SQL Injection Prevention and Mitigation

Final Project Deliverable 2 (Group 10)

CS 6348
Data and Applications Security

Praneeth Reddy Penamalli (PXP210057), Abhiram Galla (AXG230010), Avaneesh Ramaseshan Baskaraswaminathan (AXB220230) Naveen Prashanna Gurumurthy (NXG230002) Syxri Mazoch (ALM200009)

Task Summary

Praneeth Worked on designing the web pages and general research and design of the system. Also created the input sanitization and integrated the DistilBERT model into the web application.

Abhiram Worked on implementing user access control for the system and prepared statements as well as the creation of the Powerpoint slides for the demo.

Avaneesh Created the log analysis tool, as well as performing exploratory analysis using ML with Naveen.

Naveen Worked on deployment of the system, as well as testing and evaluating the system. Also performed exploratory analysis using ML with Avaneesh.

Syxri Designed and wrote the final report, including doing research to find related work. Also tested the system.

Motivation SQL injection is a critical vulnerability for any web application that stores data - as even one successful SQL injection attack comes at great cost to the company and its developers. However, SQL injection defenses are mostly reliant on the developers safeguarding against these attacks manually by either sanitizing and validating all input data, or structuring queries in such a way that protects against injections [1] - the latter of which may not always be possible in cases where a large, complex system already exists. We were therefore motivated to design a solution that defends against SQL injection attacks both accurately (protecting against what developers may miss) and without requiring a complete redesign of the system's queries. Our implementation can be used either on its own, or as an addition to existing defenses in current web applications.

Timeline

Beginning Steps In order to complete the project, our first steps were to create the main components of the website and database to test on. This included the user authentication flow pages (allowing users to register, log in, and log out), the database configuration page, and entry point routing. We also began to prepare the database for training the LLM model.

Completing the (Main) Website and Model Training We then finished creating the remaining pages for the website (such as activity logging) and trained the LLM model on the SQL injection attack database we created.

Sanitization and Logging Next, we implemented a logging schema to capture query structure, user ID, timestamp, and origin. In addition, we finalized and refined our sanitization filters such that they were robust and accurately prevented against SQLi attacks.

Securing the Website To further protect our application, we hardened security in several areas including PHP configurations, using HTTPS, and simulating a Web Application Firewall via Apache.

Final Wrap-Up Lastly, we conducted extensive penetration testing using SQLMap and custom payloads, and finalized our role-based-access control implementation for the database. We evaluated the performance of our system, and then used that information to create our demo presentation and our final report.

Introduction

As a part of Application Security (05), SQL Injection (SQLi) is one of the most critical vulnerabilities faced by developers of web applications. It allows attackers to manipulate a website's database by injecting malicious SQL queries through user inputs, often without the knowledge or consent of other users. This can lead to severe data breaches and compromises that expose sensitive information such as usernames, passwords, credit card details, and personal records. The damage caused by SQL injection attacks can result in financial losses, reputational harm, legal implications, and violations of data protection laws, making the prevention of SQLi crucial for any organization that stores sensitive data.

From a developer standpoint, preventing SQLi can be costly as all user inputs that interact with the database must be validated and checked, and it is easy for something to be missed.
This project aims to address these issues by developing an advanced security system that detects, prevents, and mitigates SQL injection attacks (including the related domain of XSS attacks) - effectively safeguarding web applications. The system focuses on two primary mechanisms to bolster security: automated input validation and sanitization, along with an LLM model that will detect SQLi attacks. These features will provide administrators with tools to easily and effectively block attacks and detect potential threats through in-depth forensic analysis.

Related Work

Over the last two decades, many solutions to the problem of defending against SQLi attacks have been proposed. One early solution was SQLrand (2004), which defends against attacks by appending a random key to all SQL statements (such as SELECT, FROM, TO) [2]. These SQL statements are then read by an intermediate proxy and validated before being translated to query the database. While effective, this provides significant overhead, and is not possible to implement in already existing systems. In addition, the key must be remembered by the developers when designing new queries. Unlike SQLrand, our implementation provides much less overhead and does not require a complete overhaul of the system.
AMNESIA, developed in 2005, augments Java code to check all queries at runtime and checks them against a pre-built model to decide whether or not the query is allowed at that location [3]. Aside from the differences in language, AMNESIA notably cannot defend against stored-variable attacks [4]. Since our implementation is based on machine learning, stored-variable attacks are easily defended against by training the LLM on malicious inputs containing stored-variable attacks.
In 2020, a paper was published with a novel technique for defending against SQL injection - one that uses the Knuth-Morris-Pratt string matching algorithm[5]. In this implementation, user-inputted strings are matched with the patterns of malicious injection strings in order to detect malicious inputs. If a malicious input is found, then the user is blocked from the webpage and data about them is logged, including the type of attack. However, this requires the algorithms to be inserted and checked manually in the system. This makes it hard to apply to existing systems, and in certain cases may come with a high overhead. Our implementation, being based on machine learning, is independent of the system itself. In addition, if new attack inputs are found, they can simply be fed to the underlying model in order to train it on those type of attacks.
Lastly, a neural-network based implementation was also published in 2020. This implementation discusses two different models and compares their detection of SQL injection inputs [6]. We expanded on the ideas brought in this work by pairing our detection model with log analysis, allowing our system to also detect other unusual activity patterns that may lead to an attempted SQL injection attack.

Implementation and Results

Implementation The website we created in order to demonstrate our system was done in the programming language PHP. The website is designed such that users and administrators can log into the system. XAMPP is used for hosting the web server. Database management is done through MySQL, allowing us to create role-based access control in order to further harden the database against attacks.

In order to prevent SQLi programmatically, we created a variation of our website that contains prepared statements (as opposed to direct queries) and input sanitization before input is passed to the queries.
Large Language Models, especially those based on the Transformer architecture, have revolutionized NLP. Compared to traditional DL methods, LLMs are much larger and exhibit stronger language understanding. So, we used LLM to predict whether an injection attack is happening. Large Language Models, especially those that are based on the Transformer architecturem have revolutionized NLP. Compared to traditional DL models, LLMs are much larger and exhibit stronger language understanding. Therefore, our model for detecting and logging SQLi attacks (on the variation that does not use prepared state- ments) was a pre-trained model called DistilBERT. The model was then further trained using Supervised Fine-Tuning on a SQL injection dataset we created in order to detect SQL injection

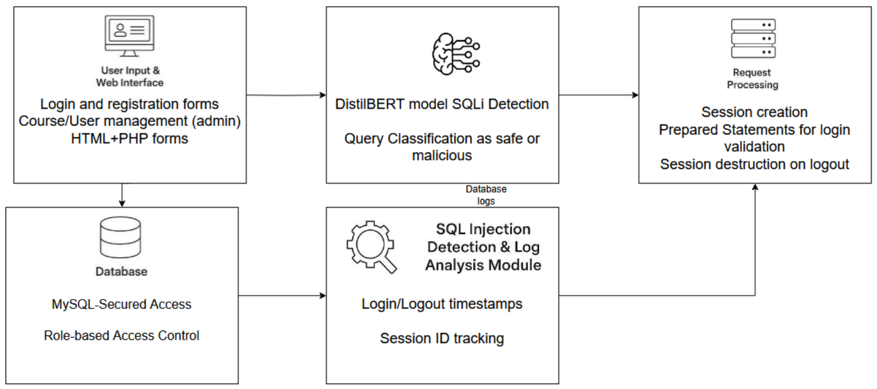
User Input & Web Interface
Login and registration forms
Course/User management (admin)
HTML+PHP forms
DistilBERT model SQLi Detection
Query Classification as safe or malicious
Request Processing
Session creation
Prepared Statements for login validation
Session destruction on logout
Database
MySQL-Secured Access
Role-based Access Control
Database logs
SQL Injection Detection & Log Analysis Module
Login/Logout timestamps
Session ID tracking

Figure 1: System Architecture

attacks. Once an attack is detected, it will log the attack and prevent the input from passing through to the query. This model obtained an accuracy of 99% on our testing set.

Security Principles Addressed:

Confidentiality Confidentiality was a major part of the design of our project, as fundamentally the goal is to prevent unauthorized access to the database. SQLi attacks are detected and logged before they are ever passed to the database, preventing leaks. Queries are also sanitized beforehand, leading to only valid queries being sent.

Integrity In our system, the integrity of data sent by the user may be violated by the user themselves. Therefore, we monitor all incoming inputs before they are ever sent to the database. This ensures that when data is read from the database, the data can always be trusted as it has not been tampered with through unauthorized access.

Authentication Authentication is done in our demo website through the log-in process. Before a user can see any information, they must log in with the correct username and password. As there is no system to register, the user must already be a part of the existing group of users. If a user fails authentication in some way, they only see a message stating that their credentials are invalid. This way, no information is leaked to the user about what information exists in the system before they are authenticated. In addition, the database provides authentication itself. If the incorrect log-in information is given, a user cannot connect to the database.

Non-repudiation We address non-repudiation in our system by way of logging information about when users last logged on and off the system, as well as logging information when a SQLi attack is performed. This helps prevent non-repudiation from being violated as we maintain information to positively identify users and their actions.

Results Our results show that SQL injection attacks are mitigated in our systems, both in our version that uses prepared statements and input sanitization and in the version that uses the DistilBERT model. Evaluation of the model shows that DistilBERT has 99% accuracy after being trained on the given dataset. This is on par with the accuracy given in the neural-network based implementation [6], showing that our system's performance is comparable to similar works in the field.

Conclusion

Our implementation provides security for web applications by defending against SQLi attacks. In one variation of our system, these attacks are mitigated by a strong filtering mechanism that sanitizes all user

input before sending it to a parametrized SQL query - effectively blocking SQLi attacks. We also trained an LLM model, DistillBERT, to detect SQLi attacks and successfully prevent them from occurring.
Future work is to be done on expanding the activity logs to include more information, such as the user's IP address and their number of failed login attempts. In addition, we would like to add enhancements to our role-based access to extend from a simple "user" and "admin" to more nuanced roles, such as instructor and auditor. We would also like to ideally implement a robust access control list to further defend the database. Additionally, we would like to combine the two systems we currently have (ML and sanitized inputs) in order to create multiple layers of security for our web application, making the chances of a successful SQLi attack extremely low.

Implementation Code

The code for the vulnerable system (without the LLM model or other defenses) can be found here.
Code for the implementation of the LLM model is here.
Finally, the variation of our system including prepared statements can be found here.

References

[1] OWASP, "Sql injection prevention." `https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html#defense-option-1-prepared-statements-with-parameterized-querie` 2025.

[2] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security* (M. Jakobsson, M. Yung, and J. Zhou, eds.), (Berlin, Heidelberg), pp. 292–302, Springer Berlin Heidelberg, 2004.

[3] W. G. J. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, (New York, NY, USA), p. 174–183, Association for Computing Machinery, 2005.

[4] M.-Y. Kim and D. H. Lee, "Data-mining based sql injection attack detection using internal query trees," *Expert Systems with Applications*, vol. 41, no. 11, pp. 5416–5430, 2014.

[5] O. C. Abikoye, A. Abubakar, A. H. Dokoro, O. N. Akande, and A. A. Kayode, "A novel technique to prevent sql injection and cross-site scripting attacks using knuth-morris-pratt string match algorithm," *EURASIP Journal on Information Security*, vol. 2020, p. 14, Aug 2020.

[6] P. Tang, W. Qiu, Z. Huang, H. Lian, and G. Liu, "Detection of sql injection based on artificial neural network," *Knowledge-Based Systems*, vol. 190, p. 105528, 2020.