International Journal of Information Technology and Knowledge Management
January-June 2011, Volume 4, No. 1, pp. 323-328

X – LOG AUTHENTICATION TECHNIQUE TO PREVENT SQL INJECTION ATTACKS

B. Indrani[1] & E. Ramaraj[2]

Recent laws governing data privacy have led to great interest in enabling secure database services. Many software systems include a Web-based component that makes Data available to the public via the Internet. These data stores are exposed to a variety of Web-based attacks. The fear of SQL injection attacks has become increasingly frequent and serious.. This paper presents a new Technique to protect Web applications against SQL injection. SQL-Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known fool-proof defense against such attacks. In this paper, we propose "X- Log Authentication Technique" to prevent SQL Injection Attacks the deployment of this technique is by appending the vulnerability Guard and X-Log Authentication as well as Stored Procedure of application scripts additionally allowing seamless integration with currently-deployed systems.

General Terms: Security, Validation, Verification.

Keywords: Database Security, World-Wide Application Security, SQL Injection Attacks, Runtime Monitoring

1. Introduction

The World Wide Web has experienced remarkable growth in recent years. Many enterprise applications deal with sensitive data. It is crucial to protect these applications from targeted attacks. Compromise of these applications represents a serious threat to organizations that have deployed them, and also to users who trust these systems to store confidential data. SQLIA's constitute an important class of attacks against web applications. Web applications that are vulnerable to SQL attacks user inputs the attacker's embeds commands and gets executed [4]. The attackers directly access the database underlying an application and leak or alter confidential information and execute malicious code [1][2]. The resulting security violations can include identity theft, loss of confidential information, and ultimately fraud. In some cases, attackers even use an SQL Injection vulnerability to take control and corrupt the system that hosts the Web application. The increasing number of web applications falling prey to these attacks is alarmingly high [3] Prevention of SQLIA's is a major challenge. It is difficult to implement and enforce a rigorous defensive coding discipline. Many solutions based on defensive coding address only a subset of the possible attacks. The Software poses a particularly difficult problem because of the cost and complexity of reworking on the existing code. Many techniques rely on complex static analysis in order to find potential vulnerabilities in the code. The evaluation of "X – Log Authentication Technique has no code modification as well as automation of detection and prevention. A recent penetration testing study of more than 250 Web applications concluded that at least 92% of Web applications are vulnerable to some form of malicious intrusions [7] In fact; SQLIA's have been included in list of top 10 threats to web applications [6]. A recent penetration testing study performed by the Imperva Application Defence Centre included more than 250 Web applications from e-commerce, online banking, enterprise collaboration and supply chain management sites and their vulnerability assessment concluded that at least 92% of Web applications are vulnerable to some form of malicious Intrusions [7]. Recent U.S. industry regulations such as the Sarbanes-Oxley Act pertaining to information security, try to enforce strict security compliance by application vendors.

1.1. SQL Injection Attacks

Over the past several years, attackers have developed a wide array of sophisticated attack techniques that can be used to exploit SQL injection vulnerabilities. In a SQL injection attack, an attacker attempts to exploit vulnerabilities in custom Web applications by entering SQL code in an entry field such as a login. If successful, such an attack can give the attacker access to data on the database used by the application and the ability to run malicious code on the Web site. Attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings [5].

[1]Department of Computer Science, Madurai Kamaraj University, Madurai

[2]Director of Computer Centre, Alagappa University, Karaikudi

Email: [2]indra.msc@gmail.com

1.2. Example of SQL Injection Attack

Tautology: A website uses this source(figure: 1), which would be vulnerable to a SQLIA. For example, if a user enters"' OR 1=1—" and"", instead of Login Id ="doe" and Password = "xyz", the resulting query is: SELECT user_ info FROM users WHERE Login Id='' OR 1=1 —' AND Password ='.

The database interprets everything after the WHERE token as a conditional statement, and the inclusion of the "OR 1=1" clause turns this conditional into a tautology. (The characters "—" mark the beginning of a comment, so everything after them is ignored.) As a result, the database returns the records for all users in the database. An attacker could insert a wide range of SQL commands via this exploit, including commands to modify or destroy database tables. This particular exploit is a simple example of SQLIA.

1.3. Piggybacked Queries

SELECT acct FROM users WHERE Login Id ='doe' AND Password = ' '; Drop table users —; Piggybacked Queries appends additional queries to the original query string the first query is generally the original legitimate query, whereas subsequent queries are the injected malicious queries. The database treats this query string as two queries separated by the query delimiter (";") and executes both. The second malicious query causes the database to drop the user_ info table in the database, which would have the catastrophic consequence of deleting all user information

```
Protected sub login_btn( )
cn.Open()
cmd = New SqlCommand("select * from user_info where
LoginID='" & t1.Text &"' and Password=" & t2.Text &
"", cn)
rd = cmd.ExecuteReader
If rd.Read Then
Response.Redirect("User_Info.aspx")
Else
Response.Redirect("ErrLogin.aspx")
End If
cn.Close()
cmd.Dispose()
rd.Close()
End sub
```

Fig. 1: Example VB Code in .NET Application

2. RELATED WORK

There are existing techniques that can be used to detect and prevent input manipulation vulnerabilities.

2.1. Prepare Statement [8]

SQL provides the prepare statement, which separates the values in a query from the structure of SQL [13]. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at run time. The prepare statement makes it harder to inject SQL queries because the SQL structure cannot be changed. To use the prepare statement, we must modify the web application entirely; all the legacy web application must be re-written to reduce the possibility of SQL injections.

2.2. Instruction–Set Randomization[2][11]

SQL rand provides a framework that allows developers to create SQL queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQL rand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQL rand requires the application developer to rewrite code.

2.3. Proxy filter [2] [10]

Scott and Sharp use a proxy to filter input and output data streams for a web application although this technique can be effective against SQLIA; it requires developers to correctly specify filtering rules for each application input. This step of the process is prone to human error and leaves the application vulnerable if the developer has not adequately identified all injection points and correctly expressed the filtering rules. Like defensive coding practices this techniques cannot provide guarantees of completeness and accuracy.

2.4. Defensive Programming [11][12]

Programmers can implement their own input filters or use existing safe API s that prevent malicious input or that convert malicious input in to safer input. Some of these approaches require programmers to learn the usage of API s so, the programmers may not be willing to use them. Improper usage OS APIs also leads to attack. It is difficult to implement It address only a subset of the possible attack The cost and complexity of retrofitting existing code.

2.5. Static Analysis [5][13]

Find Bugs and Lapse are static analysis tools that can detect input manipulation vulnerabilities in java programs however they ignore control flow or perform weak control flow analysis and therefore do not recognize the existence of user

input filters in application precisely. As a result they may generate many false positives. We can't find out the vulnerabilities introduced at the run time. Time consuming if conducted manually.Wassermann and Su proposed an approach that uses a static analysis combined with automated reasoning. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that insert a tautology in the SQL queries, but cannot detect other types of SQL injection attacks.

3. Proposed Technique

This approach addresses SQLIA's with runtime monitoring. The key insights behind the approach are that (1) the source code contains enough information to infer models of the expected, legitimate SQL queries generated by the application, and (2) an SQLIA, by injecting additional SQL statements into a query, would violate such a model. Proposed technique monitors the dynamically generated queries with the Data model which is generated by X- Log Generator at runtime and checks them for compliance. If the Data Comparison violates the model then it represent potential SQLIA' s and prevented from executing on the database and reported. For each application, when the login page is redirected to our checking page, it was to detect and prevent attacks without stopping legitimate accesses. Moreover, our technique proved to be efficient, imposing only a low overhead on the Web applications. This technique consists of three filtration techniques to prevent SQLI'S. We summarize the steps and then describe them in more detail in subsequent sections.

Vulnerability Guard: Vulnerability Guard detects the Wildcard characters or Meta characters and prevents the malicious attacks.

X – Log Authentication: X-Log valuator validate from X-Log Generator where the Sensitive data's are Stored from the Database, The user input fields compare with the data existed in X-Log generator if it is identical then the query is allowed to proceed.

Stored Procedure: Testing the size and data type of input and enforce appropriate limit. Stored Procedures is used to validate user input and to perform server side validation. The safety of stored procedures depends on the way in which they are coded and on the use of adequate defensive coding practices. These Three input filtrations are used to improve the scalability, performance and optimization.

3.1. Identify Hotspot

Scan the application code to identify hotspots - points in the application code that issue SQL queries to the underlying database. This step performs a simple scanning of the application code to identify hotspots. For the example .NET in Figure 1, the set of hotspots would contain a single element: the statement at line 4. (In .NET based applications, interactions with the database occur through calls to specific methods in the System.Data.Sqlclient namespace, 1 such as Sqlcommand- . ExecuteReader (String))

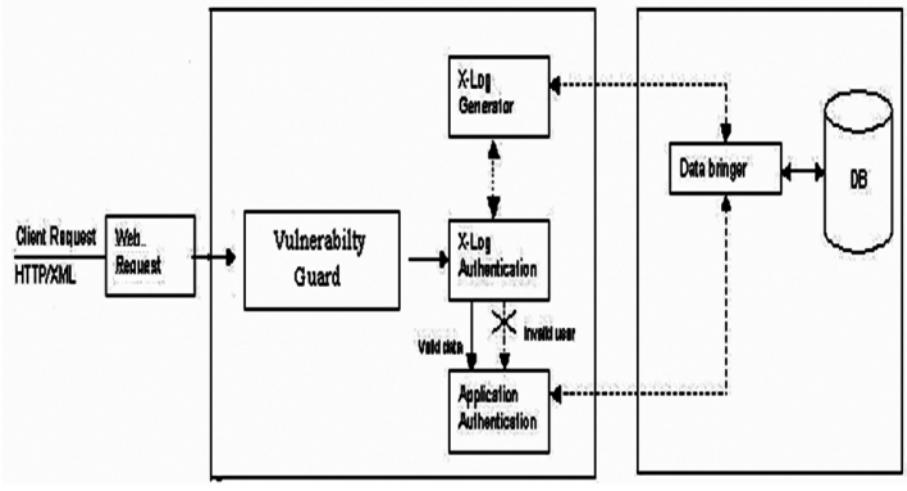

Fig. 2: Proposed Architecture

The injection process works by prematurely terminating a text string and appending a new command. Because the inserted command may have additional strings appended to it before it is executed, the malefactor terminates the injected string with a comment mark "—". Subsequent text is ignored at execution time. In our work we contribute a vulnerability Guard, to validate the user input fields to detect the wild card character (patch file) and prevent the malicious attacker. Transact-SQL statements will be prohibited directly from user input. For each hotspot, build a wildcard model, to check any malicious strings or characters append SQL tokens (SQL keywords and operators), delimiters, or string tokens to the legitimate command.

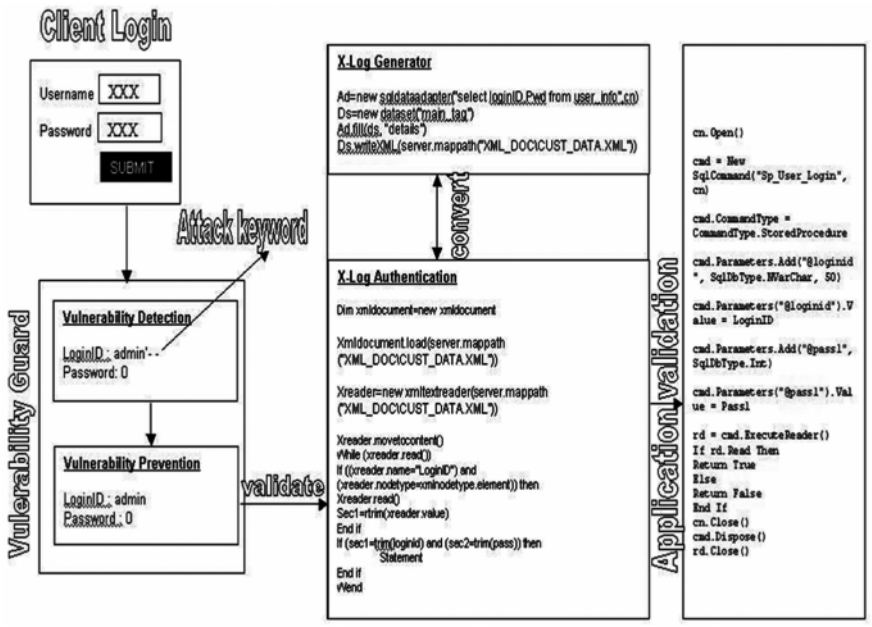

Fig. 3: Functions Generated in Vulnerability Guard, X.Log Generator, X-Log Authentication and Stored Procedure

3.2. Comparison of Data at Runtime Monitoring

In figure 2: At runtime, If the user input is given to a web application, the input is pattern matched with the patch file, if there is any meta character concatenated with user input then the vulnerability Guard check the dynamically generated queries against the Wild card model and reject queries that violate the model as a malicious user. If the pattern matching is identical with the patch file then it is termed as a legitimate user then the data comparison starts from XML-Guard. From the vulnerability Guard the validated user input fields compare with the X-Log Authentication where the Sensitive data is stored, X- Log Generator store the sensitive data from the valid database. Here there is no Query validation occurs only the data's will be validated in XML-Generator. Then the validated data send to the X-Log Authentication. If the script builds an SQL query by concatenating hard-coded strings together with a string entered by the user, As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. String concatenation is the primary point of entry for script injection Therefore, we Compare all user input carefully with X – Log Generator (Second filtration method).If the user input and Sensitive data's are identical then executes constructed SQL commands in the Application server.

3.3. Validation in Stored Procedure

In figure 3: Stored procedures can enhance data access security in several ways. Database users should be given permissions to execute stored procedure without being granted permissions to directly access the database objects on which the stored procedure operates. Besides, stored procedures should validate user input, and their parameters should not be treated as executable code. In multi tiered environments, all data should be validated before admission to the trusted zone. Data that does not pass the validation process should be rejected and an error should be returned to the previous tier.

Implement multiple layers of validation. Precautions you take against casually malicious users may be ineffective against determined attackers. For example, data validation in a client-side application can prevent simple script injection. However, if the next tier assumes that its input has already been validated, any malicious user who can bypass a client can have unrestricted access to a system.

4. EVALUATION

The proposed technique is deployed and tried few trial runs on the web server.

4.1. SQLIA Prevention Accuracy

Both the protected and unprotected web Applications are tested using different types of SQLIA's; namely use of Tautologies, Union, Piggy-Backed Queries, Inserting additional SQL statements, Second-order SQL injection and various other SQLIA s. Table 1 shows that the proposed technique prevented all types of SQLIA s in all cases. The proposed technique is thus a secure and robust solution to defend against SQLIA s.

Table 1
SQLIA'S Prevention Accuracy

SQL Injection Types	Un Protected	Protected
1. Tautologies	Not Prevented	Prevented
2. Piggy Backed Queries	Not Prevented	Prevented
3. Additional SQL-Statement	Not Prevented	Prevented
4. Second-Order	Not Prevented	Prevented
5. Union	Not Prevented	Prevented

4.2. Execution Time – Runtime Validation

The runtime validation incurs some overhead in terms of execution time at both the x – Log Authentication comparison and SQL-Query based Validation. Taken a sample website E –Transaction measured the extra computation time at the query validation, this delay has been amplified in the graph (figure: 4 and figure:5) to distinguish between the Time delays using bar chart shows Data validation performs better than Query validation. In Query validation(figure:4) the user input is generated as a query in script engine then it gets parsed in to separate tokens then the user input is compared with the statistical generated data if it is malicious generates error reporting. X-Log Authentication technique (figure: 5) states that user input is generated as a query in script engine then it gets parsed in to separate tokens, Simultaneously the xml bringer transmit the sensitive data from the database to the X – Log generator then the user input is compared with the legitimate data if it is malicious data, it will be prevented otherwise it allows to access the database in the database server. Existing techniques directly allows accessing the database in database server after the Query validation. Current technique does not allow directly to access database server because here automatically the X-Log Authentication access the data's from database and compare with the user input if user data is legitimate then it allows accessing the database in database server.

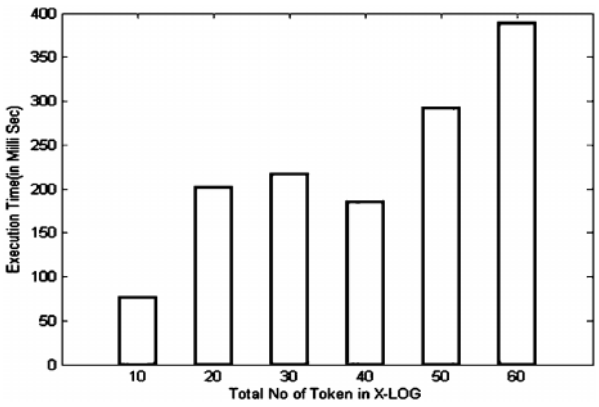

Fig. 4: Execution Time based on XML Data

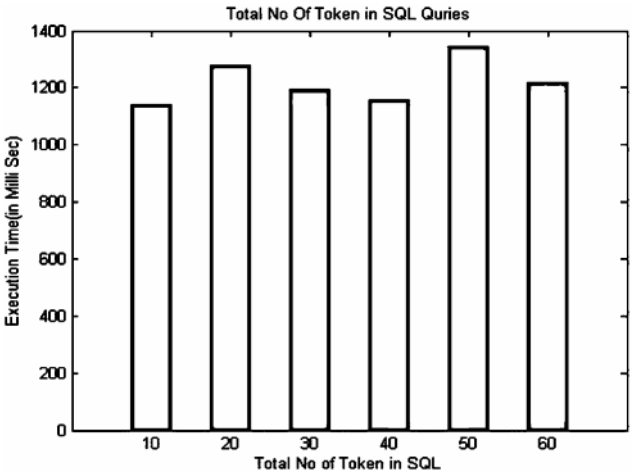

Fig. 5: Execution Time based on SQL Queries

5. DISCUSSION

Proposed technique was able to correctly identify all attacks as SQLIA's, while allowing all legitimate queries to be performed and no false positives and no false negatives are generated. Results are not compared with alternative approaches against SQLIA's because many of the automated approaches that we are aware of, only address a small subset of the possible SQLIA's. (For example, the one in [9] focuses only on tautologies.) Such approaches would not be able to identify many of the attacks in our test bed. As for all empirical studies, there are some threats to the validity of our evaluation, mostly with respect to external validity. The results of our study may be related to the specific subjects considered and may not generalize to other web applications. To minimize this risk, we used a set of web applications and an extensive set of realistic attacks.

6. CONCLUSION

The SQL Injection Attacks are extremely dangerous in comparison to other types of Web-based attacks, because the end result is data manipulation. SQL injection holes can be easy to exploit, a novel technique against SQLIA's. has the web-application code implicitly contains a policy that allows to distinguish legitimate and malicious queries. The technique is fully automated and detects, prevents, and reports SQLIA's. This technique is used to detect and prevent the SQLI flaw (wild characters & exploiting SQL commands) in vulnerability Guard and prevent the malicious attacker Transact-SQL statements will be prohibited directly from user input. X-Log Authentication checks the user input with valid database which is stored separately in X- Log Generator then the validated user input field is Send to Stored procedure Using stored procedures is to validate user input for Testing the size and data type of input and enforce appropriate limits. Stored procedure is used to improve the performance of the server side validation This

proposed technique was able to correctly identify the attacks that we performed on the applications without blocking legitimate accesses to the database (i.e., the technique produced neither false positives nor false negatives). These results show that our technique represents a promising approach to countering SQLIA's and motivate further work in this direction.

REFERENCES

[1] AMNESIA: Analysis and Monitoring for Neutralizing SQLInjection Attacks William G.J. Halfond and Alessandro Orso, College of Computing, Georgia Institute of Technology, ASE'05, November 7–11, 2005.

[2] A Classification of SQL Injection Attacks and Countermeasures: William G.J. Hal Fond and Alessandro Orso, College of Computing, Georgia Institute of Technology.Gatech.edu

[3] Eliminating SQL Injection Attacks - A Transparent Defence Mechanism Muthuprasanna, Ke Wei, Suraj Kothari Iowa State University, Ames, IA, USA SQL Injection Attacks Prof. Jim Whitehead CMPS 183. Spring 2006, May 17, 2006.

[4] WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation William G.J. Hal fond, Alessandro Orso, Member, IEEE Computer Society, and Panagiotis Manolios, Member, IEEEComputer Society. IEEE Software Engineering, 34, No.1, January/February 2008.

[5] "Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/topten.html, 2005.

[6] Web Cohort Inc., "Only 10% Web Applications Secured against Common Hacking Techniques", Stephen Thomas, Laurie Williams. Using Automated Generation to Secure SQL Statements Third International Workshop on Software Engineering for Secure Systems(SESS'07), Pages 9-May 2007.

[7] G.Wassermann and Z. Su., "Static Analysis Framework for Security in Web Application for Detecting SQL Injection Vulnerabilities", In Proceeding of the Conference, Pages 70-78, 2004.

[8] Stephen Thomas, Laurie Williams, "Using Automated Generation to Secure SQL Statements Third International Workshop on Software Engineering for Secure Systems(SESS'07)", Pages 9-May 2007.S.Boyd and A.Keromytis: SQLrand: Preventing SQL Injection Attacks. In Proceedings of the Applied Cryptography and Network Security (ANCS), Pages 292-304, 2004.

[9] W. R. Cook, S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries", ICSE 2005.

[10] D. Scott and R. Sharp, "Abstracting Application-level Web Security", In Proceedings of the 11th International Conference on the World Wide Web (WWW 2002), Pages 396–407, 2002.Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo.

[11] "Securing Web Application Code by Static Analysis and Runtime Protection", In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[12] SQL Injection Attack Examples based on the Taxonomy of Orso et al.

[13] V. B. Livshits, "Finding Security Errors in Java Programs with Static Analysis", In Proceedings of the 14th Use nix Security Symposium, Pages 271–286, Aug.2005.