Proposed Technique on 3-Tier Architecture for Developing SQL-Injection Attacks Proof Website

Nida Khan, Abdul M.Siddiqui
Dept.of CSE, RGPV Bhopal, M.P., India

Abstract: The aim of SQL Injection Attacks is to exploit the vulnerabilities of web applications with the intentions to execute the malicious SQL commands. Such type of attacks takes an advantage of poor input validation in code and web administration. They generally occurs when an attacker tried to teem a series of SQL statements into a query by frame up the user input data into web based applications, The attacker can also take advantages of flaws of web application programming security and pass erroneous malicious SQL statements. This paper proposes a new methodology for the prevention & detection of SQL injection Attacks. The proposed solution to make a secure website is to use 3-tier framework for developing web applications. In each tier, we analyze the inputted data taken from the user and make a decision, whether that is suspected or not.

Keywords: SQL Injection, Avoidance, SQLIA Prevention, SQLIA Detection, SQL Attacks.

I. INTRODUCTION

The SQL Injection is a very common type of injection attack for any Web application, in which the attackers usually provides SQL code to a user input control of a Web form to gain unauthorized and unlimited access. The attacker’s input is transmitted into an SQL query in such a way that it will form an SQL code [1], [2]. From past years many organizations provides their services on the Internet. For that they keep the information regarding their customers, their partners which is quit sensitive for that they maintain the database driven web application.

The main objective of SQL injections is to query the database in a manner that was not the intent of the application programmer. There are various techniques used in SQL injection. Most of them use SQL statement in different SQL injection techniques. Increased dependence on web applications significantly and use in the activities of our daily lives grow in the number and level of attacks that target them.

II SQL INJECTIONS: A THREAT TO WEBSITE

In today’s scenario, Information technology enabled services are very usual in commercial as well as in government sector, which results number of ecommerce website. These websites helps the users to make personal account there for online transactions, or to store their confidential data. Almost all the organization builds their personal servers to which all the organizational records and data are managed. In this way, the data and other web content become very precious to any organization or individual. As the popularity of internet increases, the use of online and automated processes are also, increases and therefore huge bulks of sensitive and critical data are being handled by the web applications. As the stakes on the information and data stored by the portals become higher, so does the sophistication of hackers. Developers and hackers are racing against each other. Developers try to make the web application secure from the threats and the hacker wish to find the loophole, so that it can steal or damage the application or data. Security threats could be with the intent of stealing confidential information, causing deliberate damage, prove capability or simply for the thrill of doing something which most others cannot do.

Therefore, a developer needs to take all the precautions to secure the organizations data by avoiding or preventing all the security threats to the website. Usually following are the most common threats to a website:

1. Brute Force: the attacker simply keep on guessing and trying various combinations, most often using an automated script, to gain unauthorized access to a system.
2. Cross-Site Scripting: Cross-site scripting (XSS) is a type of website security vulnerability which allows attackers to inject client-side script in web pages viewed by users of the website.
3. Insufficient Anti-Automation: when a web site permits an attacker to automate a process that should only be performed manually.
4. Denial of Service: In a denial-of-service attack, an attacker may block access of a website to its legitimate users. Most of the times, this is done by flooding.
5. Network Sniffers: Network sniffer can list all of the network packets in real-time from multi network card and can support capturing packets based on the applications (Socket, TDI, etc). Attacker can observe the traffic of the application to identify the loopholes for attacks.
6. SQL Injections: SQL injection is an attack in which malicious code is inserted into strings, which are later passed to the database server for parsing and execution.

III. VULNERABILITIES DUE TO DATABASE
One of the major vulnerability to website is the vulnerability due to the databases. This may be the loss of data, damage to database, unauthorized access, bypassing the authentication mechanisms or denial of services. These actions can be done by the following ways:

Injection through cookies: Cookies are those files that contain state information that are generated by Web applications and stored on the client machine. When a client returns to a Web application, cookies can be used to restore the clients state information. Since the client has control over the storage of the cookie, a malicious client could tamper with the cookies contents. If a Web application uses the cookies contents to build SQL queries, an attacker could easily submit an attack by embedding it in the cookie [3]. Injection through user input: The attackers inject SQL commands by providing suitably crafted user input. In most SQLias that target Web applications, user input typically comes from form submissions that are sent to the Web application via HTTP GET or POST requests [8]. Web applications are generally able to access the user input contained in these requests as they would access any other variable in the environment.

IV. PROBLEM DEFINITION
The SQL injections are the most common yet critical threats to any website as it is concern to the database, which is the valuable to any organization. Therefore, there must be some rules that one should be incorporated in every website to make it secure from SQL injections. Many Web applications can be exploited because the user input is being processed in an unsafe manner. All the data provided by a user must be treated as untrustworthy. One of the key requirements for a Web application’s security is the proper user input handling, which is not always an easy task. To propose the classification the inputs based on probability and use of character as a vulnerability that helps to identify in SQL detection process. Proper neutralization of such special characters used in an SQL Command to avoid the SQL injection.

V. RELATED WORK
Although the SQL injection is not a new field for research but it is one of the most popular fields for researchers. Contribution to this field such as filtering, information flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQL Injections Attacks. Some of the researches we studied for our references are shown below:

William G.J. Halfond et al.’s Scheme- [2]- This approach works by combining static analysis and runtime monitoring. SAFELI – [5] proposes a Static Analysis Framework in order to detect SQL Injection Vulnerabilities. SAFELI framework aims at identifying the SQL Injection attacks during the compile-time. This static analysis tool has two main advantages. Firstly, it does a White-box Static Analysis and secondly, it uses a Hybrid-Constraint Solver. Thomas et al.’s Scheme - Thomas et al., in [6] suggest an automated prepared statement generation algorithm to remove SQL Injection Vulnerabilities. They implement their research work using four open source projects namely: (i) Net-trust, (ii) ITrust, (iii) WebGoat, and (iv) Roller. Based on the experimental results, their prepared statement code was able to successfully replace 94% of the SQLIVs in four open source projects.

Ruse et al.’s Approach - In [7], Ruse et al. propose a technique that uses automatic text case generation to detect SQL Injection Vulnerabilities. The main idea behind this framework is based on creating a specific model that deals with SQL queries automatically. Ali et al.’s Scheme - [8] adopts the hash value approach to further improve the user authentication mechanism. They use the user name and password hash values SQLIPA (SQL Injection Protector for Authentication) prototype was developed in order to test the framework. Roichman and Gudes Scheme – [9] suggests using a fine grained access control to web databases. The authors develop a new method based on fine-grained access control mechanism. The access to the database is supervised and monitored by the built-in database access control.

SQL-IDS Approach - Kemalis and Tzouramanis in [10] suggest using a novel specification-based methodology for the detection of exploitations of SQL injection vulnerabilities. The proposed query-specific detection allowed the system to perform focused analysis at negligible computational overhead without producing false positives or false negatives.

SQLrand Scheme - SQLrand approach [11] is proposed by Boyd and Keromytis. For the implementation, they use a proof of concept proxy server in between the Web server (client) and SQL server; they de-randomized queries received from the client and sent the request to the server. SQLIA Prevention Using Stored Procedures - Stored procedures are subroutines in the database which the applications can make call to [12]. Parse Tree Validation Approach - Buehrer et al. [13] adopt the parse tree framework. They compared the parse tree of a particular statement at runtime and its original statement. They stopped the execution of statement unless there is a match.

Dynamic Candidate Evaluations Approach - In [14], Bisht et al. propose CANDID. It is a Dynamic Candidate Evaluations method for automatic prevention of SQL Injection attacks. This framework dynamically extracts the query structures from every SQL query location which are intended by the developer (programmer). Hence, it solves the issue of manually modifying the application to create the prepared statements.

Input rectification was first described by Rinard et al. in [20]. The idea is to define a “comfort zone” for which software has been well-tested and is highly unlikely to fail. This idea was extended to automatically defining the comfort zone and rectifying anomalous inputs to forms in the comfort zone [19]. This idea has also been explored in the particular application of enforcing consistency in data structures [21].
VI. PROPOSED ARCHITECTURE

The proposed technique consists of four-stage security mechanism, which are as follows:

1. Presentation Tier: This tier works on client side “web page at browser”; it detects the use of special characters with the help of regular expressions. This technique is called validations. With this mechanism, we are able to protect the web page controls like (textboxes) from any kind of special character, which is meaningful to the SQL. The only drawback of this stage is that, it cannot handle the attacks from the query strings, cookies etc.

2. Business Tier: This stage works on the code written for the processes. This stage detects the malicious SQL code, inject by various other techniques like query strings, URL’s. The data filtration stage analyzes the data before passing on to the database.

3. Data Access layer: Parameterized query is parameterized database access API provided by development platform such as Prepare Statement in Java or SQL Parameter .NET. Instead of composing SQL by concatenating string, each parameter in a SQL query is declared using place holder and input is provided separately. Parameterized queries keep the query and the data separate through the use of placeholders known as "bound" parameters.

   This helps in preventing SQLIA by not allowing the structure of the query to be altered; rather it merely “fills in” the input parameters into their positions and keeps the rest of the query structure intact. Since a majority of the SQLIA techniques rely on altering the query structure for injection attacks, this serves as a very effective combative technique. The sanitizer classify the inputs as follows-

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning in Transact-SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Query delimiter</td>
</tr>
<tr>
<td>'</td>
<td>Character data string delimiter.</td>
</tr>
<tr>
<td>--</td>
<td>Comment delimiter.</td>
</tr>
<tr>
<td>/* ... */</td>
<td>Comment delimiters. Text between /* and */ is not evaluated by the server.</td>
</tr>
<tr>
<td>xp_</td>
<td>Used at the start of the name of catalog-extended stored procedures, such as xp_cmdshell.</td>
</tr>
</tbody>
</table>

In fig. 1, It is clearly seen that, how we can monitor the data at different stages during the working of GET & POST methods.

Following is the algorithm, used in the proposed methodology for the detection and prevention of SQL injections.

In fig. 2, The simple idea to implement the SQL injection detection mechanism is to check each & every input before passing it to the database. For this, we need to perform an analysis that defines us the suspected inputs for different data input fields. Therefore, Based on this analysis we can easily identify the nature of inputs, whether it is an RTF input or non-RTF input.
VII. IMPLEMENTATION

The complete implementation of the proposed work carried out using the Microsoft .Net framework 3.5 with Visual Studio 2008, SQL Server 2008, IIS Server and a vulnerability scanner tool Netsparker. Through which we scan our own developed website for any vulnerability threat and SQL attacks.

We also demonstrate the whole project by a web application developed by VB.Net & ASP.Net along with the database in SQL Server 2008. The demonstrated website works in two modes at every instance. Whether its login or search, inserting or updating, etc. Following are the two modes:

1. SAFE Mode: In this mode the website works under the guidance of Sanitizer, which sanitizes the inputs and always looking for any intrusion or malicious code in the input. The development of sanitizer leads to a mechanism that helps to decide whether the input data is legitimate and does not contain any malicious code.

   The most important feature of our proposed plan for detecting & preventing the SQL injections is that it never overhead the data in database. It can be apply to any existing web model without perform any alteration in database schema.

Figure 2: Algorithm for SQL Injection Detection & Prevention
Proposed Login Query: Stored procedure

```sql
CREATE PROCEDURE [dbo].[spsecurelogin]
@username nvarchar(20),
@password nvarchar(20)
AS
BEGIN
SET NOCOUNT ON;
SELECT
   username, password
FROM
   tbl_user
WHERE
   hashbytes('SHA',username) = hashbytes('SHA',@username)
   AND
   hashbytes('SHA',password) = hashbytes('SHA',@password);
END
```

The above stated query is free from any malicious code, in-fact if user somehow transmits malicious data to the SQL procedure; our proposed SQL query will compute the Hash values with “SHA” (128 Bits) then it will be compared with database. Therefore, due to applying operation of hash values the input values are sanitizes completely and guarantees you solution in legal way.

Similarly, with the handling of query strings: The QueryString collection is used to retrieve the variable values in the HTTP query string. The HTTP query string is specified by the values following the question mark (?), like this:

```
Response.Redirect
("admin.aspx?adm_name=nida")
```

The line above generates a variable named adm_name with the value "nida". The Query strings are also generated by form submission, or by a user typing a query into the address bar of the browser

While at the redirected page, this query string handled by the code for further use like

```
Dim admin_name as String
admin_name = Request.QueryString("adm_name")
```

hence, it is the duty of the application programmer to check the validity of this query string value and make sure that, there is no malicious characters or code in the value.

The IsValid Code looks like:

```vbnet
Public Function IsValid(ByVal str As String) As Boolean
    Dim rx As New Regex
    
    Dim bool As Boolean = rx.IsMatch(str)
    Return bool
End Function
```

```
Dim admin_name as String
admin_name = Request.QueryString("adm_name")
```

While, the plain text sanitizer simply checks the presence of any trouble making characters.

The PT_Sanitizer code looks like:

```vbnet
Public Function PT_Sanitizer (ByVal Str As String) As Boolean
    Dim sb As Byte() = System.Text.ASCIIEncoding.ASCII.GetBytes(Str.ToCharArray())
    Dim i As Integer = 1
    While i <= Str.Length - 1
        If (sb(i) = 32) Or (sb(i) >= 65) And (sb(i) <= 90) Or (sb(i) >= 97) And (sb(i) <= 122) Or (sb(i) >= 48) And (sb(i) <= 57) Then
            i = i + 1
        End If
    End While
    Return True 'Valid Query
End Function
```

© 2013, IJARCSSE All Rights Reserved
2. UNSAFE Mode: The unsecure mode in the demonstrating website is never uses any SQL injection detection or prevention strategy, it simply pass the data from the HTML input to the database.

VIII. EXPERIMENTAL RESULT & ANALYSIS

Table 2: Result obtained by using the vulnerability scanner

<table>
<thead>
<tr>
<th>Total Request</th>
<th>Injections Detected</th>
<th>Normal Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>19560</td>
<td>12098</td>
<td>7462</td>
</tr>
<tr>
<td>125076</td>
<td>32697</td>
<td>92379</td>
</tr>
<tr>
<td>30867</td>
<td>21087</td>
<td>9780</td>
</tr>
<tr>
<td>8963</td>
<td>3601</td>
<td>5362</td>
</tr>
</tbody>
</table>

While working with demonstrating website, by using the vulnerability scanners and concurrent users, we successfully generate the above number of request in four different iterations. The proposed implemented system gives the appropriate results by analysing the inputs provided to them and finally output the counts of the Valid & Injected queries.

While working with demonstrating website, by using the vulnerability scanners and concurrent users, we successfully generate the above number of request in four different iterations. The proposed implemented system gives the appropriate results by analysing the inputs provided to them and finally output the counts of the Valid & Injected queries.

![Figure 3: Graphical representation of results](image)

Following is comparative study of existing SQL detection techniques with the proposed technique.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Tautology</th>
<th>Illegal Structured</th>
<th>Piggybacking</th>
<th>Union</th>
<th>Stored Procedure</th>
<th>Inference</th>
<th>Alternate Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROPOSED</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hash + Salt[18]</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQL GUARD[13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMNESIA[2]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

![Figure 4: Comparison with existing system for SQL injections detection & prevention](image)

IX. CONCLUSION

In this paper, we have concentrated on the specific area of SQL injection. According to OWASP’s Ten Most Critical Web Application Security Vulnerabilities [16], many SQL injection-related issues are among the most harmful threats to web applications. Since we have in this thesis only covered SQL injection aspects, we would like to suggest that further studies should be made on other threats to related security issues, especially such that relate to application security. The reason is that several authors have mentioned that organizations spend most security resources on operating system and network level security [15, 16, 17], and not enough on application layer security. If further studies will be made on application layer security issues, and particularly on web application, it would be possible to compile results from all of these into general security guidelines, which could be used in developing more secure web applications.

REFERENCES


