Alinteri J. of Agr. Sci. (2024) 39(2): 160-165
e-ISSN: 2587-2249
info@alinteridergisi.com

http://dergipark.gov.tr/alinterizbd
http://www.alinteridergisi.com/

SEPTIC: Vulnerabilities in the DBMS and injection attacks are noted

1.S. Anitha,2.Paramhans Yadav,3.Kakunuri Sandhya,4.Pooduri Abhilash,5.Penumatasa Sai pramodvarma

1.Asst.Professor, Computer science and Engineering
CMR Engineering College, Medchal , T.S , India
2,B.Tech ,Computer Science and Engineering
CMR Engineering College, Medchal,T.S, India

Abstract:

Although databases are still the most widely used backend storage in businesses, injection attacks can be carried out against them since they are frequently connected with weak applications like web frontends.Because of a semantic discrepancy between how SQL queries are actually processed by databases and how they are commonly understood to be performed, these attacks are particularly powerful.This results in minute vulnerabilities in the way apps validate input. The method for preventing DBMS attacks that we present in this paper, called SEPTIC, can also help identify application vulnerabilities. The method was put into practice in MySQL and tested in experimental settings with different protection strategies.In contrast to other solutions, our data demonstrate that SEPTIC produces neither false positives nor false negatives.Additionally, they demonstrate that a minor performance overhead—roughly 2.2%—is introduced by SEPTIC.

I. INTRODUCTION

WEB applications have been around for more than two decades and are now an important component of the economy, as they often serve as an interface to various business related activities. Databases continue to be the most commonly used backend storage in enterprises, and they are often integrated with web applications. However, web applications can have vulnerabilities, allowing the data stored in the databases to be compromised. SQL injection attacks (SQLI), for example, continue to rise in number and severity [2], [14]. Commonly used defenses are validation functions, web application firewalls (WAFs), and prepared statements. The first two inspect web application inputs and sanitize those that are considered dangerous, whereas the third bounds inputs to placeholders in the SQL queries.1 Other anti-SQLI mechanisms have been developed but less adopted. Some of these monitor and block SQL queries that deviate from specific models, but the inspection is made without full knowledge about how they are processed by the DBMS [5], [6]. In all these cases, developers and system

administrators make assumptions about how the server-side scripting language and the DBMS work and interact, which sometimes are simplistic, whereas in others are blatantly wrong. For example, programmers usually assume that the PHP function mysql_real_escape_stringalways effectively sanitizes inputs and prevents SQLI attacks, which is not true. Also, they often assume that values retrieved from a database do not need to be validated before being inserted in a query, leading to second-order injection vulnerabilities. This is visible when, for instance, the code admin' - - is sanitized by escaping the prime character before sending it to the database, but the DBMS unsanitizes it before actually storing it. Later, the code is retrieved from the database and used un sanitized in some query, carrying out the attack. Such simplistic/wrong assumptions seem to be caused by a semantic mismatch between how an SQL query is expected to run and what actually occurs when it is executed (e.g., the programmer expects it to be sanitized but the DBMS unsanitized it). This mismatch may lead to vulnerabilities, as the protection mechanisms may be ineffective (e.g., they may miss some attacks). To avoid this problem, SQLI attacks could be handled inside, after the server-side code processes the inputs and the DBMS validates the queries, reducing the amount of assumptions that are made. The mismatch and this solution are not restricted to web applications, meaning that the same problem can be present in other business applications. In fact, injection attacks are a generic form of attack, transversal to all applications that use a database as backend. This idea of handling attacks inside has been quite successful in the realm of binary applications, to stop attacks irrespectively of the developers ability to followsecure programming practices or not. In that case, inside means that protection mechanisms are inserted in programming libraries or operating systems. Examples include address space layout randomization, data execution prevention, or canaries/stack cookies. In this paper, we propose a similar idea for applications backed by databases. We propose to block injection attacks inside the DBMS at runtime. We call this approach SElf- ProtecTIng databases from attaCks (SEPTIC). The DBMS is an interesting location to add protections against such attacks because it has an unambiguous knowledge about what will be considered as expressions clauses, predicates, and of an SQL statement. No mechanism that actuates outside of the DBMS has such knowledge. We address two categories of database attacks: SQL injection attacks, which continue to be among those with highest risk and for which new variants continue to appear and stored injection attacks, including stored cross-site scripting, which also involve SQL queries. For SQLI, we propose to catch the attacks by comparing queries with query models, improving an idea that has been previously used only outside of the DBMS [5], [6] and by comparing queries with validated queries with a similarity method, accuracy. improving detection For stored injection, we employ plugins to deal with specific attacks before data are inserted in the database. SEPTIC relies on two newconcepts. Before detecting attacks, the mechanism can be trained by forcing calls to all queries in an application. The result is a set of query models. However, as training may be incomplete and not cover all queries, we introduce the notion of putting in quarantine queries at runtime for which SEPTIC has no query model. The second concept, aging, deals with updates to query models after a new release of an application, something that is inevitable in real world software. Both concepts allow a reduction of the false negative (attacks not detected) and false positive (alerts for nonattacks) rates.

Architecture and data flows of SEPTIC.

II. EXISTING SYSTEM

AMNESIA [17] and CANDID [3] are two of the first works about detecting SQLI by comparing the structure of an SQL query before and after the inclusion of inputs and before the DBMS processes the queries. Both use query models to represent the queries and do detection. AMNESIA creates models by analyzing the source code of the application and extracting the query structure. Then, AMNESIA instruments the source code with calls to a wrapper that compares queries with models and blocks attacks. CANDID also analyzes the source code of the application to find database queries, then simulates their execution with benign strings to create the models. On the contrary, SEPTIC does not involve source code analysis or instrumentation. With SEPTIC, we aim to make the DBMS protect itself, so both model creation and attack detection are performed inside the DBMS. Moreover, SEPTIC aims to handle the semantic mismatch problem, so it analyzes queries just before they are executed, whereas AMNESIA and CANDID do it much earlier. These two tools also cannot detect attacks that do not change the structure of the query (syntax mimicry).

ϖ Buehrer et al. [6] present a similar scheme that manages to detect mimicry attacks by enriching the models (parse trees) with comment tokens. However, their scheme cannot deal with most attacks related with the semantic mismatch problem. SqlCheck [43] is another scheme that compares parse trees to detect attacks. SqlCheck detects some of the attacks related with semantic mismatch, but not those involving encoding and evasion. Again, both these mechanisms involve modifying the application code, unlike SEPTIC.

ϖDIGLOSSIA [42] is a technique to detect SQLI attacks that was implemented as an extension of the PHP interpreter. The technique first obtains the query models by mapping all query statements' characters characters to shadow except user inputs,and computes shadow values for all string user inputs. Second, for a query execution, it computes the query and verifies if the root nodes from the two parsed trees are equal. Like SEPTIC, DIGLOSSIA detects syntax structure and mimicry attacks but, unlike SEPTIC, it neither detects second-order SQLI once it only computes queries with user inputs, nor encoding and evasion space characters attacks as these attacks do not alter the parse tree root nodes before the malicious user inputs are processed by the DBMS. Although better than AMNESIA and CANDID, it does not deal with all semantic mismatch problems.

Disadvantages

o There is no SEPTIC which did not report false positives and did not miss detections (false negatives).

o There is no Process for complex and dynamic queries.

III. PROPOSED SYSTEM

A query is represented by a list of stacks in the proposed system's database management system (DBMS). While this is the most popular approach, other DBMSs might use other data structures. In this scenario, the tests for attack detection would need to be modified to take advantage of the information that is now accessible, or it would be feasible to translate across data structures.

The administrator must still exert some manual labor with SEPTIC in order to evaluate the QM in the quarantine data store or to start the training. Although a lot of work was spent into removing these kinds of operations off the crucial path of deploying an application into the field, it would have been ideal to have a fully automated approach.

Queries can be processed if they match a QM from an earlier version of the program thanks to the aging process.It's possible,

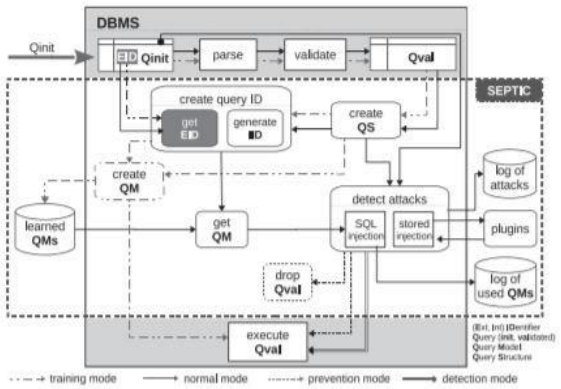
though, that these models are no longer appropriate because they allow attacks to suit these QMs. Using a more aggressive senescence period is one way to get around this restriction, but there are trade-offs that must be well understood.

SQL injection is the main focus of the current evaluation. To fully detect stored injection attacks, like as XSS, more work would be required.

Advantages

- A log with all analyzed queries was checked to determine if there were malicious queries that had remained unblocked.
- The log of attacks was verified to find out if SEPTIC had erroneously flagged a benign query as malicious (false positives).

IV. SYSTEM ARCHITECTURE



V.IMPLEMENTATION

- **DataOwner**

The data owner uploads their data to the cloud server using this module. The data owner encrypts the file and the index name before storing them in the cloud for security reasons.A particular file may be able to be deleted by the data encryptor.Additionally, he will be able to see the transactions based on the files he uploaded to the cloud and perform the subsequent tasks: Owners of Register and Login Data, Create digital signatures based on desc and add data material about the military, courts, government, and sports, such as ccat, cname, cpublication, and ccreato. Browse, enc data desc, upload, and include a picture. View every piece of data together with ratings and ranks using a digital signature.View every piece of uploaded data and the ranking with no digital signature. View the file download request and provide consent.SQL Procedures --- Separate the entire page into two sections: one for inputting DBMS queries and the other for results display. SQL Injection occurs when a query is not complete (insert,update,select,delete).

- **DataUser**

The user enters his or her password and user name to log in to this module.User actions after logging in include seeing your profile,Ask the application server for the secret key, then see the response. Use a keyword to search data, read full details, and, with permission, immediately obtain the secret key. Verify the signature before downloading the file. Should the signature be incorrect, do not download

- **ApplicationServer**

In order to provide data storage services, the application server oversees a cloud and performs certain tasks like viewing all data owners and authorizing View and approve each end user. View all content with rankings and ratings that has a digital signature, or view all content with rankings and ratings that doesn't have one. observe user search activity, View every SQL Injection Intruder along with their IP address, date, and time.

View the chart of all documents ranked. View every intrusive party and provide a link to the chart (name of the number of attacked documents).

- **SignatureGenerator**

The person who creates the digital signature is known as the signature generator. They can also perform the following tasks: login, view all owner papers, provide the choice to create a digital signature, view all data contents with ranks, and provide the option to create a secret key using RSA.

VI.CONCLUSIONANDFUTURE WORK

This study investigated a novel defense against online and commercial application database threats. It proposed the notion of intercepting attacks within the database management system, so shielding it against stored injection and SQL injection attacks.Furthermore, we demonstrated that sophisticated assaults, such as those connected to the semantic mismatch issue, can be identified and stopped by implementing protection within the database management system.Secondly, it offered a method for determining application code vulnerabilities in the event that assaults were discovered. Additionally, SEPTIC—a method built within MySQL—was presented in this study.SEPTIC uses a learning phase, as well as quarantine and aging procedures that deal with query models—creating and managing them—to perform detection. The method was tested with open-source PHP web apps and other kinds of programs, as well as with artificially created code that had vulnerabilities added. According to this evaluation, the mechanism outperformed all other tools in the literature and the WAF that is most frequently used in practice in detecting and blocking the attacks it was designed to handle. It was also able to identify application code vulnerabilities when an attack tries to exploit them. An impact of about 2.2% is shown by the performance overhead evaluation of SEPTIC inside MySQL, indicating the applicability of our method in real-world systems.

REFERENCES:

[1] B. Ahuja, A. Jana, A. Swarnkar, and R. Halder, “On preventing SQL injection attacks,” *Adv. Comput. Syst. Secur.*, vol. 395, pp. 49–64, 2015.

[2] AkamaiTechnologie,Cambridge,MA,US A, “Q1 2016 state of the Internet/ security report,” Tech. Rep., vol. 3, no. 1, Jun. 2016.

[3] S.Bandhakavi,P.Bisht,P.Madhusudan,and V. N. Venkatakrishnan, “CANDID: Preventing SQL injection attacks using dynamic candidate evaluations,” in *Proc. 14th ACM Conf. Comput. Commun. Secur.*,” Oct. 2007, pp. 12–24.

[4] C. A. Bell, *Expert MySQL*. New York, NY, USA: Apress, 2007.

[5] S.W.BoydandA.D.Keromytis, “SQLrand: Preventing SQL injection attacks,” in *Proc. 2nd Appl.CryptographyNetw.Secur.Conf.*,2004, pp. 292–302.

[6] G.T. Buehrer, B. W.Weide, and P. Sivilotti, “Using parse tree validation to prevent SQL injection attacks,” in *Proc. 5th Int. Workshop Softw. Eng. Middleware*, Sep. 2005, pp. 106– 113.

[7] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy, “BenchLab: An open testbed for realistic benchmarking of web applications,” in Proc. 2nd USENIX Conf. Web Appl. Develop., 2011, pp. 37–48.

[8] M. Ceccato, C. D. Nguyen, D. Appelt, and L.C. Briand, “SOFIA: An automated security oracle for black-box testing of SQL-injection vulnerabilities,” in Proc. 31st IEEE/ACMInt. Conf. Autom. Softw. Eng., Sep. 2016,pp. 167–177.

[9] J. Clarke, SQL Injection Attacks and Defense. Rockland, MA, USA: Syngress, 2009.

[10] Common Vulnerabilitiesand Exposures,2014.[Online].Available:http://cve.mitre.org

[11] SolidIT: DB-Engines Ranking, Aug. 2015. [Online].Available:http://dbengines.com/en/ranking

[12] A. Douglen, "SQL smuggling or, the attack that wasn't there," COMSEC Consulting, Inf. Secur., London, U.K., Tech. Rep., 2007.

[13] M. Dowd, J. Mcdonald, and J. Schuh, Art of Software Security Assessment. London, U.K.: Pearson Edu., 2006.

[14] J. Fonseca, N. Seixas, M.Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," Trans. Dependable Secure Comput., vol. 11, no. 2, pp. 89–100, Mar./Apr.2014.

[15] Gambas, 2015. [Online]. Available: http://gambas.sourceforge.net/

[16] G. Modelo-Howard, C. Gutierrezand, F. Arshad, S. Bagchi, and Y. Qi., "pSigene: Webcrawling to generalize SQL injection signatures," in Proc. 44th IEEE/IFIP Int. Conf. Dependable Syst. Netw., Jun. 2014, pp. 45–56.

[17] W. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL- injection attacks," in Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng., Nov. 2005. pp. 174– 183.

[18] M. Howard and D. LeBlanc, Writing Secure Code for Windows Vista, 1st ed., Microsoft Press Redmond, WA, USA, 2007.