An Approach to Detect and Prevent SQL Injection Attacks using Web Service

Shabnam R. Makanadar[1], Vaibhav V. Solankurkar[2]

[1, 2] Students, Department of Computer Science, Bharati Vidyapeeth's College of Engineering, Kolhapur, Maharashtra, India

[1]*makandarshabnam777@gmail.com*,

[2]*vaibhavsolankurkar@gmail.com*

Abstract: ***This paper includes SQL injection attacks which are a methodology, which targets the information in a database. The injections may occur due to the poor input validation code. When attacker inserts a series of SQL queries, there is a possibility of SQL injection. In the several kinds of Browsers, Software's or websites which contain confidential information, SQL injection attack may leak confidential information, such as credit card numbers, from web applications, databases and even corrupt the database by an attacker. This paper deals with the development of detecting and preventing SQLIA's application running with Asp.Net platform using web services.***

Keywords: Database security, world-wide web, web application security, SQL injection attacks, Runtime Monitoring

1. Introduction

SQL Injection attacks are one of the most common hacker attacks used on the web. SQL injection is a software vulnerability that occurs when data entered by users is sent to the SQL interpreter as a part of an SQL query. Attackers provide specially crafted input data to the SQL interpreter and trick the interpreter to execute unintended commands. Attackers utilize this vulnerability by providing specially crafted input data to the SQL interpreter in such a manner that the interpreter is not able to distinguish between the intended commands and the attacker's specially crafted data. The interpreter is tricked into executing unintended commands. A SQL Injection attack exploits security vulnerabilities at the database layer. By exploiting the SQL injection flaw, attackers can create, read, modify, or delete sensitive data. The database holds the content, the users IDs, the settings, and more. To gain access to this valuable resource is the ultimate prize of the hacker. Accessing this can gain him/her an administrative access that can gather private information such as usernames and passwords, and can allow any number of bad things to happen.

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 Prevention of SQLIA's is a major challenge. It is difficult to implement and enforce a rigorous defensive coding discipline.

2. Databases and SQL

2.1 General information—The most popularly-used databases in CS environments are IBM's DB2, Oracle, and Microsoft SQL Server; all of these allow queries and commands to be passed to the database via SQL. Common uses for the databases include: a) simple data storage, b) information linking, and c) credential authentication.

2.2 Database components—Databases, in this paper, will refer to the Database Management System (DBMS) that may manage relational data structures (RDBMS), flat files, extensible markup language (XML) data, etc. The DBMS is the software which handles all aspects of data management including physical storage, reading and writing data, security, replication, error correction, and other functions.

2.3 SQL—SQL is a computer language for communication with databases. The communicating parties are typically a "front end" application that sends a SQL statement across a connection to a database that holds the data. SQL has many capabilities, but the most common needs in business are to:

- Read, analyze and report on existing data,
- Create new records holding data,
- Modify existing data,
- Append data to an existing record, and
- Delete data or records.

3. Techniques Of SQLIA's

The different types of attacks are generally not performed in isolation; many of them are used together or sequentially, depending on the specific goals of the attacker. Note also that there are countless variations of each attack type. For space reasons, we do not present all of the possible attack variations but instead present a single representative example.

3.1 Tautologies

The general goal of a tautology-based attack is to inject code in one or more conditional statements so that they always evaluate to true. The consequences of this attack depend on how the results of the query are used within the application. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an injectable field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the

query to be returned. In general, for a tautology-based attack to work, an attacker must consider not only the injectable/vulnerable parameters, but also the coding constructs that evaluate the query results. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned. *Example*: In this example attack, an attacker submits " ' or 1=1 - - " for the *login* input field (the input submitted for the other fields is irrelevant). The resulting query is:

SELECT accounts FROM users WHERE login='' or 1=1 -- AND pass='' AND pin=

The code injected in the conditional (OR 1=1) transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them.

3.2 Union Query

In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Attackers do this by injecting a statement of the form: UNION SELECT <rest of injected query>. Because the attackers completely control the second/injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query. *Example*: Referring to the running example, an attacker could inject the text "' UNION SELECT cardNo from CreditCards where acctNo=10032 - -" into the login field, which produces the following query:

SELECT accounts FROM users WHERE login='' UNION SELECT cardNo from CreditCards
where acctNo=10032 -- AND pass='' AND pin=

Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "Credit Cards" table. In this case, the database would return column "cardNo" for account "10032." The database takes the results of these two queries, unions them, and returns them to the application.

3.3 Stored Procedures

SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors ship databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system.

Additionally, because stored procedures are often written in special scripting languages, they can contain other types of vulnerabilities, such as buffer overflows, that allow attackers to run arbitrary code on the server or escalate their privileges [9].

```
CREATE PROCEDURE DBO .is Authenticated
@userName varchar2, @pass varchar2, @pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" +@userName+ "' and pass='" +@password+
"' and pin=" +@pin);
GO
```

Stored procedure for checking credentials.

Example: This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, we assume that the query string constructed at lines 5, 6 and 7 of our example has been replaced by a call to the stored procedure defined in Figure 2. The stored procedure returns a true/false value to indicate whether the user's credentials authenticated correctly. To launch an SQLIA, the attacker simply injects " ' ; SHUTDOWN; --" into either the userName or password fields. This injection causes the stored procedure to generate the following query:

SELECT accounts FROM users WHERE
login='doe' AND pass=' '; SHUTDOWN; -- AND pin=

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down.

3.4 Extended Stored Procedures

Using Extended stored procedure attack ,an attacker can Reset the IIS(Internet Information Services).There are several extended stored procedures that can cause permanent damage to a system. Extended stored procedure can be executed by using login form with an injected command as the LoginId:';execmaster..xp_xxx;-- Password:[Anything]
LoginId:';execmaster..xp_cmdshell'iisreset';--
Password:[Anything]
select password from user_info where LoginId='';
exec master..xp_cmdshell 'iisreset'; --' and Password=''

This Attack is used to stop the service of the web server of particular Web application. Stored procedures primarily consist of SQL commands, while XPs can provide entirely new functions via their code. An attacker can take advantage of extended stored procedure by entering a suitable command. This is possible if there is no proper input validation. xp_cmd shell is a built-in extended stored procedure that allows the execution of arbitrary command lines.

3.5 Alternate Encodings

In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques. This attack type is used in conjunction with other attacks. In other words, alternate encodings do not provide any unique way to attack an application; they are simply an enabling technique that allows attackers to evade detection and prevention

techniques and exploit vulnerabilities that might not otherwise be exploitable. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment operators. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Common scanning and detection techniques do not try to evaluate all specially encoded strings, thus allowing these attacks to go undetected. Contributing to the problem is that different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding. For example, a database could use the expression char (120) to represent an alternately-encoded character "x", but char(120) has no special meaning in the application language's context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here we simply provide an example of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the *login* field:

"legalUser'; exec (0x73687574646f776e) - -".

The resulting query generated by the application is:

SELECT accounts FROM users WHERE login='legalUser'; exec(char(0x73687574646f776e)) -- AND pass='' AND pin=

This example makes use of the char() function and of ASCII hexadecimal encoding. The char() function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string SHUTDOWN." Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the SHUTDOWN command.

4. Related Work

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

4.1 Web Vulnerability Scanning

Web vulnerability scanners crawl and scan for web vulnerabilities by using software agents. These tools perform attacks against web applications, usually in a black-box fashion, and detect vulnerabilities by observing the applications' response to the attacks. However, without exact knowledge about the internal structure of applications, a black-box approach might not have enough test cases to reveal existing vulnerabilities and also have false positives.

4.2 Intrusion Detection System (IDS)

Valeur and colleagues propose the use of an Intrusion Detection System (IDS) to detect SQLIA. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model in that it builds expected query models and then checks dynamically-generated queries for compliance with the model. Their technique, however, like most techniques based on learning, can generate large number of false positive in the absence of an optimal training set. Su and Wassermann propose a solution to prevent SQLIAs by analyzing the parse tree of the statement, generating custom validation code, and wrapping the vulnerable statement in the validation code. They conducted a study using five real world web applications and applied their SQLCHECK wrapper to each application. They found that their wrapper stopped all of the SQLIAs in their attack set without generating any false positives. While their wrapper was effective in preventing SQLIAs with modern attack structures, we hope to shift the focus from the structure of the attacks and onto removing the SQLIVs.

4.3 Combined Static and Dynamic Analysis

MNESIA is a model-based technique that combines static analysis and runtime monitoring. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIA's and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIA's. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives Livshits and Lam use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIA's and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology. The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

5. Proposed Technique

This Technique is used to detect and prevent SQLIA's with runtime monitoring. The solution insights behind the technique are that for each application, when the login page is redirected to our checking page, it was to detect and prevent SQL Injection attacks without stopping legitimate accesses. The contribution of this work is as follows: A new automated technique for preventing SQLIA's where no code modification required, Web service which has the functions ofdb_2_XMLGenrerator and XPATH_ Validator such that it is an XML query language to select specific parts of an XML document. XPATH is simply the ability to traverse nodes from XML and obtain information. It is used for the temporary storage of sensitive data's from the database, Active Guard model is used to detect and prevent SQL Injection attacks. Service Detector model allow the Authenticated or legitimate user to access the web applications. If the Data Comparison violates the model then it represents potential SQLIA's and prevented from executing on the database. This proposed technique consists of two filtration models to prevent SQLIA'S. 1) Active Guard filtration model 2) Service Detector filtration model. The steps are summarized and then describe them in more detail in following sections.

5.1 Active Guard Filtration Model

Active Guard Filtration Model in application layer build a Susceptibility detector to detect and prevent the Susceptibility characters or Meta characters to prevent the malicious attacks from accessing the data's from database.

5.2 Service Detector Filtration Model

Service Detector Filtration Model in application layer validates user input from XPATH_Validator where the Sensitive data's are stored from the Database at second level filtration model. The user input fields compare with the data existed in XPATH_Validator if it is identical then the Authenticated /legitimate user is allowed to proceed.

5.3 Web Service Layer

Web service builds two types of execution process that areDB_2_Xml generator and XPATH_ Validator. DB_2_Xml generator is used to create a separate temporary storage of Xml document from database where the Sensitive data's are stored in XPATH_ Validator, The user input field from the Service Detector compare with the data existed in XPATH_ Validator, if the data's are similar XPATH_ Validator send a flag with the count iterator value = 1 to the Service Detector by signifying the user data is valid.

5.4 Identify Hotspot

This step performs a simple scanning of the application code to identify hotspots. Each hotspot will be verified with the Active Server to remove the susceptibility character the sample code (figure: 2) states two hotspots with a single query execution.(In .NET based applications, interactions with the database occur through calls to specific methods in the System Data. Sql client namespace, 1 such as Sql command- . Execute Reader (String)) the hotspot is instrumented with monitor code, which matches dynamically generated queries against query models. If a generated query is matched with Active Guard, then it is considered an attack.

5.5 Comparison of Data at Runtime Monitoring

When a Web application fails to properly sanitize the parameters, which are passed to, dynamically created SQL statements (even when using parameterization techniques)it is possible for an attacker to alter the construction of back-end SQL statements.

6. Architecture

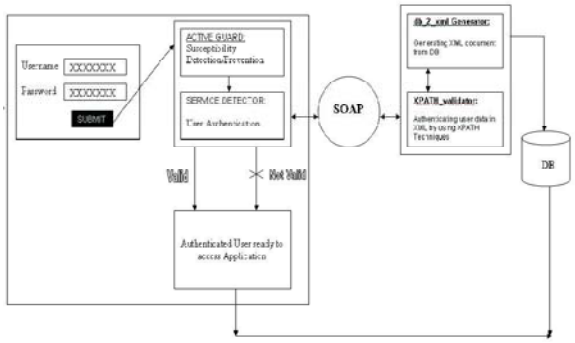

Figure 1. Architecture of the proposed approach implementation

7. Conclusion

SQL Injection Attacks attempts to modify the parameters of a Web-based application in order to alter the SQL database. Any procedure that constructs SQL statements could potentially be vulnerable, as the diverse nature of SQL and the methods available for constructing it provide a wealth of coding options.

8. Acknowledgment

I express my deep sense of gratitude and appreciation towards my research guide Prof. A. M. Patravale for his continuous inspiration and valuable guidance in throughout my dissertation work.

References

[1] Indrani Balasundaram and Dr. E. Ramaraj, "An Approach to Detect and Prevent SQL Injection Attacks using Web Service."
[2] William G.J. Hal fond and Alessandro Orso , "AMNESIA: Analysis and Monitoring for Neutralizing SQL Injection Attacks"
[3] W. G. J. Halfond and A. Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL Injection Attacks".