SYSTEM TOOL FOR PREVENTION OF INJECTION ATTACKS INSIDE THE DATABASE

[1]S. Md. Riyaz Naik, [2] M. Naga Deeskhitha, [3] G. Mounika, [4] T. N.S.Pranvi, [5] P. Prameela
[1] Assistant Professor, Dept of CSE, SANTHIRAM ENGINEERING COLLEGE, Nandyal, India.
[2,3,4,5] IV-B.Tech CSE, SANTHIRAM ENGINEERING COLLEGE, Nandyal, India.

ABSTRACT: Now –a- days almost everything gets digitalized, these makes the sensitive information to be in the hands of malicious attackers which leads to inappropriate results and data violation. Normal user access the information using front-end interfaces(web pages) These information gets stored at backend storage enterprises called databases. But some attackers perform attacks like SQL injections on databases, which corrupts the information stored in the databases. Our system proposes a method which avoids SQL injection by preventing by integrating with the database and avoid queries to be executed without proper validation.

I. Introduction:

As per the Open Web Application Security Project (OWASP), the Structured Query Language Injection Attack (SQLIA) is regarded as number one web application vulnerability in the past years[1]. In this modern computer era database has become very essential in any web applications. All the web applications that are being developed has database connectivity in some form, hence making it database dependent.Databases continue to be the most commonly used backend storage in enterprises, and they are often integrated with web applications. 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. 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 database. 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. This mismatch may lead to vulnerabilities, as the protection mechanisms may be ineffective. 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 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. We propose to block injection attacks inside the DBMS at runtime. We call this approach SElf- ProtecTIng databases from attaCks. No mechanism that actuates outside of the DBMS has such knowledge.

II. Literature Survey:

2.1.Preventing SQL Injections in Online Applications:

SQL Injection Attacks are a relatively recent threat to the confidentiality, integrity and availability of online applications and their technical infrastructure, accounting for nearly a fourth of web vulnerabilities [2].We present our study on the prevention of SQL Injections: overview of proposed approaches and existing solutions, and recommendations on preventive coding techniques for Java-powered web applications and other environments. Online data theft has recently become a very serious issue, and recent cases have been widely publicized over concerns for the confidentiality of personally identifiable information (PII). Application-level vulnerabilities, which are believed to account for 70% to 90% of overall flaws, are now the main focus of attackers

and researchers. Online applications (websites and services) are especially at risk due to their universal exposure and their extensive use of the firewall-friendly HTTP protocol [3]. Moreover, database security is too often overlooked in favour of web and application server security, resulting in backend databases being a major target for attackers which are able to use them as easy entry points to organizations' networks. Protecting online applications (e.g. websites) and web services against SQL Injection Attacks has thus become a major concern for organizations, which face threats that can go far beyond the expected reach of the public web or application server. While several effective prevention methods have been developed, ensuring full protection against SQL Injections remains an issue on a practical level. This paper also presents our solution, the SQLDOM4J [4], which targets Java environments. Freely based on McClure's SQL DOM, it enables developers to construct and execute safe SQL statements easily. The main concepts behind our SQL Injection prevention strategy are strong typing and the separation of control and data channels within SQL statements. These are implemented by leveraging both the strongly-typed nature of OO applications and JDBC's pre-compiled type-binded statement interface. In this will show that the use of the SQLDOM4J API to build and execute database queries effectively protects applications against SQL Injection Attacks.

2.2.CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations:

SQL injection attacks are one of the topmost threats for applications written for the Web. These attacks are launched through specially crafted user input on web applications that use low level string operations to construct SQL queries [5]. In this work, we exhibit a novel and powerful scheme for automatically transforming web applications to render them safe against all SQL injection attacks. Our technique for detecting SQL injection is to dynamically mine the programmer-intended query structure on any input, and to detect attacks by comparing them against the intended query structure. We propose a simple and novel mechanism, called Candid, for mining programmer intended queries by dynamically evaluating runs over benign candidate inputs [6]. This mechanism is theoretically well founded and is based on inferring intended queries by considering the symbolic query computed on a program run. Our approach has been implemented in a tool called CANDID that retrofits Web applications written in Java to defend them against SQL injection attacks. We report extensive experimental results that show that our approach performs remarkably well in practice. The widespread deployment of firewalls and other perimeter defenses for protecting information in enterprise information systems in the last few years has raised the bar for remote attacks on networked enterprise applications. However, such protection measures have been penetrated and defeated quite easily with simple script injection attacks, of which the SQL Command Injection Attack (SQLCIA) is a particularly virulent kind. An online application that uses a back end SQL database server, accepts user input, and dynamically forms queries using the input, is an attractive target for an SQLCIA. In such a vulnerable application, an SQLCIA uses malformed user input that alters the SQL query issued in order to gain unauthorized access to the database, and extract or modify sensitive information. SQL injection attacks are extremely prevalent, and ranked as the second most common form of attack on web applications in 2006 in CVE (Common Vulnerabilities and Exposures list) [7]. The percentage of these attacks among the overall number of attacks reported rose from 5.5% in 2004 to 14% in 2006. The recent SQLCIA on CardSystems Solutions that exposed several hundreds of thousands of credit card numbers is an example of how such attack can victimize an organization and members of the general public. By using Google code search, analysts have found several application programs whose sources exhibit these vulnerabilities. Recent reports suggest that a large number of applications on the web are indeed vulnerable to SQL injection attacks, that the number of attacks are on the increase, and is on the list of most prevalent forms of attack.

2.3.SQLrand: Preventing SQL Injection Attacks:

Now we are presenting a practical protection mechanism against SQL injection attacks. Such attacks target databases that are accessible through a web frontend, and take advantage of flaws in the

input validation logic of Web components such as CGI scripts. We apply the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to the attacker [8]. Queries injected by the attacker will be caught and terminated by the database parser. We show how to use this technique with the MySQL database using an intermediary proxy that translates the random SQL to its standard language. The prevalence of buffer overflow attacks as an intrusion mechanism has resulted in considerable research focused on the problem of preventing and detecting, or containing such attacks. Considerably less attention has been paid to a related problem, SQL injection attacks. Such attacks have been used to extract customer and order information from e-commerce databases, or bypass security mechanisms [9]. The intuition behind such attacks is that pre-defined logical expressions within a pre-defined query can be altered simply by injecting operations that always result in true or false statements. This injection typically occurs through a web form and associated CGI script that does not perform appropriate input validation. These types of injections are not limited strictly to character fields. Similar alterations to the "where" and "having" SQL clauses have been exposed, when the application does not restrict numeric data for numeric fields. Standard SQL error messages returned by a database can also assist the attacker. In situations where the attacker has no knowledge of the underlying SQL query or the contributing tables, forcing an exception may reveal more details about the table or its field names and types. This technique has been shown to be quite effective in practice.

III. PROPOSED WORK:

Author proposes a similar idea for applications backed by databases. Here we are going to block injection attacks inside the DBMS/database at runtime. We call this approach SElf- ProtecTIng databases from attaCks because it is used to stop the attacks by itself. The DBMS is an interesting location to add protections against such attacks because it has an unambiguous knowledge about what will be considered as clauses, predicates, and expressions 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[10] and for which new variants continue to appear [11] and stored injection attacks, including stored cross-site scripting, which also involve SQL queries. We showed that it is possible to detect and block sophisticated attacks. Assist on the identification of the vulnerabilities in the applications.

3.1.Modules Description:

3.1.1. DBMS Injection Attacks:-

We denominate semantic mismatch as an incorrect perception about how the SQL queries are executed by the DBMS—the developer expects queries to be processed in a certain way but they are actually run in a different manner. This mismatch often leads to mistakes in the implementation of protections in the source code of applications, making these vulnerable to SQL injection and other attacks involving the DBMS.

SQL injection attacks are also carried out by modifying the user input query as follows[12]:

A. Tautologies: This attack aims at injecting code in conditional statements so that they always return true. The main purpose is to bypass authentication pages and obtain data from them. The attack is termed efficacious when the code either displays all of the records or carries out some operation on the data.

B. Union queries: The attacker carries a union operation between the legitimate query and his own malicious query. This returns a data-set which is the union of the results of the original first query and the injected second query.

C. Stored Procedures: The aim of these attacks is to execute stored procedures present in the database. Stored procedures are a set of user defined functions that can be looked up and reused anytime. SQL attacks can be formulated to execute stored procedures provided by a specific database. Sometimes it might even include procedures that interact with the operating system. Furthermore, since stored procedures are often written in special scripting languages, they may contain other types of vulnerabilities, like buffer overflows, which will allow attackers to run arbitrary code on the server or escalate their privileges [13].

3.1.2. System Approach And Architecture:-

This system is implemented by a module inside the DBMS, allowing every query to be checked for attacks. The semantic mismatch problem is resolved because queries are evaluated for detection purposes near the end of the DBMS dataflow, just before the query is executed. As SEPTIC is inside the DBMS, it is independent from the application (e.g., from the application programming language) and the way it builds queries (e.g., dynamically). This lets SEPTIC analyze queries issued by any kind of application. However, with support from the application, SEPTIC can also contribute to the identification of the vulnerabilities.

3.1.3. Query Representations And Identifiers:

In this we first processes queries validated by the DBMS and represents them by QSs and QMs, depending if it executes in normal operation (prevention or detection mode) or in training mode. Also, each query model is known by a query identifier (ID). Therefore, the core of SEPTIC relies on queries, their representations, and identifiers. This section presents detailed information about them.

IV. RESULT AND ANALYSIS:

Fig a:All the Users

In the above figure we will see no. of users and their details(info) who are connected to that particular sector.

Fig b: Details of the User

In the above figure we have the user details in their profile and there we can able to check the account information and able to see the alerts.

Fig c: Alerts for User

In the above figure user can able to see the alerts and change their credentials.

V. CONCLUSION:

We demonstrate the problem of privacy preserving dataset integration. This is a new form of protection from attacks against web and business application databases. We believe that our approach offers all desirable properties, including a nice balance between privacy and utility, and with high practicality. We showed that it is possible to detect and block sophisticated attacks, including those related with the semantic mismatch problem.

VI. FUTURE ENHANCEMENT:

This system will explores a new point in the design space by identifying the attacks inside the DBMS, which has the benefit of precluding the semantic mismatch problem. In this case, either it is possible to perform a translation between data structures or the tests for attack detection would have to be

adapted to leverage from the available information. The aging process allows queries to proceed if they correspond to a QM of a previous version of the application. However, it is possible that these models are no longer acceptable, as they may let attacks fit these QMs.

REFERENCES:

[1]. J. V. William G.J. Halfond and A. Orso, -A classification of sql injection attacks and countermeasures,‖2006.

[2].Kar, Debabrata, and Suvasini Panigrahi. "Prevention of SQL Injection attack using query transformation and hashing." 2013 3rd IEEE International Advance Computing Conference (IACC). IEEE, 2013.

[3]Janot, Etienne, and Pavol Zavarsky. "Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM." OWASP App. Sec. Conference. 2008.

[4]Sadeghian, Amirmohammad, Mazdak Zamani, and Azizah Abd Manaf. "A taxonomy of SQL injection detection and prevention techniques." 2013 International Conference on Informatics and Creative Multimedia. IEEE, 2013.

[5] Bisht, Prithvi, Parthasarathy Madhusudan, and V. N. Venkatakrishnan. "CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks." ACM Transactions on Information and System Security (TISSEC) 13.2 (2010): 1-39.

[6] Tajpour, Atefeh, and Mohammad JorJor zade Shooshtari. "Evaluation of SQL injection detection and prevention techniques." 2010 2nd International Conference on Computational Intelligence, Communication Systems and Networks. IEEE, 2010.

[7] Johari, Rahul, and Pankaj Sharma. "A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection." 2012 International Conference on Communication Systems and Network Technologies. IEEE, 2012.

[8]Boyd, Stephen W., and Angelos D. Keromytis. "SQLrand: Preventing SQL injection attacks." International Conference on Applied Cryptography and Network Security. Springer, Berlin, Heidelberg, 2004.

[9] Liu, Anyi, et al. "SQLProb: a proxy-based architecture towards preventing SQL injection attacks." Proceedings of the 2009 ACM symposium on Applied Computing. 2009.

[10] T. Gigler, B. Glas, N. Smithline, and A. van der Stock, “OWASP Top 10: The ten most critical web application security risks – RC2,” OWASP Foundation, Tech. Rep.,2017.

[11] D.RayandJ.Ligatti,“Defininginjectionattacks,”inProc.Int.Conf.Inf. Secur., 2014, pp.425–441.

[12] Inyong Lee, Soonki Jeong, Sangsoo Yeo, Jongsub Moon, “A novel method for SQL injection attack detection based on removing SQL query attribute values”, Mathematical and Computer Modelling

[13] Indrani Balasundaram, Dr. E. Ramaraj, “An approach to prevent and detect SQL attacks in database using web service”, International Journal of Computer Science and Network Security

[14] W. Halfond, A. Orso, and P. Manolios, “WASP: Protecting web applica- tions using positive tainting and syntax-aware evaluation,” IEEE Trans. Software Eng., vol. 34, no. 1, pp. 65–81,2008.

[15] W. Xu, S. Bhatkar, and R. Sekar, “Practical dynamic taint analysis for countering input validation attacks on web applications,” Dept. Comput. Sci., Stony Brook Univ., Stony Brook, NY, USA, Tech. Rep. SECLAB- 05-04,2005.