Query Log Analysis for SQL Injection Detection

Alexandra Rocha[1], Rui Alves[1] and Tiago Pedrosa[1,2,3]
[1]*Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal*
[2]*Research Centre in Digitalization and Intelligent Robotics (CeDRI), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal*
[3]*Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SuSTEC), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal*

Keywords: SQL Injection, IDS, MySQL, Attacks, Detection.

Abstract: Nowadays, more and more services are dependent on the use of resources hosted on the web. The realization of operations such as access to the account bank, credit card operations, among other operations, is something increasingly common in current times, demonstrating not only human dependence on the internet connection, as well as the need to adapt the web resources to the daily life of society. As a result of this growing dependency, web resources now provide a greater amount of confidential information, making the risk of a cyberattack and information leaking grow considerably. In the web context, one of the most well-known attacks is SQL injection that allows the attacker to exploit, through the injection of malicious queries, access to confidential information. This paper suggests a solution for the detection of SQL injection via web resources, using the analysis of the logs of the executed queries.

1 INTRODUCTION

The rapid evolution of information technologies has made today's society increasingly hostage to an internet connection (StudyCorgi, 2022). The number of resource-dependent quotidian services hosted on the web are increasing massively, something that has been accentuated more than ever with the appearance of COVID-19 (Subudhi and Palai, 2020), where many services that were previously manual processes have been transformed into digital processes (Portela et al., 2021), (Nachit and Belhcen, 2020). The growing of digital dependency makes an increasing number of companies to use web-hosted resources to support their business model (Dutta and Prasad, 2020). However, despite the advantages for organizations, it is necessary to take into account that, in the same way that digital dependence increases, the risk of cyberattack also increases (Ghosh, 2021), where often web resources are the most "desirable" targets for attackers for containing information of important value of the organizations. According to OWASP (Owasp, 2022), one of the most common attacks performed on web resources is SQL injection(Mukherjee et al., 2015; Bhateja et al.,). An SQL injection attack is the "injection" of a SQL query built from untrusted input data from an application's input components. When successful, an SQL injection exploit allows the attacker to: read and/or modify sensitive data stored in the database; perform administration operations on the database server (such as shutdown the DBMS). Thus, this paper suggests a solution for the detection of SQL injection attacks. The proposed solution, focused for now on the MySQL server, it detects attacks using the history of query executions on the database server, using REGEX, and some intelligent processes. The rest of the paper is organized as follows: Section 2 resumes the background analysis and related work for the detection/prevention of SQL injection attacks; in Section 3 is described the implementation of the proposed solution; Section 4 presents the preliminary results; Section 5 presents the conclusions and considerations for future work.

2 RELATED WORK

Many solutions for SQL injection detection have emerged in the literature of the area, proposing other approaches besides more conventional mechanisms such as Web Application Firewalls (WAFs). The use of machine learning techniques to support the de-

Rocha, A., Alves, R. and Pedrosa, T.
Query Log Analysis for SQL Injection Detection.
DOI: 10.5220/0011667200003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 471-476
ISBN: 978-989-758-624-8; ISSN: 2184-4356

tection process is common in some similar solutions identified. However, the mode of operation differs from the solution presented, because they act as an Intrusion Detection System (IDS) (Hasan et al., 2019) and use another type of logs (Azman et al., 2021) to perform the detection. All approaches of this type have a good success rate in the detection process.

Other solutions (Lee et al., 2012; Katole et al., 2018) follow an approach closer to that followed by the presented solution. Building a detection mechanism based on removing the content from the attributes values of SQL queries. After removing the values of the attributes, they perform the comparison of the new version of the query with the content of the query defined in a static way. Despite the similarities, the method of comparison of queries and removal of values seems to be less efficient than the method used in the presented approach.

Implementing a proxy between the web server and the application server was another of the techniques followed (Boyd and Keromytis, 2004). This solution presents a proxy, which when receiving SQL, translates, and validates it before forwarding it to the database. This is a simple syntactic validation, because the proxy is not aware of the semantics of the query itself.

Another major solution in the area is AMNESIA (Halfond and Orso, 2007). This is a fully automated, generic approach for detecting and preventing SQL injection attacks. Its two main points are: through an analysis of the code is built the model of legitimate queries that can be generated in the application; the second point deals with monitoring of dynamically generated queries at runtime and verification of compliance with the statically generated model.

3 PROPOSAL SOLUTION

The solution presented was designed to work as an IDS. The Figure 1 shows the overall architecture of the solution. As mentioned, in the current state of development, the solution was only tested on the MySQL server, however, on other database servers (ex: SQL Server (Gribkov, 2022)) it is also possible to configure a mechanism similar to General Query Log (Documentation, 2022) present in MySQL to store the history of executed queries.

The solution requires enabling the MySQL General Query Log. It is based on this mechanism that the detection of SQL injection attacks is performed, since it allows the storage of the executed queries. Thus, after activating this mechanism, in the same execution

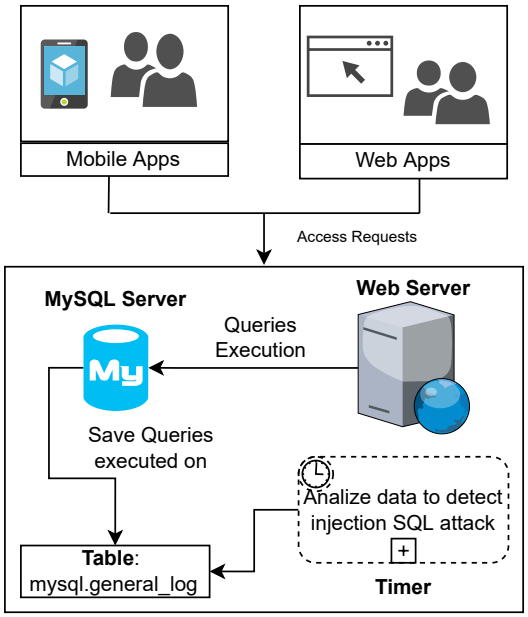

Figure 1: Overview of the proposed architecture.

environment of the database server, a service is configured to analyze the queries executed with a periodicity defined by the administrator, each query will be analyzed only once. For cases of non-detection of suspicious expressions of SQL injection, the query under analysis is marked as not malicious. When malicious expressions are detected in the queries, the service notifies, the administrator via email.

The existence of different techniques for SQL injection, makes one of the main difficulties in detecting this type of attacks is the identification of malicious expressions. In general, the main techniques can be classified into 2 groups (Rai et al., 2021): Inband SQL Injection, Inferential SQL Injection, each possessing several subtypes. Thus, for our first approach, the service created, performs the analysis of the stored queries trying to identify injection attacks with the Boolean based Blind SQL technique (subtype of the Inferential SQL Injection group). In Listing 1 it's possible to see the set of regular expressions used to identify some malicious expressions used in this technique.

Listing 1: Regular Expressions for Boolean based Blind SQL detection.

```
if re.findall(r'[0-9.-]+=[0-9.-]+',str(row)):
    #Detect number=number
elif re.findall(r'%+', str(row)):
    #Detect %%
elif re.findall(r'--+', str(row)): #--
    #Detect comments on mysql
elif re.findall(r"'([a-z]\w)'", str(row)):
    #Detect 'character' = 'character'
elif re.findall(r"'+'", str(row)):
    #Dectect field empty
elif re.findall(r'[||+]', str(row)):
    #Detect strings || +
```

```
elif re.findall(r"_IF(1=1,'true','false')",str(row)):
    #Detect if
```

Through the expressions illustrated in Listing 1, expressions such as *1=1* or *'a'='a'* that are often used, in the Boolean Based Blind SQL technique are quickly identified, allowing to alert the administrator of a possible injection attack in progress. However, as mentioned, there are several ways to carry out SQL injection attacks, in this way, to increase the success of detection, a second check, the queries, is performed in cases where the first scan does not detect any malicious expression.

As already mentioned, the first check was designed to identify the most used SQL injection techniques more quickly. The second verification is based on the concept of fuzzy string matching (Kuruvilla, 2022; Kalyanathaya et al., 2019; Kostanyan, 2017) for the detection of malicious expressions. This concept corresponds to the identification of two strings, string characters or entries that are approximately similar but are not exactly the same. It can be implemented using different techniques, in our solution, the Levenshtein Distance(Sharma, 2022; Yujian and Bo, 2007; Haldar and Mukhopadhyay, 2011) is used. This approach provides a measure of the number of single character insertions, exclusions or substitutions that vary between the sequences under analysis. In a real application context, the execution of a trusted query may vary. One of the main reasons for this variation is the content of the parameters inserted in the clause(s) *where* for filtering, which can be different from execution to execution. Thus, the solution presented, in the analysis of queries, removes, using regular expressions, the content of these parameters, reducing the occurrence of false positives. This removal allows, in the context of the proposed solution, that the Fuzzy string matching technique presents more reliable results.

Thus, the queries of the application are numbered (in the current state, this numbering is performed by the programmer). The Listing 2 shows an example of this numbering. In the definition of the query, the identifier that is intended to be assigned to the query is added as the SQL comment in the begin of the query.

Listing 2: Query Numeration Example.

```
#1
select * from table where field='field'
```

In addition to the numbering, an application query table is created, where all queries that should be monitored by the system are recorded. This table is a JSON file (see listing 3) where the query identifier and its contents are registered.

Listing 3: Queries table example.

```
[
    {"id":1,"content":select * from table where field"}
]
```

The main function of this table is to save an example of the execution (without the contents of the parameters of the clause(s) *where*) of the queries that will be monitored. This example does not contain any malicious expression and is used, as will be detailed, by the Fuzzy string matching technique in the analysis of the queries recorded in the logs.

The analysis process of the second mechanism begins with the search for the identifier that was assigned to the query under analysis. After this process, through regular expressions, is removed all the contents of the parameters of the clause(s) *where*. With the identifier found, it is obtained, through the query table, the secure content of the query under analysis. From this point, a comparison is performed, using python's fuzzywuzzy (Foundation, 2022) library, between the secure content and the contents of the preprocessed logs of the query in processing. The query execution is classified as malicious when the difference obtained is greater than 10%.

The definition of this value is justified because, in the testing phase, when querys are created dynamically (common in data filtering mechanisms), the *Inner joins* performed between the different tables can be different. Thus, the definition of Threshold (10%) was necessary to reduce the rate of false positives.

In cases where the difference is greater than 10%, as in the first check, the administrator is alerted. For both analysis processes existing in the proposed solution, in some contexts, namely, when more than one execution with suspected SQL injection is detected, in a short time (default 1m) the database server is blocked/shutdown preventively.

4 PRELIMINARY RESULTS

False positives/false negatives was one of the problems identified in the development phase. Thus, to try to reduce the occurrence of these events, a simpler test scenario was defined to perform the first validation of the solution. Using a web platform (iamvinitk, 2022), changed intentionally, to enable the exploitation of SQL injection attacks, along with sqlmap (G and Stampar, 2022; Bizimana and Belkhouja, 2017; Baklizi et al., 2022), which is a penetration testing tool that automates the process of detecting and exploiting SQL injection failures, it was defined that the information to be obtained in attacks on the test platform would be:

Table 1: Data obtained in different contexts tested.

	BD Name	Tables Name	User Data
Context 1	Yes	Yes	Yes
Context 2	Yes	Yes	Yes
Context 3	Yes	No	No

- name of the database.
- name of existing tables.
- data of registered users.

4.1 Experimental Conditions

Only one virtual machine was used, where the test platform was configured, already with the change made, and installed the respective database server (MySQL). To obtain the information described above, the execution of the attack, through sqlmap, was performed in 3 ways:

- without protection/detection mechanisms against SQLInjection;
- using WAF(SpiderLabs, 2022) with default settings;
- using the solution illustrated in this paper.

To avoid unequal environments with each repetition, sqlmap was executed on the same installation machine of the target platform, opting to ignore the sqlmap cache with each test. Each scenario (form of attack + information to obtain) was repeated 5 times with a break of 5s between each execution. The results obtained for the presented test scenarios will be detailed in section 4.2.

4.2 Results

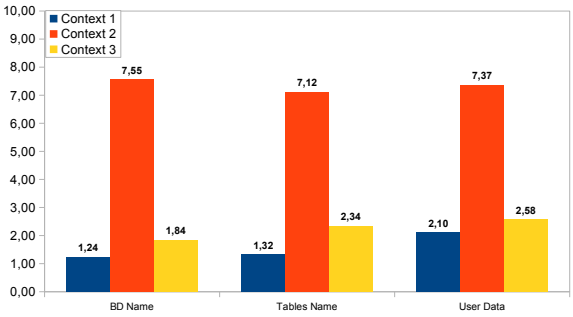

Figure 2: Time, in s, obtained in each context in the execution of the 3 defined steps.

In the table 1 and the figure 2 it's possible to analyze the results obtained in the tests performed. It turns out, unsurprisingly, that for context 1, the sqlmap can obtain information in the 3 stages tested with a relatively reduced time. For the case of context 2, the sqlmap allows the bypass to be performed to the WAF (Son, 2022), allowing the obtaining of data in the 3 stages tested, despite the elapsed execution time in the 3 stages (approximately 7s). Finally, the solution presented in this paper, which presents low execution times for the 3 steps, however, as the database server is blocked after the detection of querys with suspected SQL injection, only makes it possible to obtain the database name.

5 CONCLUSION

Despite the evolution of programming technologies and frameworks, it is still quite frequent the emergence of vulnerabilities that enable the exploitation of SQL injection in applications today, causing high-value resources to be compromised. The inexperience of programmers with new technologies (Dinerman, 2015; Kiskis, 2019; Mohd Yunus et al., 2018), limitations of more conventional protection mechanisms (ex: WAFs) (Vicente, 2019), too old code, are some reasons for this problem still exist.

The need for solutions that contribute to increased protection and monitoring against SQL injection attacks gives the solution presented in this paper some relevance. The simplicity of configuration, use of regular expressions and the logic of Fuzzy makes the success rate and efficiency in the process of detecting attacks is encouraging, as revealed in the first tests still performed in the development environment.

Another advantage of the solution is the independence over the technologies used in the web application that monitors, that is, as long as the database server used is MySQL, this solution can monitor any web application regardless of the technology used in its development.

However, despite the advantages described, the limitations of use to the MySQL server and the current mechanism of construction of the query table are points that must be worked out. Often programmers do not have any security training ending up not giving it the proper value (Roshaidie et al., 2020), this way, assigning the programmer the role of deciding which queries to put in the query table, may affect the success of the solution's attack detection. Finally, given the popularity (Chand, 2022) of other existing database servers in the market, the restriction of use to the MySQL server becomes an unfavorable point to the solution that should be corrected in future work. As it was mencioned previously, another problem that emerged in the early tests, still in the development phase, was false negatives and/ or false positives. Direct text matching is usually quite complex to perform

where it is often done using large ML algorithms. Another way to give more support to the work will be to expose the developed system to other test scenarios, which will certainly allow to refine the regular expressions and the Fuzzy logic used.

In addition, the alert and control mechanisms that the developed solution has, are contradictory with regard to the positioning of the solution in the IDS and/ or IPS concepts. The initial requirements, made this solution as IDS, which although more tests were needed to refine the detection, was demonstrated in section 4, which as an IDS produces minimally satisfactory results. However, the preventive blocking of the database server, in the case of the detection of a malicious query, brings the solution closer to the IPS concept. However, in the current state of development, the preventive blocking of the database server is an aggressive option for production environments, which, along with the poor maturity of the attack detection mechanism of the presented solution, makes it impossible to put the solution as a valid IPS option in SQL injection attacks.

5.1 Future Work

Despite the technical potential of the solution and the favorable results still obtained in development tests, there is still a long way to go so that the solution presented is possible to be used in production. In this way, the following points were left for future work:

- Testing more realistic scenarios to confirm detection success rates.
- Perform tests between the solution presented, the WAF and the solutions presented in section 2.
- Change the solution so that it is possible to use in other database servers (SQL Server, Oracle).
- Modify the current query table generation mechanism for a mechanism that does not depend on the programmer.
- Redesign the solution to work as an Intrusion Protection System (IPS).
- Replaced the blocking of the database server, for redirecting traffic to a honeypot.
- Explore the solution as a forensic analysis tool for SQL injection attacks.

ACKNOWLEDGEMENTS

This work was partially supported by the Norte Portugal Regional Operational Programme(NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, through the European Regional Development Fund (ERDF), within project “CybersSeCIP” (NORTE-01-0145-FEDER-000044). The authors are grateful to the Foundation for Science and Technology (FCT, Portugal) for financial support through national funds FCT/MCTES (PIDDAC) to CeDRI (UIDB/05757/2020 and UIDP/05757/2020) and SusTEC (LA/P/0007/2021).

REFERENCES

Azman, M. A., Marhusin, M. F., and Sulaiman, R. (2021). Machine learning-based technique to detect sql injection attack. *Journal of Computer Science*, 17(3):296–303.

Baklizi, M., Atoum, I., Abdullah, N., Al-Wesabi, O. A., Otoom, A. A., and Hasan, M. A.-S. (2022). A technical review of sql injection tools and methods: A case study of sqlmap. *International Journal of Intelligent Systems and Applications in Engineering*, 10:75–85.

Bhateja, N., Sikka, S., and Malhotra, A. chapter 34.

Bizimana, O. and Belkhouja, T. (2017). Sql injections and mitigations scanning and exploitation using sqlmap.

Boyd, S. W. and Keromytis, A. D. (2004). Sqlrand: Preventing sql injection attacks. In Jakobsson, M., Yung, M., and Zhou, J., editors, *Applied Cryptography and Network Security*, pages 292–302, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chand, M. (2022). Most popular databases in the world. https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/. Updated: Jan 17, 2022 Accessed: 2022-07-09.

Dinerman, K. (2015). Why sql injection vulnerabilities still exist: 8 reasons developer’s can’t eliminate them. https://www.rapid7.com/blog/post/2015/10/27/eight-reasons-why-sql-injection-vulnerabilities-still-exist-a-developer-s-perspective/. Updated: Thu, 31 Aug 2017, Accessed: 2022-07-09.

Documentation, M. (2022). The general query log. https://dev.mysql.com/doc/refman/8.0/en/query-log.html. Accessed: 2022-07-10.

Dutta, I. and Prasad, P. (2020). The global impact of online resources in business management. pages 805–807.

Foundation, P. S. (2022). fuzzywuzzy 0.18.0. https://pypi.org/project/fuzzywuzzy/. Accessed: 2022-07-14.

G, B. D. A. and Stampar, M. (2022). sqlmap - automatic sql injection and database takeover tool. https://sqlmap.org/.

Ghosh, A. (2021). An overview article on 600

Gribkov, E. (2022). How to check sql server query history. https://blog.devart.com/sql-server-query-history.html. Updated: September 25th, 2020 Accessed: 2022-07-10.

Haldar, R. and Mukhopadhyay, D. (2011). Levenshtein distance technique in dictionary lookup methods: An improved approach. *CoRR*, abs/1101.1232.

Halfond, W. G. J. and Orso, A. (2007). Detection and prevention of sql injection attacks. In Christodorescu, M., Jha, S., Maughan, D., Song, D., and Wang, C., editors, *Malware Detection*, pages 85–109, Boston, MA. Springer US.

Hasan, M., Balbahaith, Z., and Tarique, M. (2019). Detection of sql injection attacks: A machine learning approach. In *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*, pages 1–6.

iamvinitk (2022). Online-retail. https://github.com/iamvinitk/Online-Retail. Updated: 29 Jun, 2022 Accessed: 2022-07-09.

Kalyanathaya, K., D., A., and G., S. (2019). A fuzzy approach to approximate string matching for text retrieval in nlp. *Journal of Computational Information Systems*, 15:26–32.

Katole, R. A., Sherekar, S. S., and Thakare, V. M. (2018). Detection of sql injection attacks by removing the parameter values of sql query. In *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, pages 736–741.

Kiskis, A. (2019). Why sql injection attacks are still plaguing databases. *International Journal of Hyperconnectivity and the Internet of Things*, 3:11–18.

Kostanyan, A. (2017). Fuzzy string matching with finite automat. In *2017 Computer Science and Information Technologies (CSIT)*, pages 9–11.

Kuruvilla, V. P. (2022). A comprehensive guide to fuzzy matching/fuzzy logic. https://nanonets.com/blog/fuzzy-matching-fuzzy-logic/. Accessed: 2022-07-14.

Lee, I., Jeong, S., Yeo, S., and Moon, J. (2012). A novel method for sql injection attack detection based on removing sql query attribute values. *Mathematical and Computer Modelling*, 55(1):58–68. Advanced Theory and Practice for Cryptography and Future Security.

Mohd Yunus, M. A., Brohan, M., Mohd Nawi, N., Salwana, E., Najib, N., and Liang, C. (2018). Review of sql injection : Problems and prevention. *JOIV : International Journal on Informatics Visualization*, 2:215.

Mukherjee, S., Sen, P., Bora, S., and Pradhan, C. (2015). Sql injection: A sample review. In *2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7.

Nachit, H. and Belhcen, L. (2020). Digital transformation in times of covid-19 pandemic: The case of morocco. *SSRN Electronic Journal*.

Owasp (2022). Top 10 web application security risks. https://owasp.org/www-project-top-ten/. Accessed: 2022-07-09.

Portela, D., Brito, D. V., and Monteiro, H. (2021). Using digital technologies in response to the covid-19 pandemic in portugal. *Portuguese Journal of Public Health*, 39(3):170–174.

Rai, A. K., Miraz, M. M. I., Das, D., Kaur, H., and Swati (2021). Sql injection: Classification and prevention. *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*, pages 367–372.

Roshaidie, M., Liang, P., Jun, C., Yew, K., and tuz Zahra, F. (2020). Importance of secure software development processes and tools for developers.

Sharma, S. (2022). Fuzzywuzzy python library. https://www.geeksforgeeks.org/fuzzywuzzy-python-library/. Updated: 29 Jun, 2022 Accessed: 2022-07-09.

Son, D. (2022). Sqlmap tamper script for bypassing waf. https://securityonline.info/sqlmap-tamper-script-bypassing-waf/. Updated: 29 Jun, 2022 Accessed: 2022-07-09.

SpiderLabs (2022). Open source web application firewall. https://github.com/SpiderLabs/ModSecurity. Updated: 29 Jun, 2022 Accessed: 2022-07-09.

StudyCorgi (2022). Impact of the internet on society. https://studycorgi.com/impact-of-the-internet-on-society/. Updated: Sep 24th, 2021 Accessed: 2022-07-09.

Subudhi, R. and Palai, D. (2020). Impact of internet use during covid lockdown. *Journal of Humanities and Social Sciences Research*, 2.

Vicente, G. (2019). The top 5 reasons why waf users are dissatisfied. https://hdivsecurity.com/bornsecure/the-top-5-reasons-why-waf-users-are-dissatisfied/. Updated: May 21, 2019, Accessed: 2022-07-09.

Yujian, L. and Bo, L. (2007). A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095.