Evaluating the Effectiveness of Conventional Fixes for SQL Injection Vulnerability

Swathy Joseph and K.P. Jevitha

Abstract The computer world is definitely familiar with SQL as it plays a major role in the development of web applications. Almost all applications have data to be stored for future reference and most of them use RDBMS. Many applications choose its backend from the SQL variants. Large and important applications like the bank and credit-cards will have highly sensitive data in their databases. With the incredible advancement in technology, almost no data can survive the omniscient eyes of the attackers. The only thing that can be done is to make the attackers work difficult. The conventional fixes help in the prevention of attacks to an extent. However, there is a need for some authentic work about the effectiveness of these fixes. In this paper, we present a study of the popular SQL Injection Attack (SQLIA) techniques and the effectiveness of conventional fixes in reducing them. For addressing the SQLIA's in depth, a thorough background study was done and the mitigation techniques were evaluated using both automated and manual testing. We took the help of a renowned penetration testing tool, SQLMap, for the automated testing. The results indicate the importance of incorporating these mitigation techniques in the code apart from going for complex fixes that require both effort and time.

Keywords Web-attacks · SQLIA · SQL injection

S. Joseph (✉) · K.P. Jevitha
Department of Computer Science and Engineering,
Amrita Vishwa Vidyapeetham, Coimbatore, India
e-mail: swathyjoseph90@gmail.com

K.P. Jevitha
e-mail: kp_jevitha@cb.amrita.edu

A. Nagar et al. (eds.), *Proceedings of 3rd International Conference on Advanced Computing, Networking and Informatics*, Smart Innovation, Systems and Technologies 44, DOI 10.1007/978-81-322-2529-4_44

1 Introduction

SQL Injection is defined by OWASP as "an attack that consists of insertion or injection of a SQL query via input data from client to the application" [1]. SQLIA has probably existed for ages, dating back to when SQL databases were first linked to web applications. Despite its age, it still persists. SQLIA tops the recent list of OWASP's Top 10 web-attacks [2]. The evidences that these attacks still exist are the Lizamoon [3] and the Lilupophilupop [4] attacks that compromised almost a million websites.

The key objective of this work is to evaluate the effectiveness of the conventional techniques in reducing the SQLIA. It was seen that the use of legacy code in application development is one of the main reasons for this attack. The other reasons include lack of input sanitization, architectural issues etc. We hope that our work throws some light on the effectiveness of the code level defense techniques. The important SQLIA's considered here are the boolean-based blind, stacked queries, time-based blind attack and the error-based injection attacks [5]. The conventional fixes for SQLIA evaluated in this work include the use of parameterized queries, whitelist validation and stored procedures.

The rest of the paper is structured as follows: In Sect. 2, we present the related work in this domain. In Sect. 3, we give a briefing about the different SQLIA techniques and the intent for using them. In Sects. 4 and 5, we describe the use of SQLMap tool and the testbed setup. In Sect. 6, we discuss about the different defensive coding techniques and the implementation of code level mitigations. In Sect. 7, we present the evaluation results and we conclude in Sect. 8.

2 Related Works

Diallo et al. [6] has a survey on the various dimensions of SQLIAs that includes the different classes of attacks and the available approaches against them. Bono and Domangue [7] describe the application of attack ideologies and the mitigations. Input sanitization, prepared statements, stored procedures, principle of least privileges and security audits are listed out in this paper. Shar and Tan [8] conclude that the best strategy against the injection attacks is the integration of defensive coding with runtime prevention methods. This work also gives a summary of the various detection methods and run-time prevention methods. Ahmad et al. [9] has a detailed study of each attack for the purpose of categorization of SQLIAs. This paper categorizes the SQLIA into order-wise attacks, those against the database and finally as blind attacks. They conclude that this classification can help in reducing the possibility of occurrence of the vulnerability. Bisht et al. [10] throw some light onto the prevention of SQLIA. Jane and Chaudhari [11] propose an approach which is related to the inference of the intended query structure within an application. This paper states that, for an input to be a useful candidate, it should be benign and the

program's control must flow in the same path. The programmer intention is mined out and hence the approach is concluded to be quite promising. Halfond and Osro [12] propose a technique that uses finite state machines for the detection of SQLIAs and thereby preventing them. This technique finds the hotspots and creates automaton for every hotspot. When a query is issued, this is verified against its corresponding automata model and only if they match the user is allowed to continue.

3 Important SQLIA Techniques

In this section, we present and discuss the important kinds of SQLIAs known to date [13]. These techniques are used either individually or a few of them together to perform the attack. Out of the described techniques, time-based and boolean based attacks are inferential attacks.

Boolean-based blind technique: The attacker has no direct knowledge about the authentication he has to obtain. This can be done by attaching true or false statements along with legitimate SQL queries and observing the responses.

Time-based technique: The intent of the attacker is to verify the precision of his guesses based on the time injection queries used. SQL statements used in this type of injection attack holds the back-end database for a certain number of seconds. The attacker notes the response time to infer if the injection is successful or not.

Error-based technique: This kind of attack exploits the error messages returned by the database management software. If these are observed carefully, database fingerprinting might be possible. Once the attacker gets to know the details of the backend, he can use technology-specific methods for the attack. But this kind of attack works only for the applications that are configured to disclose back-end dbms error messages.

Piggy-backed technique: This technique is also known as the stacked queries method. This attack uses multiple queries separated by a semicolon which can be applied for data manipulations.

These are the first-order injection techniques. There are second-order SQLIAs that take in the attack-intended input normally and the effect of that input will be noticeable only from the next access to the database.

4 SQLMap

The SQLMap [5] tool, is an open source penetration testing tool developed in Python. It automates the detection and exploitation of SQL injection flaws and takes advantage of the vulnerabilities to gain access to the contents in the backend database system. It supports the exploitation of 5 different injection types and provides support for a number of databases. The tool uses the bisection algorithm

```
python sqlmap.py -u "http://localhost:8080/bookstore_current-0.1-dev/Login.jsp"
POST –data="FormName=Login&Login=admin&Password=admin&FormAction=login&
ret_page=&querystring=" –p "Login" –proxy "http://127.0.0.1:8081" –beep --risk=1
--level=2
```

Fig. 1 Sample SQLMap usage

for the implementation of boolean-based and time-based injections. Figure 1 provides a sample usage of the SQLMap. The tool was used to test all the applications by changing the level and risk options available within the tool. The risk argument specifies the risk of tests that were to be done like the default tests or heavy query tests. The level argument specifies the level of tests to be performed. The number of tests performed increases as the level value is increased. The http request is given as a parameter and p indicates the injectable parameter.

5 Testbed Description

This section gives a description of the testbed used for evaluating the different SQLIA prevention techniques. For the purpose of comparison, we have used four different test applications from the Amnesia [12] Testbed. The subjects are of varying sizes. A brief description of what the applications dealt with, is given below:

- Bookstore: online purchasing of books.
- Classifieds: advertise items and pets to be sold.
- Employee directory: details of the employees in a company.
- Events: details of various events to be held/hosted.

The applications were Java based with MySQL 5.6 as their back-end. These applications were deployed using tomcat on windows 7. The SQLMap gives the penetration tester, a broad range of options which eases his work. Initially, on the raw code, with no fixes implemented, the SQLMap tool was run. It was seen that 3 types of injections were possible for all the applications. The injections possible were error-based, time-based and boolean-based blind [5]. The manual testing on these applications also showed the same result. Table 1 gives the vulnerability of each application

Table 1 Number of injectable pages per application

Application	Number of web-pages per application	Number of pages injectable
Bookstore	19	2
Classifieds	20	3
Employee directory	14	2
Events	13	2

6 Mitigation Techniques

Three mitigation techniques were tried out—the parameterized queries, the stored procedures and the whitelist validation.

Parameterized queries: This is language dependent. Java's JDBC provides the use of prepared statement class, while PHP provides the PHP Data Object (PDO) package. This supports the placeholders and named parameters. Since our applications are Java-based we go for the usage of prepared statements. Parameterized queries provide the benefit that they take the user input as such.

```
E.g.: PreparedStatement prepStat = con.prepareStatement("select mem_id,
mem_lvl from membrs where mem_login =? and mem_passwd=?");
```

can be used instead of:

```
rs = openrs( stat, "select mem_id, mem_lvl from membrs where
mem_login = " + toSQL(sLogin, adText) + " and mem_passwd="
+ toSQL(sPassword, adText));
```

Whitelist validation: The Blacklist validation is popular. However, the whitelist validation is more effective than the other one. Here we accept only what is known to be true. We took the help of regular expressions for implementing this and used the java.util.regex package. Table 2 shows the expressions used for the whitelist validation. Data type, data size, range etc. are to be kept in mind while designing a regular expression. So the bottom line is that, the applications would accept only the input in the mentioned formats.

Stored procedures: This makes the task easier when repetitive tasks are to be used. The SQL code is predefined in the database and then accessed from the application. Stored procedure method is also said to give the effect of prepared statements mentioned above, if used safely.

```
E.g.: begin select mem_id, mem_level into res1, res2 from membrs
    where mem_login = plog and mem_passwd = ppas;
```

Table 2 Sample regular expressions used for whitelist validation

Regular expression	Purpose
”\ \w*@\ \w*[.]\ \w{3}”	For accepting an email_id
”\ \d{13}”	For accepting credit card number
”\ \d{7}”	Phone number
”\ \w*”	First and last name
”\ \w{3,15}”	Login and password

We used this as the SQL procedure for accepting credentials for a login page in the application which was stored in the database and later accessed from application using:

```
String simpleProc = "{ call pbookstre2 (?,?,?,?) }";
```

Hence the test applications were re-coded using these three conventional fixes separately.

7 Evaluation and Results

In this section, we describe how the effectiveness of the above mentioned techniques were evaluated. The injection points were found out and then the tool was run over the re-coded applications. The results were taken in the same manner varying the risk and level parameters. Similarly the results were taken for every injection point. Studying the queries that caused the injections even after the implementation of the mitigation techniques revealed that they were complex and nested queries. An example of a http request which caused error-based injections possible bypassing the whitelist validation technique is:

```
Payload: FormName=Login&Login=admin' AND (SELECT 6410 FROM
(SELECT COUNT(*),CONCAT(CHAR (58, 103, 112, 97, 58), (SELECT
(CASE WHEN (6410=6410) THEN 1 ELSE 0 AND =login&
ret_page=&querystring=
```

Table 3 shows a sample reading taken for the bookstore application when prepared statements completely secured the application from injection attacks. Table 4 shows a sample reading for classifieds application. It was seen that the injections were possible even after the use of prepared statements. Most of the queries found its way to the data stored in the database through the virtual database of MySQL called information_schema.

The injections possible in the testbed applications were found to be of three major types: boolean-based blind, time-based blind and error-based injections [5]. The frequency of the injection attacks are depicted in Fig. 2. The time-based boolean attack seems to be used heavily for the exploitation purpose, followed closely by the error-based injection techniques.

The pie-chart in Fig. 3 shows the analysis of the fixes after recoding the applications. Out of all the injections possible after the application of the fixes, the analysis showed that 64 % of the queries bypassed the prepared statement fix. The queries were mostly of second-order. 22 % of the queries bypassed stored procedures and 14 % bypassed the whitelist validation. Our analysis leads to a conclusion that the prepared statements prevent the first-order SQL injections to an extent but not the second order injections. There are a few loopholes mentioned in [14] with the use of stored procedures. A code which uses a procedure which has

Table 3 A sample reading with zero injections possible when prepared statement is used

Risk	Level	Number of injections before applying fix	After applying fix 1: prepared statements	After applying fix 2: stored procedures	After applying fix 3: whitelist validation
1	1	3	0	0	0
1	2	3	0	0	0
1	3	3	0	0	0
1	4	3	0	0	0
1	5	3	0	0	0
2	1	3	0	0	0
2	2	3	0	0	0
2	3	3	0	0	0
2	4	3	0	0	0
2	5	3	0	0	0
3	1	3	0	1	3
3	2	3	0	1	3
3	3	3	0	1	3
3	4	3	0	1	3
3	5	3	0	1	3

Table 4 A sample reading for classifieds application which shows injections are possible even when prepared statement is used

Risk	Level	Number of injections before applying fix	After applying fix 1: prepared statements	After applying fix 2: stored procedures	After applying fix 3: whitelist validation
1	1	3	2	0	0
1	2	3	2	0	2
1	3	3	2	0	2
1	4	3	2	0	2
1	5	3	2	0	2
2	1	3	2	0	2
2	2	3	2	0	2
2	3	3	2	0	2
2	4	3	2	0	2
2	5	3	2	0	2
3	1	3	2	1	2
3	2	3	2	1	2
3	3	3	2	1	2
3	4	3	2	1	2
3	5	3	2	1	2

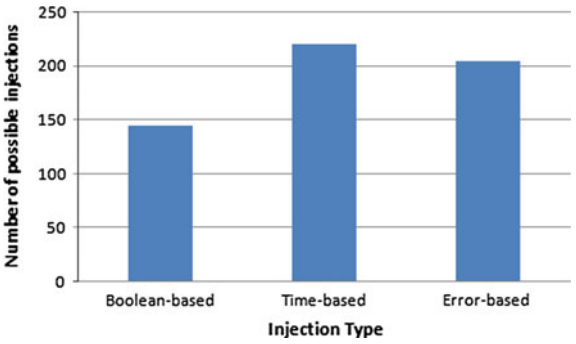


Fig. 2 Frequency graph of the injection attacks possible

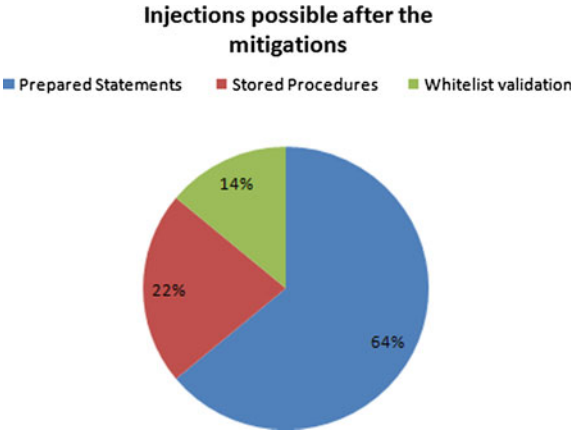
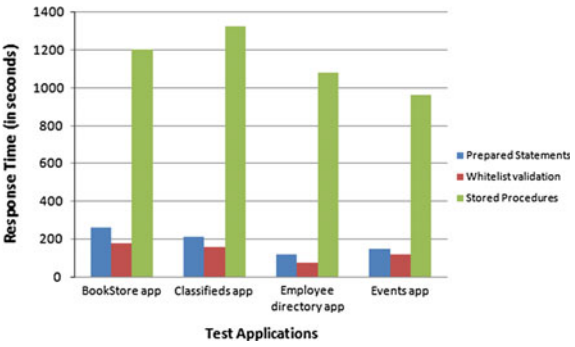


Fig. 3 Analysis of effectiveness of the fixes

exec(@input) stored in its backend database will execute the input given by the user regardless of what it is. While designing the stored procedures, the design of parameters with larger sizes will make the application prone to SQL injections. The designer should take ample care to allocate only limited sizes to the parameters in the stored procedures. Given that the developer pays attention to the above criteria, the stored procedures can effectively work against the first and second order injections. The regular expressions used proved to be efficient. However, the one used for accepting the user's name was seen to be injectable as we did not mention any particular constraints for it. Integrating validation with both the techniques mentioned above is almost unbreakable for the attackers. The use of regular expressions for whitelist validation is the most effective way to do it. There is nothing as good as a properly crafted regular expression for the validation. The entire data in the database could be dumped if injection was possible, through the injection points. Our analysis shows that the whitelist validation can control the first and second order injections to a good extent by efficiently utilizing the power of regular expressions.

Fig. 4 Response time



The Fig. 4 shows the response time of different applications to the SQLMap tool. The execution took approximately equal time for both prepared statements and whitelist validation. The time taken by the tool was very high for applications using stored procedures compared to the ones using the other two fixes. We believe that the difference in time for execution of applications that use stored procedure, is due to the fact that the actual functionality is stored in the database and hence access time is quite large.

8 Conclusion

The conventional mitigations techniques were evaluated based on parameters like the response time and decrease in the number of injections possible. The work converged to a conclusion that by the usage of the conventional mitigation techniques, we can reduce the number of injections considerably. Techniques which focus on prevention of the attack should incorporate defensive coding. Dynamically built SQL statements should be created properly with the conventional mitigation techniques mentioned rather than using the simple concatenation technique. This will make sure that only the legitimate queries are passed to the database server. Using validation along with the other two techniques proved to be a reasonable solution for this attack. The study of effectiveness of the conventional fixes showed their inability for ultimate eradication and this opens the scope for further research work.

References

1. OWASP: https://www.owasp.org/index.php/SQLInjection
2. OWASP Top 10 list: https://www.owasp.org/index.php/Top_10_2013-Top_10
3. LizaMoon the Latest SQL-Injection Attack: http://blogs.mcafee.com/mcafee-labs/lizamoon-the-latest-sql-injection-attack

4. Lilupophilupop: Tongue-twister SQL injection attacks pass one million mark: http://www.infosecurity-magazine.com/news/lilupophilupop-tongue-wister-sql-injection/
5. SQLMap: https://github.com/sqlmapproject/sqlmap/wiki
6. Kindy, D.A., Pathan, A.K.: A Detailed survey on various aspects of SQL injection in web applications: vulnerabilities, innovative attacks and remedies. In: International Journal of Communication Networks and Information Security, vol. 5, no. 2, pp. 80–92 August 2013
7. Bono, S.C., Domangue, E.: SQL Injection: A Case Study, Whitepaper Oct 2012
8. Shar, L.K., Beng, H., Tan, K.: Defeating SQL Injection. IEEE Comput. Soc. **46**(3), 69–77 (2013) (IEEE)
9. Ahmad, K., Shekhar, J., Yadav, K.P.: Classification of SQL injection attacks. In: VSRD-TNTJ, vol. I, no. (4), pp. 235–242(2010)
10. Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. In: ACM Transactions on Information and System Security, vol. 13, no. 2, p. 139. ACM (2010)
11. Jane, P.Y., Chaudhari, M.S.: SQLIA: Detection and prevention techniques: a survey. IOSR J. Comput. Eng. **2**, 56–60. IOSR J. (2013)
12. Halfond, W.G.J., Orso, A.: AMNESIA: analysis and monitoring for neutralizing SQL injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183. ACM, New York (2005)
13. Clarke, J.: SQL Injection Attacks and Defense. Elsevier Inc (2009)
14. Howard, M., LeBlanc, D.: Writing Secure Code, 2nd edn. Microsoft Press, Washington (2003)