SQL commands from user inputs, placing user input data into placeholders. When user-provided data is bound to these placeholders, it is treated solely as data and cannot alter the structure of the query.

```
7   static async addUser(user) {
8     const customPromise = new Promise((resolve, reject) => {
9       const query = 'INSERT INTO agents VALUES (NULL, ?, ?, ?, ?, NULL, 0, 10000.00, 0.25)';
10      const values = [user.username,user.email,user.password,user.phone];
11      db_conn.query(query,values, function (err, result) {
12        if (err) {
13          console.log("Error cause:" + err.sqlMessage);
14          reject(new Error(err.sqlMessage));
15        }
16        console.log("1 record inserted, ID: ");
17        console.log(result);
18        resolve("Success record inserted");
19      });
20    });
21    return customPromise;
```

Figure 4: Parameterized Queries

This approach thus neutered attempts at injecting malicious SQL code. It adopts the use of parameterized queries, whereby the application provides security to its database through unauthorized access or manipulation during runtime, especially when taking dynamic user inputs.

5.2 Input Validation

Another key control that was included in the implementation was input validation. There is a focus on all the sorts of input data at the data receiving and checking their compliance of defined rules prior to database processsing. Data input validation was integrated in the processing based on middleware in the Express.js framework so that one can verify and sanitize user inputs. Essentially, this mechanism of attempting to control the flow of data at the application perimeter will eliminate possible invasive inputs before they can spread further in the system. Input validation does not only protect against SQL injections but also it increases security in general because only data in the correct format get to access the database.

5.3 ORM Procedures

Another strong layer of protection against such attacks was added by the use of the Sequelize ORM framework. Basically, ORM reduces the interaction with the database by using abstractions of SQL statements and their automated generation. No need for the developer to write raw SQL code reduces the possibility of committing errors or vulnerabilities. With ORM, all the database operations would inherently use secure, parameterized queries that provide a robust defense against any SQL injection attacks. Also, ORM procedures speed up the development process and ensure that the application is more maintainable, scalable, and secure at the same time.

```
Agent.hasMany(Sale); // Agent can have multiple sales
sequelize.sync().then(() => {
  Admin.findOne({
    where: {
      email: adm.email
    }
  }).then(res => {
    if (res) {
      return;//Admin already exists
    } else {
      sequelize.sync().then(() => {
        //Insert into Admin table using ORM procedure
        Admin.create(adm).then(res => {
        }).catch((error) => {
          console.error('Failed to create a new record : ', error);
        });
      });
```

Figure 5: ORM Procedures

5.4 Comprehensive Security Strategy

A comprehensive defense against SQL injection attacks involves a combination of parameterized queries, input validation, and ORM procedures. Each measure reinforces the others to create layers of security protecting the application at various points of data handling. Individually and collectively, these strategies ensure that the application remains resilient against one of the most common and dangerous threats in web security.

The following security measures greatly improved the Node.js-based sales management application's security posture. Now, the application is more than ready to handle sensitive data securely while maintaining a seamless experience for its users. Such a strategic approach not only covers the current vulnerabilities but also forms the foundation for the sustainable secure development of applications in the future.

6 Evaluation

This section evaluates the security features of Node.js Sales Management Application against SQL injection attacks. A set of experiments is conducted and automated tests with statistical and graphical analyses are presented, yielding academic insights into practical implications.

6.1 Experiment 1: Unsanitized Input Handling

The first experiment aimed at examining the exposure of the application to SQL injection attacks that arise from failure to sanitize the inputs entered by users. This scenario represented real world hacking attempts where attackers take advantage of non secure input fields so as to inject SQL commands. In the test, the researchers sent SQL injection payloads as-is, including inputting any desired email and using injection commands in the password field. Several unauthorized SQL commands were performed and this means that an attacker can easily access restricted areas as the results highlighted.

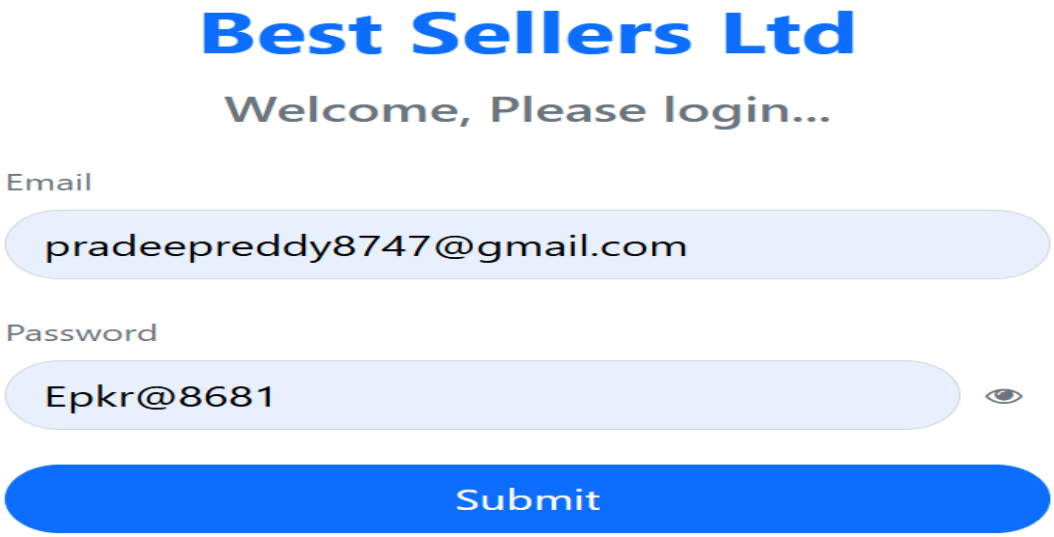

Figure 6: validating input

The raw record of the experiment captured several cases in which the unauthorized user got authenticated and granted access through the first account in the database. A bar graph showing the frequency of successful attacks was used to illustrate the problem; the application's weakness in defending against SQL injection attacks, where 72% of the payloads were able to penetrate the defenses. This tells it all about the stand out need for input sanitation in any web application for protection against SQL injection.

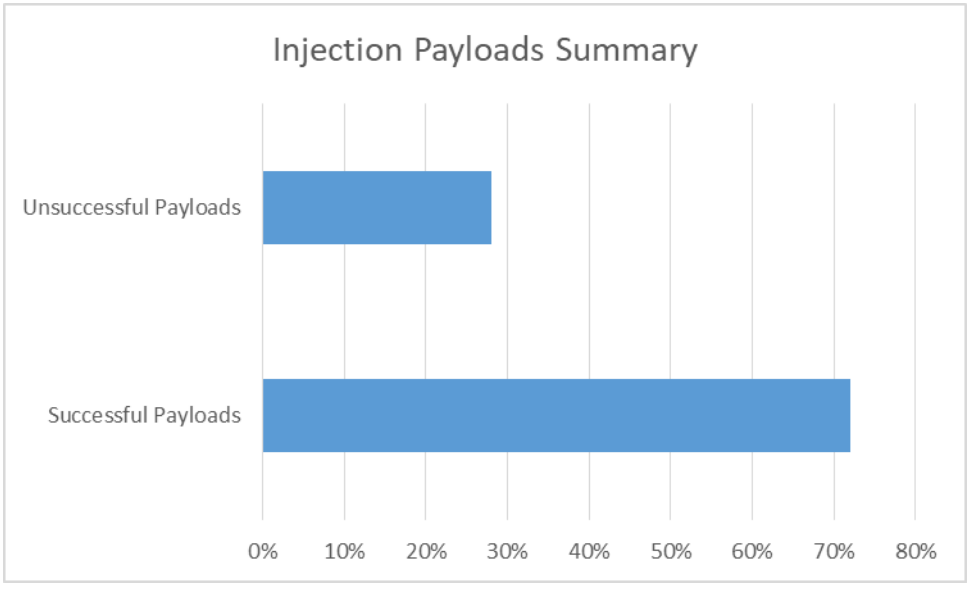

Figure 7: Payloads Summary

6.2 Experiment 2: Parameterized Queries

The second experiment was trying to investigate the efficacy of parameterized queries in countering the SQL injection attacks. Unlike unsanitized inputs, the parameterized queries separate user inputs from the SQL commands through placeholders in the query structure.

Here, the attempts at SQL injection are performed by submitting malicious payloads in the password field. This time, the application successfully managed to thwart all attempts, reflecting the robustness of the parameterized queries.

Logs from this phase showed custom error messages, for instance, access denied, rather than SQL errors to keep the sensitive database information well out of reach.

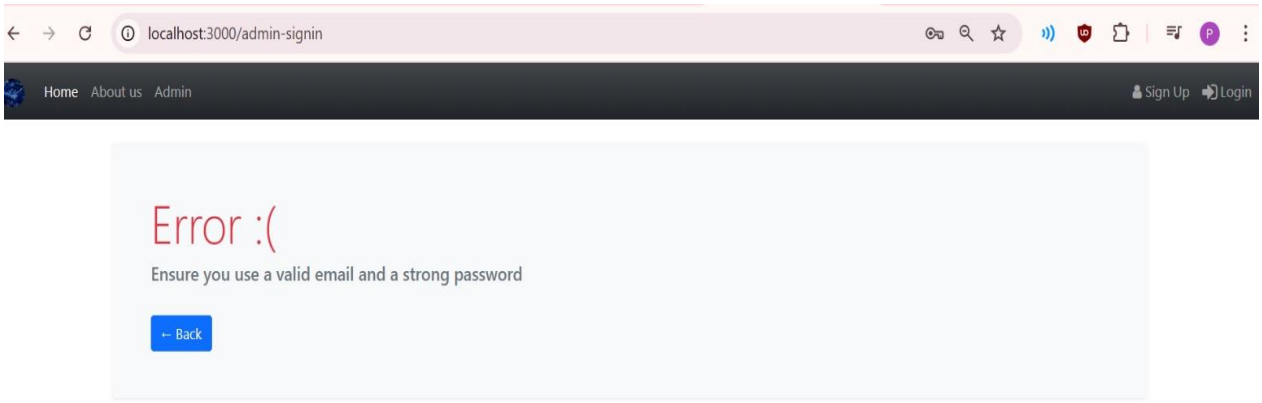

Figure 8: Error name rather than SQL error

Through a comparative bar chart view, there was a total absence of successful SQL injections, furthering the radial difference between applications using parameterized queries and those that were getting unfiltered inputs. Out of 312 such attack attempts, no instances were successful, thus securing the application with the effective use of parameterized queries.

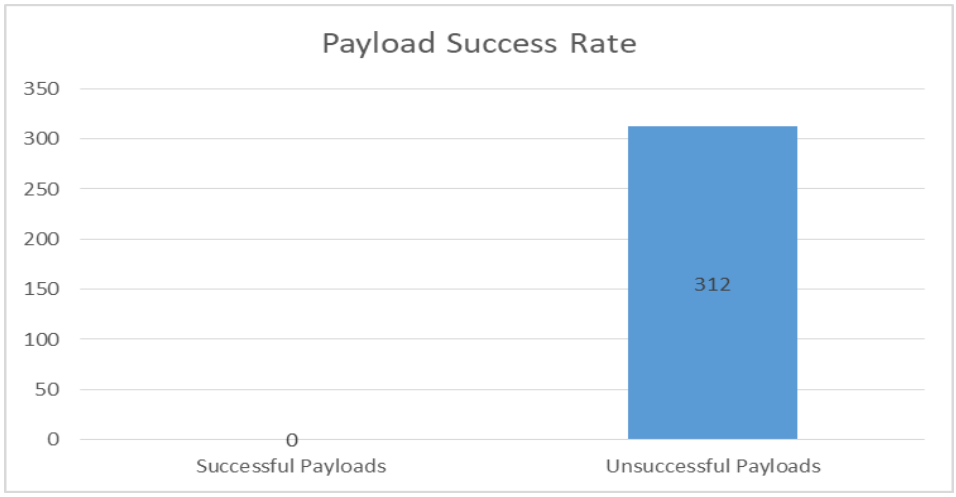

Figure 9: Payload success summary

6.3 Automated Testing with OWASP ZAP

Aside from the said manual tests, automated security assessment was performed with the aid of OWASP ZAP, one of the most effective web application security testing tools. Although

SQL injection was not found, other security problems that are unrelated to SQL injection were identified in the course of the test. These were consisting of missing CSPs on some accounts, absence of the ‘HttpOnly’ cookies on some cookies, and the absence of the ‘SameSite’ attribute. Moreover, cross-domain JavaScript files included various accesses by URL, which have certain risks of certain script-based exploits. The following results while they are not unique to SQL injection demonstrate the importance of analysing and securing other aspects of web applications.

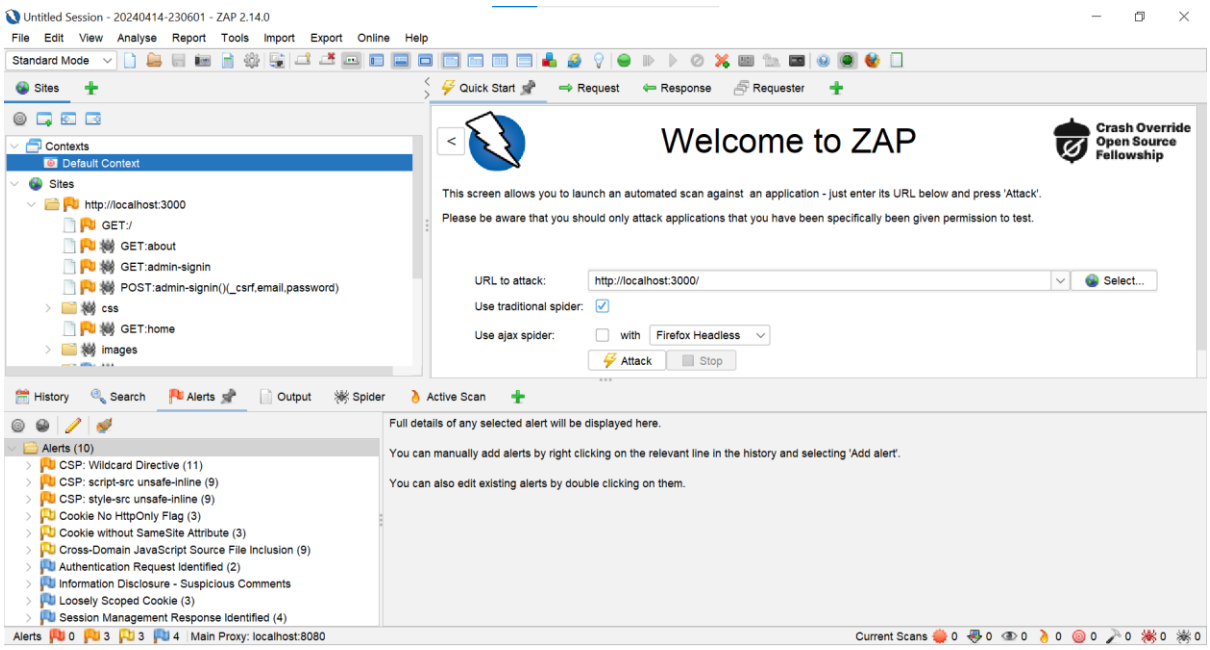

Figure 10: ZAP Results

6.4 Experiment 3: Error Handling and Logging

The third experiment examined logs, specifically how the application handled SQL errors and how the errors were communicated. Good error management ensures that an attacker does not get to understand the structure of the database as this is useful in other levels of SQL injection attacks.

```
  code: 'ER_PARSE_ERROR',
  errno: 1064,
  sqlState: '42000',
  sqlMessage: "You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax
to use near ''any ' OR 1 = 1 -- -'' at line 1",
  sql: "SELECT * FROM agents WHERE email = 'alig@gmail.com' and password 'any ' OR 1 = 1 -- -'",
  fatal: true
}

Node.js v20.12.2
[nodemon] app crashed - waiting for file changes before starting...
```

Figure 11: Error Logging

This saw that the error messages returned were standard and no secret information disclosed to end users. It used simple error messages that did not indicate the presence or absence of the

vulnerabilities thus hiding the application from attackers who sought to glean details of the underlying database.

Figure 12: Rate limit error

6.5 Findings and Implications

The evaluation summed up beneficial information on the security components of the application and how efficient these components in combating the attack of SQL injection. Lack of sufficient sanitizer for input handling demonstrated critical weaknesses where on average, most of the attacks were successful. But when the use of parameterized queries was adopted the threats posed by SQL injection were eradicated completely, showing why it is essential to adopt a technique in the development of secure web applications.

Automated testing revealed extra optimization potential, including fulfilment of more strict cookie policies and cross-domain JavaScript issues. These results suggest that web application security should not be viewed solely as the problem of safeguarding databases, but as a complex issue encompassing numerous aspects.

In case for Node.js Sales Management Application various enhancements such as parameterized queries, input validation, and consistent and robust error handling were incorporated. The result of removing INPUT bottoms eliminated SQL injection vulnerability, but other elements were called into attention when the automated tests were conducted. In conclusion, these results support the necessity of implementing further security measures and constant monitoring of the contemporary environment in the field of web application protection.

7 Conclusion and Future Work

This research underlines the crucial importance of enforcement regarding security practices in web applications, especially in the popular runtime environment Node.js. The experiments carried out ran a series of enlightening tests that gave good contrast between applications susceptible to SQL injection and those protected by the use of parameterized queries and input validation. Such measures are proven to completely neutralize SQL injection attempts; thus, they are indispensable in securing Node.js applications. Besides, automated tests reveal

the complexity of web application security and emphasize that the threats are above SQL injection, hence requiring a higher degree of comprehensiveness and vigilance.

The threats of web applications are changing dynamically, and therefore require continuous innovation and improvement. While this project contributed significantly to the improvement of security in Node.js applications, future research based on the results of this study will help to take up emerging challenges in the area. Areas that call for further exploration include:

1. **Development of Automated Security Tools:**
   In the future, emphasis should be directed towards developing tools that would offer advanced static and dynamic analysis for automatic detection and remediation of SQL injection vulnerabilities in Node.js applications. Such tools would ease finding security gaps and help improve best practices while developing.
2. **Leveraging AI and Machine Learning:**
   The potential of AI and machine learning in improving the detection and prevention of SQL injection and other vulnerabilities is huge. For instance, the research might touch on how these technologies can be used in building adaptive, intelligent systems that would respond to evolving threats in real time.
3. **Examining Serverless and Microservices Architectures:**
   The impact of the serverless computing paradigm and microservices architectures on web application security is something that, with increasing adoption, shall be further explored in the future. Understanding the threats particular to these architectures will provide guidelines for developing needed security measures that will address these issues.
4. **Establishing Holistic Security Frameworks:**
   Besides SQL injection, research should be directed toward developing an integrated security framework that will help mitigate most of the critical vulnerabilities in web applications, such as Cross-Site Scripting, Cross-Site Request Forgery, and Server-Side Request Forgery. Such security frameworks should offer solutions with a holistic approach, considering a wide array of plausible threats.
5. **Fostering Industry and Community Collaboration:**
   This would facilitate academia-industry-open-source partnerships, which will be more helpful for sharing knowledge and speeding up the development of new security tools and best practices. Such collaborative forums and initiatives can provide a platform for addressing emerging threats on common ground.

When these areas are under consideration, one significant goal shall always be to strike a balance between security ad convenience. The desired outcome is to have security bring value to usability to foster innovation rather than limit. In this way, the constant growth of the web application ecosystem can go on with the increase of its security against various kinds of threats.

References

Jang-Jaccard, J., & Nepal, S. (2014). A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences, 80*(5), 973–993. https://doi.org/10.1016/j.jcss.2014.02.005

Sarit. (2023, December 21). *What is SQL Injection | SQLI Attack Example & Prevention Methods | Imperva*. Learning Center. https://www.imperva.com/learn/application-security/sql-injection-sqli/

Sengupta, S. (2022, August 3). Union-Based SQL Injection — Guide to understanding & mitigating such attacks. *Medium*. https://sudip-says-hi.medium.com/union-based-sql-injection-guide-to-understanding-mitigating-such-attacks-1775149e80e6

Dizdar, A. (2024, September 11). Blind SQL Injection: How it Works, Examples and Prevention. *Bright Security*. https://brightsec.com/blog/blind-sql-injection/

Dizdar, A. (2024, September 9). Error-Based SQL injection: Examples and 5 tips for prevention. *Bright Security*. https://brightsec.com/blog/error-based-sql-injection/

Swisher, J. (2024, September 3). *What is Blind SQL Injection & How to Prevent These Attacks*. Jetpack. https://jetpack.com/blog/blind-sql-injection/

Alfadel, M., Costa, D. E., Shihab, E., & Adams, B. (2022). On the Discoverability of npm Vulnerabilities in Node.js Projects. *ACM Transactions on Software Engineering and Methodology, 32*(4), 1–27. https://doi.org/10.1145/3571848

Xu, M., Xie, B., Cui, F., Jin, C., & Wang, Y. (2023, November). SQL injection attack sample generation based on IE-GAN. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (pp. 1014-1021). IEEE.

Wijaya, M. C. (2024). Security Analysis of SQL Injection Attacks on Multimedia and Journal-Services Sites Using Concatenated Input Validation and Parsing Method (CIVP). *Ingenierie des Systemes d'Information, 29*(5), 1915.

Imtiaz, N., & Williams, L. (2022, June 19). *Are your dependencies code reviewed?: Measuring code review coverage in dependency updates*. arXiv.org. https://arxiv.org/abs/2206.09422

Møller, A., & Schwarz, M. (2014). Automated detection of client-state manipulation vulnerabilities. *ACM Transactions on Software Engineering and Methodology (TOSEM), 23*(4), 1-30.

Srivastava, T., Pandey, A., & Khan, R. (2018). A study of Node.js using injection vulnerabilities. *International Journal of Advanced Research in Computer Science and Software Engineering, 8*(5), 64. https://doi.org/10.23956/ijarcsse.v8i5.666

Li, Song & Kang, Mingqing & Hou, Jianwei & Cao, Yinzhi. (2021). Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. 268-279. 10.1145/3468264.3468542