h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

University of Applied Sciences

– Department of Computer Science–

Web application firewall to filter out SQL injection attacks

submitted by

Feldbusch, Dennis

matriculation number 770265

Referent :

ABSTRACT

The increasing number of SQL injections highlights the need for effective web application security measures. These injection attacks are a particularly common and damaging type of attack. In order to mitigate this risk, the objective of this work is to research the question of how reliable blacklist-based regular expressions can be to block SQL injection attacks. In order to answer this question a web application firewall (WAF) was developed which uses multiple regular expressions blocking SQL injections. The regular expressions are based on perceptions of literature review as well as documentation of common databases. With the aim of accomplishing the previously mentioned objective, two test cases were developed, which tests the positive and negative scenarios.

The test results show that the WAF is capable of identifying 95.6% of actual SQL injection attacks, which demonstrates that the vast majority of the tested injections were matched by the regular expressions used. However, the WAF also blocks 0.56% of legitimate requests, which highlights the need of continuous refinement.

In order to achieve a more accurate result, one possible direction for future work could be the limitation to a single database management system. Consequently, the regular expressions do not have to match multiple variations of functions, special chars and so on. As a result, the detection rate can be increased, while the false positive rate can be decreased.

CONTENTS

1 INTRODUCTION

1.1 PROBLEM DEFINITION AND RELEVANCE

SQL injection attacks are a prevalent and critical threat to the security of web applications and the data they store. This type of attack allows an attacker to execute malicious code in a web application's database, potentially resulting in the unauthorized access, manipulation, or theft of sensitive data. [1] As the number of web applications continues to grow [2], the threat of SQL Injection attacks has become increasingly pressing.

Given the severe consequences of a successful SQL Injection attack, the development of an effective defense mechanism is of utmost importance. One such defense mechanism is the use of a web application firewall (WAF) that can detect and prevent these attacks.

1.2 OBJECTIVE

The objective of this project is to create a reliable web application firewall (WAF) based on blacklisting regular expressions, designed to detect and prevent SQL injections. Furthermore, it aims to evaluate blacklisting-based rules by filtering out SQL injections based on SQL injection patterns. This goal can be summarized in the following research question:

> "How reliable is a regular expression based rule to block SQL injections in a web application firewall?"

To fully assess the reliability of this approach, a series of tests will be conducted. These tests will provide a clear picture of the WAF's detection rate and false positive rate, which are important indicators of its overall effectiveness. The results of these tests will contribute to the efforts to mitigate the risk of SQL injections.

1.3 STRUCTURE AND APPROACH

The structure of this work is based on the previously stated objective. It is divided into six chapters, with the current chapter 1 representing the introduction. Chapter 2 shows the technical foundations relevant to the work. The essential terms include web application firewalls, SQL injections, as well as regular expressions. The implementation of the web application firewall, including the utilization of regular expressions, is outlined in chapter 3. In addition, chapter 4 describes the test case configuration and summarizes the results of the tests conducted. Based on the theoretical foundation, the implementation as well as the test results, the subsequent evaluation takes place

in chapter 5. The final chapter 6 contains both a conclusion and an outlook, which is intended to raise questions and suggestions.

2

THEORETICAL FOUNDATIONS

The theoretical foundations serve as the basis for further elaboration of the topic. A comprehensive understanding of these principles is essential for the accurate interpretation and analysis of the relevant terminology. The objective of this chapter is to enable the understanding and evaluation.

2.1 WEB APPLICATION FIREWALL

Firewalls are one way to secure internal services which are accessible from the internet. There are four common types of firewalls. [3]

- Packet Filtering Firewalls
- Circuit Level Gateways
- Application Level Gateways
- Stateful Multilayer Inspection Firewalls

Mostly, the third type of firewall is used to protect web applications [1] [3]. Such a firewall analyzes all traffic from and to a web application. Therefore, it has to be placed between the server which should be protected, and the internet. Due to the fact that the requests from the client are sent through the internet and the WAF has to inspect the whole traffic, the WAF is the first point of contact for the client. Figure 2.1 presents an overview of this architecture.

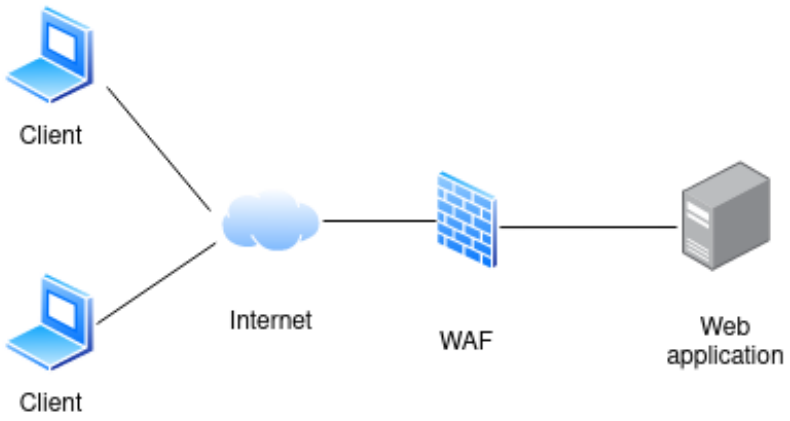

Figure 2.1: WAF architecture overview (own representation)

1 A web application is an application accessible through the web browser [4], [5]. With this kind of application it is possible to provide nearly every kind of functionality [5]. Examples for this are online forms, shopping carts, video streaming, social media, games, or e-mail services [5].

The objective of a web application firewall is to identify all possible threats to a web application [6]. This is achieved by applying rules, which can be positive or negative. The positive rules define traffic, that is allowed in the web application, whereas negative rules define forbidden traffic [6]. Since WAFs work on the applications layer, they can analyze the request in plain text [7]. This is what differentiates this type of firewall from the others listed above. One exception is the Stateful Multilayer Inspection Firewall, which operates not only on the application layer but on multiple ones [3].

2.2 SQL INJECTION

The Structured Query Language (SQL) is a language used to interact with a database. It is often used within a web application to communicate with a database to store, read, edit or delete data [4]. Mostly, those SQL statements also process user input to make it more dynamic and interactive.

For this reason, it can be vulnerable to SQL injections. This means that users can extend the statements by submitting maliciously crafted user input to generate new valid SQL statements that gets executed [1].

Consider the following SQL statement presented in listing 2.1 which selects all data of a user with the `name` provided by user input.

```
String user_input = URL.query;
String query = "SELECT * FROM user WHERE name = '" + user_input + "';";
```

listing 2.1: Example definition of a SQL statement in Java

This example shows the concatenation of the variable `user_input` and the statement defined in quotes. When a user inputs the `name` *dennis*, the `SELECT`-statement returns the dataset for user *dennis*. However, if the user inputs a string such as `dennis' OR 1=1 --` the following statement results:

```
SELECT * FROM user WHERE name = 'dennis' OR 1=1-- ';
```

listing 2.2: Example of an extended query

Since the `--` represents a comment in SQL the rest of the statement is ignored and so represents a valid query. Thus, the statement returns all stored information in this `user`-table, which might reveal personal data of users. Furthermore, attackers can modify the user input, so they can delete data or read data from other tables. According to [1], one user input scenario might be the query part of the URL.

In order to mitigate the risk of a SQL injection attack there are multiple ways to do so. One of them is the use of prepared statements [1]. The idea is to eliminate the risk which comes with special chars and causes the majority of successful SQL injections [1]. Prepared Statements use bind variables which ensures any input to be treated as data and not as instructions to the database [1].

```
PreparedStatement ps = (SELECT * FROM user WHERE name=?);
ps.setString(1, URL.query);
```

listing 2.3: Example usage of prepared statements in Java

However, [1] states that prepared statements alone are not able to prevent SQL injections. Even more important is to have multiple security mechanisms consistently applied such as escaping the variables and the validation of the data [1].

2.3 REGULAR EXPRESSIONS

Since the WAF should be based on regular expressions, it is important to understand the basics of this concept.

Regular expressions are a sequence of characters, which can fulfill several tasks such as matching patterns, replacing text, or searching patterns in a text. In summary, they are used to process any form of text. [8, Chapter 1] Specifically, regular expressions can be used to search for specific patterns of characters, such as a particular sequence of letters or numbers, or to search for more complex patterns, like email addresses, or phone numbers. With this kind of pattern matching, regular expressions can be used to validate user input. One of the key features of regular expressions is their ability to use metacharacters to represent sets of characters. For instance, the period . character represents any single character, while the asterisk * character represents zero or more instances of the preceding pattern. Furthermore, regular expressions can use quantifiers, which specify the number of times a pattern should occur. [8, Chapter 2]

As there is no official definition of regular expressions, there exists a wide range of variants. The regular expression used in this work follows the Perl flavor.

The following example shows a regular expression which can be used to match a mobile phone number.

```
^(0|\+49)[1-9][0-9]{8,10}$
```

listing 2.4: Example of a regular expression

The regular expression in listing 2.4 matches a mobile phone number in Germany. This has to start with either a zero or a +49 and have to be followed by a number between one and nine. Attached to this pattern the regular expression describes a block of eight to ten occurrences of any digit.

3

IMPLEMENTATION OF THE WAF

Although there are numerous attempts and measures to mitigate SQL injections, the number of attacks is increasing [2]. One explanation for this could be that it is difficult for developers to implement all the necessary security measures. For this reason, this work tries to figure out other ways to prevent this vulnerability. The idea behind this is to mitigate the risk of forgotten or falsely implemented security measures. This section describes the implementation of a web application firewall, developed for this work. This can help a developer to alleviate some security concerns, allowing them to focus more on the development of the application itself.

To check all incoming requests to a web application, the WAF has to run between the client and the server. Therefore, the firewall needs to listen on a specific port. The firewall then checks all requests sent to the specified port. Consequently, the WAF blocks the request if it matches any of the regular expressions. Otherwise, if none of the regular expressions match, the firewall forwards the request to the intended endpoint.

REGULAR EXPRESSIONS

As explained in section 2.2 one attack vector is the query part of the URL. In order to mitigate the risk of SQL injections caused by queries, the regular expression tries to identify those injections. Therefore, the WAF extracts the query part of the URL and checks it against multiple regular expressions, which tries to match SQL injection patterns. When receiving a request, its query is parsed into a string without the leading ? character. In order to unify the comparison with the regular expressions the query then is decoded whereby all the encoded variants are presented through their decoded characters. Since the work is limited to the identification of SQL injection in the query part of URLs, the objective of the regular expressions is to match only those strings that are potentially malicious. This work has developed and proposed three regular expressions in order to mitigate the risk of SQL injections. The ideas and concepts that underlie these expressions are referred to in previous works, namely [9] and [10].

```
1)  '|--|;|#|(/\*.*\*/)|\|\||<|>|!=

2)  (?i)(EXEC|CHAR|ASCII|BIN|HEX|UNHEX|BASE64|DEC|ROT13|CONVERT|CHR)
    .*\(.*\)

3)  (?i)(UNION.*SELECT)
```

listing 3.1: Regular expressions used to identify SQL injection attacks

The intent of the first regular expression is to match special characters which could be interpreted by the database. [10] and [9] lists multiple special characters that could affect or be interpreted by the database. The first regular expression matches the following character sequences: `(')` `(--)` `(;)` `(#)` `(||)` `(<)` `(>)` `(!=)` as well as `(/**/)`. Due to the decoding of the query, the regular expression is in no need to match the hexadecimal representation of these characters. However, the = is a special case. The regular expression does not match this character due to the fact that the = is a common sign in URL queries to assign values and would so result in a high false positive rate.

In order to allow the regular expressions to match different spellings, the case is not taken into account by appending `(?i)` to the beginning of the second and third regular expression [8, Chapter 2]. These regular expressions are intended to match special keywords which could cause the database to interpret them. The second regular expression matches the following keywords: `EXEC`, `CHAR`, `ASCII`, `BIN`, `HEX`, `UNHEX`, `BASE64`, `DEC`, `ROT13`, `CONVERT` as well as `CHR`, which are described in [9] and [10]. The `CHR` keyword is a special case since it is not mentioned in [9]. However, it represents the same function as `CHAR` in other databases like Postgres [11], so it was included in this regular expression. In addition to the keywords, all of them have to follow the pattern of an opening and closing bracket `FUNCTIONNAME.*(.*)`, which symbolizes a function call [9].

In contrast, the third regular expression matches `UNION SELECT` statements which are no functions but `SELECT` statements. Matching single keywords like `SELECT`, `DROP`, `DELETE`, or `UNION` is more likely to produce false positives. For this reason, the third regular expression only matches strings which contains the composed keywords `UNION` followed by a `SELECT` [9], [10].

The WAF compares the input string of the query with all three regular expressions. Subsequently, the system blocks the request if any of the regular expressions matches the input string.

4 TEST CASES FOR THE EVALUATION OF RELIABILITY

To evaluate the reliability of the regular expression several test cases were made which are discussed in the following chapter.

4.1 TEST CASE SETUP

To test the regular expressions described in chapter 3, multiple requests are sent to a web application which is protected by the web application firewall. The WAF inspects each request to detect if it contains a SQL injection pattern. Therefore, a client has been developed which takes a word list as argument and crafts HTTP GET requests with the words from the word list as query parameter. These requests are then sent and processed by the WAF. Afterwards, the client counts the number of requests with a status code 400, which represents detected SQL injections by the developed WAF. Finally, it prints the number of blocked requests, which then can be used to calculate the percentage of blocked requests.

The test cases are split into two sets: test set `T_injection`, which contains actual SQL injections and test set `T_valid` which contains requests without SQL injections. `T_injection` includes known vulnerable SQL injections which were taken from [12]. The developed client sends the requests to the WAF in form of HTTP GET requests with the SQL injection as a parameter. A crafted request would look like this:

```
http://localhost:8080/?id=1' OR 1=1
```

Whereas `1' OR 1=1` originates from the word list. Since `T_injection` only contains SQL injections, the ideal case would be to reach a rate of 100% of blocked requests.

`T_valid` includes requests without SQL injections and so represents valid query requests. The requests are taken from Assetnote Wordlists, which generates monthly wordlists from publicly available data like common parameters [13]. Unlike the first set, the second set does not contain any SQL injection. Thus, the ideal case would be to reach a rate of 0% of blocked valid requests.

4.2 TEST CASE RESULTS

The results of the test cases are shown in the following table 4.1.

Test case	Number of requests	Number of blocked requests	Percentage of blocked requests
`T_injection`	5313	5077	95.6%
`T_valid`	301851	1681	0.56%

Table 4.1: Test case results

As the results highlights, the WAF blocks 95.6% of SQL injections. The rest of the requests are not detected as SQL injections and are passed to the web application. Listing 4.1 shows an example a SQL injection attempt which is not blocked.

```
http://localhost:8080/?id=) AND 9170=2793
```

listing 4.1: Example of an unblocked SQL injection attempt

Upon examination of these undetected SQL injections, the majority are requests containing the keyword `AND` which is a SQL keyword to logically connect `WHERE` clauses. However, matching this keyword would result in a high number of false positives, due to the fact that the word `AND` is not uncommon in regular requests.

On the other hand, the WAF blocks 0.56% of valid requests. One example of a falsely blocked request is shown in listing 4.2

```
http://localhost:8080/?id=space;-2
```

listing 4.2: Example of a falsely blocked request

By inspecting the falsely blocked requests, it is recognizable that the vast majority of blocked requests are due to the `;` character. However, since 2014 the W3C no longer recommends the semicolon as query separator [14].

5 EVALUATION OF THE WAF

The developed WAF based on blacklisting regular expressions has been tested in terms of its ability to detect and prevent SQL injections. The results indicate that the WAF is capable of detecting 95.6% of the tested SQL injections, providing a high level of protection against this type of attack. However, it has also been observed that 0.56% of valid requests are being blocked by the WAF.

It is recognizable that the implemented WAF is capable of detecting the majority of SQL injections while only blocking a small number of valid requests. For this reason, the WAF is considered a good solution for preventing SQL injections. However, the results also indicate that the WAF is not 100% reliable and there is still room for improvement. One drawback of this work is that characters like ' are blocked. On the one hand, this character could represent the special character used in databases to symbolize strings and could be misused to cause a SQL injection. On the other hand, the character is also used in the English language and could be used in a valid request as part of a comment text, which would then lead to a false positive. This highlights the need for continuous refinement of the regular expressions to minimize false positives and maximize the detection of SQL injections. Thereby, the objective should be to ensure that the WAF strikes an appropriate balance between security and accessibility.

Blacklist-based WAFs have inherent limitations, including the difficulty of keeping the blacklist up-to-date with new and evolving threats. Additionally, the potential for attackers to evade detection by crafting their attacks in ways that do not match the regular expression patterns poses a risk. Moreover, the demonstrated WAF tries to detect SQL injections of all types of database management systems (DBMS). This includes SQL patterns of multiple SQL databases, which all come with their own characteristics like the `CHR` function in the postgres database, stated in chapter 3. As a result, the WAF must cover numerous patterns in order to achieve a high detection rate. To counteract this, the WAF could be extended to be configurable. Thereby, the user could choose which SQL database is used by the web application and the WAF could then be configured to only cover the patterns of that specific database management system.

Another limitation of this work is the performance and scalability of the WAF. In order to be able to use the WAF in production, several tests have to be made in regard to the performance. Important parameters could be the used programming language as well as the complexity of the used regular expressions.

In conclusion, the developed WAF is capable of detecting a high number of SQL injections. This characteristic makes it a good solution for an additional protection measure.

6 CONCLUSION

6.1 SUMMARY

This work presented an approach to the detection of SQL injections in web applications. The proposed solution is based on the use of a WAF that uses regular expressions to detect SQL injection attacks. The WAF is implemented in the programming language Go and can be used as a standalone application. For the evaluation, the WAF has been tested using a set of SQL injection attacks as well as valid requests. The results indicate that the WAF is capable of detecting 95.6% of SQL injections, providing a high level of protection against this type of attack. However, it has also been observed that 0.56% of valid requests are being blocked by the WAF. This demonstrates that it is essential to continuously improve and refine regular expressions to minimize the occurrence of false positives and maximize the detection of SQL injections. Nevertheless, the WAF can be seen as an appropriate additional method to protect against SQL injections.

6.2 OUTLOOK

It is also important to consider the scalability and performance of the WAF, as the increasing complexity and volume of web traffic can place significant demands on the WAF. Robust performance and scalability metrics, such as latency, throughput, and resource utilization, should be regularly monitored and evaluated to ensure that the WAF continues to provide effective protection in a high-performance environment.

In addition, the WAF can be extended to support other types of attacks, such as cross-site scripting (XSS) attacks, as well as other types of web application attacks, such as HTTP request smuggling and HTTP response splitting attacks.

Another possible extension could be the implementation of an AI and machine learning algorithm, which should handle the continuously evolving threat of SQL injections.

To better align the WAF with the application, it can be helpful to define a common set of rules. For instance, a rule can be defined to prohibit the ; character in the parameter of the URL. When testing valid requests, the example in listing 4.2 wouldn't cause a false negative. As a result, this rate could be decreased.

BIBLIOGRAPHY

[1] Neil Daswani, Christoph Kern, and Anita Kesavan. "SQL Injection." In: *Foundations of Security: What Every Programmer Needs to Know*. Berkeley, CA: Apress, 2007, pp. 123–138. ISBN: 978-1-4302-0377-3. DOI: doi:10.1007/978-1-4302-0377-3_8. URL: https://doi.org/10.1007/978-1-4302-0377-3_8.

[2] *Vulnerabilities by type*. URL: https://www.cvedetails.com/vulnerabilities-by-types.php (visited on 01/29/2023).

[3] Stephen Woodall. "Firewall design principles." In: *Computer Networks and Computer Security. Coursework paper, North Carolina State University, USA* (2004). URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0e70de176b7c7fb049794ad29fd448b005001e29.

[4] Mehdi Jazayeri. "Some Trends in Web Application Development." In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 199–213. DOI: doi:10.1109/FOSE.2007.26.

[5] Adam Volle. *Web application*. URL: https://www.britannica.com/topic/Web-application (visited on 01/14/2023).

[6] Victor Clincy and Hossain Shahriar. "Web Application Firewall: Network Security Models and Configuration." In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. 2018, pp. 835–836. DOI: doi:10.1109/COMPSAC.2018.00144.

[7] Namit Gupta, Abakash Saikia, and D Sanghi. "Web application firewall." In: *Indian Institute of Technology, Kanpur* 61 (2007), p. 62. URL: https://www.cse.iitk.ac.in/users/dheeraj/btech/namitg+abakashs.pdf.

[8] J. Goyvaerts and S. Levithan. *Regular Expressions Cookbook*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2012. ISBN: 9781449319434. URL: https://books.google.de/books?id=6k7IfACN_P8C.

[9] Naghmeh Moradpoor Sheykhkanloo. "A Pattern Recognition Neural Network Model for Detection and Classification of SQL Injection Attacks." In: *International Journal of Computer, Electrical, Automation, Control and Information Engineering* 9 (6 2015), pp. 1443–1453. ISSN: 2010-376X. URL: http://researchrepository.napier.ac.uk/Output/690348.

[10] Oluwakemi Abikoye, Abdullahi Abubakar, Haruna Dokoro, AKANDE OLUWATOBI, and Aderonke Kayode. "A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm." In: *EURASIP Journal on Information Security* 2020 (Aug. 2020). DOI: doi:10.1186/s13635-020-00113-y.

[11] *9.4. string functions and operators*. 2022. URL: https://www.postgresql.org/docs/15/functions-string.html (visited on 02/06/2023).

[12] Decal. *WERDLISTS/SQL 3.TXT.XZ at master Decal/Werdlists*. URL: https://github.com/decal/werdlists/blob/master/SQL-injection/SQL_3.txt.xz (visited on 01/26/2023).

[13] Assetnote. URL: https://wordlists-cdn.assetnote.io/data/automated/httparchive_parameters_top_1m_2022_12_28.txt (visited on 01/26/2023).

[14] Erika Doyle Navara, Silvia Pfeiffer, Ian Hickson, Travis Leithead, Steve Faulkner, Robin Berjon, and Theresa O'Connor. *HTML5*. W3C Recommendation. W3C, Oct. 2014. Chap. 4.10.22.6. URL: https://www.w3.org/TR/2014/REC-html5-20141028/forms.html.