Vulnerable Web Application For the Analysis of SQL Injection

A Technical Report Submitted to the Department of Computer Science

Presented to the Facult of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Zane Belkhayat
Spring 2021

On my honor as a University Student, I have neither given nor received
Unauthorized aid on this assignment as defined by the Honor Guidelines
For Thesis-Related Assignments

Advisors
Yuan Tian, Department of Computer Science
Mohammad Mahmoody, Department of Computer Science

Vulnerable Web Application For the Analysis of SQL Injection

Zane Belkhayat
University of Virginia
zb4fw@virginia.edu

ABSTRACT

SQL injections are one of the most dangerous vulnerabilities currently plaguing the internet. They can lead to leakage of private information to bad actors, leading to further damage in the form of identity theft, financial fraud or other possible crimes. The central question is to find out how to prevent these attacks from occurring in the first place. Previous research has shown that while only a small percentage of websites are vulnerable to SQL injections, they are extremely damaging when taken advantage of. For this reason, I plan to research and investigate the common ways SQL injections are performed and defended against. This research will be performed through a combination of online investigation and practical application, ie. creating a prototype webs application which relies on an SQL database vulnerable to SQL injections with various countermeasures against such a threat. From this, I hope to advance knowledge in regards to this common form of cyber-attack.

1 INTRODUCTION

When SQL injection attacks occur, they leave large amounts of data vulnerable to bad actors. Though a relatively straightforward mistake to remediate, a surprisingly large amount of web sites/applications are at risk of SQL injection. When these attacks occur, they can lead to massive breaches in data resulting in millions lost for those compromised web sites. In 2015, a toy company named Vtech suffered a massive data breach due to an SQL injection attack. This exposed 5 million parent accounts and 6.3 million child profiles. Though a massive data breach, it is far from uncommon for companies to lose user data like this. The same type of attack affected Sony, the U.S. Navy, and the department of homeland security [3].

Currently, there is little emphasis placed upon the security of databases throughout the educational process. In order to remedy this, I propose the creation of a tool which can be used for educating software engineers on the dangers and countermeasures of SQL injections. This tool will be of significant use to students, educators, and learners of all ages. It will allow users to learn penetration techniques and defense mechanisms in a no stakes environment, advancing knowledge from the basics of SQL injection to more advanced methods.

2 SQLI BACKGROUND AND RELATED WORK

SQL injection occurs when user input is somehow utilized in the formation of an SQL query which is then executed. Though websites which are vulnerable to SQL injection have become more uncommon, when they occur, they are often used to devastating effects. Depending on the size and popularity of a web application, this can leave millions of users' data exposed to attackers. This often leads to millions in losses as well, with even minor injection attacks costing upwards of hundreds of thousands [3]. In addition to losses made while trying to fix the vulnerability, companies often face litigation afterwards due to a variety of reasons such as improper data storage and security. Considering the threat level of these vulnerabilities, it is imperative that engineers take them into account when designing systems which rely on SQL databases.

Similar works to the proposed tool include purposefully vulnerable web applications such as DVWA (Damn Vulnerable Web Application). Though these provide some methods of testing SQL injection techniques, none are dedicated fully to exploring the variety of countermeasures and attack methods included under the umbrella. This tool seeks to fill that niche by providing an application which showcases a breadth and depth of this particular vulnerability not seen in others.

3 LITERATURE REVEW

In order to provide a basis for the proposed tool, prior literature in the field has been reviewed and summarized. In the report A Classification of SQL Injection Attacks and Countermeasures, SQL vulnerabilities are classified by their mechanism. These groups are known as "injection through user input," "injection through cookies," "injection through server variables," and "second order injection" [1]. In a full release of the tool, all mechanisms of injection would be explored, but for the prototype it was decided to narrow focus to user input and second order injection, allowing for more exploration of methods of attack and countermeasures.

In the same report, methods of attack are classified by their intent, given a description, and an example. The following section of attack methods summarizes some examples classified in the report, while also modifying a mechanism to be included in these methods. In the case of the vulnerable web application, it makes more sense to consider second/first order injection as methods of attack, allowing users to differentiate between their subsequent consequences. However, as mentioned in the report it is imperative to realize that SQL injections are often combinations of attack methods, and defense against one does not provide defense against all [1].

In the section following, titled methods of defense, some countermeasures from the report and an article titled SQL Injection Cheat Prevention Cheat Sheet from the OWASP organization are summarized. These include basic methods which can be implemented by the average developer in little time to help prevent SQL injection attacks. Considering the educational goal of helping programmers who are new to SQL query safety, it was decided more advanced defense techniques would be unnecessary to explore in the tool.

4 ATTACK METHODS

There various types of SQL injections which can occur, lead to different outcomes and therefore different methods of defense. These types can be classified in a variety of different ways and the basic classification scheme for the prototype tool will be expanded upon below. The following variations of SQL injection are some of the more common attacks and what can be utilized/examined through the use of my vulnerable web application.

Blind Injection: One of the more important methods of defending against SQL injection attacks includes limiting user information. If a user can get the exact error that occurred when they attempted to pass in an SQL string, it makes cracking the database much easier. For this reason, web applications often **limit error reporting**. This makes it so the SQL injection must be performed "blind." Though this is a roadblock in the process, a dedicated attacker can get around this issue through a variety of methods [3].

First Order Injection: First order injections occur when the input of a user is immediately placed into a vulnerable SQL query. In this case, attackers can expect to see immediate results from their injection, resulting in exposed data, or modifications to the database table as soon as they submit their input [3].

Second Order Injection: Second order injections occur when a web application utilizes information already within the database in a vulnerable SQL query. Developers often explicitly trust the information currently contained within the database, and this results in a problem when hackers input tainted information which gets stored. Though there is not an immediate breach of information/modification to the table upon input, the attacker can cause damage later down the line either directly or to another user when their SQL query is pulled from the database and executed [3].

Encoded Injection: Some methods of defense attempt to detect SQL query statements in a user's input. Attackers can get around this by encoding their query statements through a variety of methods. Utilizing URL encoding it is possible to avoid detection of characters commonly used in SQL statements such as an apostrophe. This allows attackers to inject web applications even if they have some form of input protection [3].

5 METHODS OF DEFENSE

Just as there are a variety of methods for attack, there are a variety of methods used in counter measures. These are known as defensive coding practices and are used to harden applications against injection attacks [3]. Below are common methods of defense against SQL injection, some of which are implemented in my vulnerable web application for testing.

Limit Error Reporting: This is the countermeasure most often employed to begin the process of defense against SQL injections. Disallowing users from seeing the errors caused by their input outside of specific cases (ie. improper password formatting etc.) keeps important

information hidden from attackers. This forces them to do **blind injection**, which can still be dangerous to a website, but is a significant impediment to the process of attack [3].

Input Sanitizing: This countermeasure seeks to eliminate risks by sanitizing user input. In this case, user input which contains characters or strings which are deemed as a threat may be blocked, or have the risky data removed. Ideally this will remove all SQL query statements from a user's input but there are work arounds such encoding the attack or wrapping the input so portions which may be deleted still form an SQL query [3].

Escaping Input: Similarly to input sanitizing, the method of escaping the input attempts to eliminate risks by modifying user input. However, in this case characters which may lead to SQL injections are escaped, making them part of the string instead of the command. Also, similarly to input sanitizing, this method can be avoided through encoding, or exploiting vulnerabilities in the escape method, such as escaping the escape [2].

Parameterized Input: One of the most effective countermeasures to SQL injection attacks is parameterizing the input. Supported by most modern languages, this simple method allows a user to specify an SQL statement with blank variables, and then supply user input as a parameter to those queries. In this fashion, user input can never be utilized or interpreted as an SQL query (unless utilized again elsewhere in an unsafe manner) [2].

Limit Database Permissions: A good practice defensive measure, limiting database permissions makes it so not every user can access or modify the database table at will. Disabling specific methods of modifying the database can help reduce system harm when an attacker does manage to inject SQL code to a web application [2].

Use of Stored Procedures: Similar to parameterized input, using stored procedures can prevent user input from ever being treated as something other than a parameter. The difference between the two is that SQL code for stored procedures is stored within the database itself. However, because their implementation is up to the developer, it is possible for poorly implemented/utilized stored procedures to end up causing SQL injection vulnerabilities [2].

Input Type Checking: This method of defense is often forgotten as most user input comes in the form of a string. However, type checking provides a powerful tool especially when user input is needed for any type of number. By making sure user input is the necessary type, it can prevent SQL injection attacks entirely in some cases, as they cannot be formed with invalid characters such as apostrophes and characters [2].

Encoded Input: Encoding a user's input achieves a similar result to parameterizing it. By encoding a user's input in some method such as hex, it disallows any special characters from being input into an SQL query string. This creates an impasse for would be attackers, as it becomes impossible to add characters that could be interpreted as part of the SQL query. This is not considered best practice however, and it is recommended to use input parameterization instead [2].

6 SYSTEM DESIGN

Taking note of similar tools to aid in developing defensive and penetration testing skills, I modeled my example web application similarly to the DVWA (Damn Vulnerable Web App). This utilizes an event driven client server architecture in order to respond to user input in an appropriate manner with feedback. The tool is designed to use a server, interacting with a database on the backend. It will allow user input to various forms using different measures of defense against SQL injection. Advanced countermeasures such as dynamic runtime analysis and precise taint tracking were deemed useless to the intended audience of beginners in the field of SQL security, and unfeasible for the scope of the project. Considering the tool's original goal as an educational enhancement, it will be released through GitHub as an open-source project.

Utilizing Amazon Webservices, I created a Microsoft Windows Server 2016 (t3a.xlarge) utilizing a 2017 Microsoft SQL Server Express. The database itself is set up to simulate a basic record of user healthcare information, with columns for names, providers, addresses, etc. After connecting my web application to the database, I began developing different pages with varying levels of security to test the various methods of SQL injection and defense. The first page which greets a user upon navigating to the website is a guiding page to various SQL forms where countermeasures are used in various ways.

For the first challenge page, I created a completely insecure "register user" button. This takes direct user input and concatenates it to an SQL query string which is then executed. In addition, when something goes wrong, the error message is displayed to users without regard for origin, allowing them to see if their SQL query was

malformed and take additional measures. This would represent a completely vulnerable web application, which an attacker can exploit with ease.

For the second challenge page, I removed the error message display. This allows a user to test blind SQL injection methods, where they must use alternative techniques to secure data and the like. Both the first and second challenge pages test second order injection.

The fourth and fifth challenge pages are utilized for second order injection testing. They both eliminate first order injection by using parameterized input for whatever the user supplies. However, on the fifth page, information from the database is pulled and concatenated to an SQL query. This input can be controlled from the fourth page.

The sixth challenge page utilizes a type of poor input sanitizing where a string is looked over once for some common SQL query language, and if found this is removed from the string. However, an intelligent attacker can user this to form further SQL queries by writing commands which form after the sanitization process.

The seventh challenge page utilizes a vulnerable input escaping method which attempts to escape single quotes with a backslash wherever they are placed in a string. However, this can be exploited if the attacker uses another backslash in front of their single quote. This will escape the escape, allowing the attacker to exploit the vulnerability.

Some techniques were deemed unfeasible/not implemented for this initial prototype of the tool. For example, SQL injection where the parameters are passed by URL was not included, due to the additional web application architecting which would need to take place. During planning, a button which reset the database to its initial state was also considered. However, this feature proved to be more complicated than initially envisioned and was scrapped.

8 RESULTS AND CONCLUSIONS

After presenting the web application to various peers with some guided, they concluded that the experience was educational. They universally agreed it broadened their understanding of how web applications defend against SQL injection attacks as well as how they may still be vulnerable. On the negative reports, it was mentioned that the web application was not the most eye-catching experience, and did not seem tailored toward those completely new to the realm of SQL injection.

One clear conclusion from the research and implementation of this tool is that utilizing parameterized input is the most effective way to protect a web application against SQL injection vulnerabilities. Though other countermeasures provide some level of defense, when improperly implemented they are all vulnerable to modified SQL injection in some way. Parameterizing input acts as a one size fits all solution in systems which can support it.

Beyond educational value, this web application has the potential to spark social change in the web application development sphere. By emphasizing security during the creation of SQL queries, one can take a spend time now, save money later approach. Considering the disastrous consequences of even minor SQL injection vulnerabilities, it is important to foster a culture of security mindedness throughout the development process. This application helps to bring about those outcomes.

9 FUTURE WORK

Given more time to work on the project, I would like to introduce more methods of attack and defense for further learning outcomes. For example, in my research for SQL injections, I came across another vulnerability type called cross site scripting. This vulnerability allows an attacker to cause scripts to run automatically on a victim's machine when they access a trusted website. Ideally, I could expand my challenge pages to take into account various methods of attack for this vulnerability and countermeasures. Features which were scrapped/passed over due to time constraints would also be viable, such as a database reset button in case something breaks due to user input, as well as http based injections, where parameters are passed via the URL.

There is also the possibility of further exploration for SQL injection. Some defensive techniques which were mentioned did not make the list of challenge pages. There countermeasures to SQL injection attacks include use of stored procedures and limiting database permissions, which come with their own unique drawbacks. Given the scope of the project, these additions were deemed unfeasible. Further development of the tool would include a challenge page utilizing these defensive measures.

On the architecture side, I would like to remake the web application utilizing different services to minimize the cost of running. For example, there is cost associated with running both the server and database for the application currently. Ideally, the application would come as an easily installable package, which connects to a low/no-cost database. This would allow a larger audience to access the web application, opening up its educational opportunities to more people.

Finally, in keeping with the theme as an educational tool, I would work on the user interface for a cleaner and more responsive web application. Blind injections could be a toggle as well, allowing a user to decide whether they want direct error reporting or not. Other options could be added too, such as a button to reset the database in case experimentation led to some sort of large malfunction. Additionally, due to complaints by some users about the lack of beginner friendly content, the challenge pages could come with tutorials in the form of hints as to how/why certain methods of SQL injection can crack the page.

REFERENCES

[1] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. 2006. A Classification of SQL Injection Attacks and Countermeasures. Retrieved April 22, 2021 from https://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf

[2] OWASP. SQL Injection Prevention Cheat Sheet. Retrieved April 24, 2021 from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[3] The Cyber Team. 2015. VTech Hack Reminds Us that SQL Injection Can Have Serious Consequences. Retrieved April 25, 2021 from https://coar.risc.anl.gov/consequences-of-sql-injection-attacks/#:~:text=Although%20mitigating%20this%20vulnerability%20is,incur%20costs%20that%20exceed%20%24196%2C000.