4. DATA CLEANSING ALGORITHM

4.1 INTRODUCTION

Web Applications have a three tiered framework, namely the client tier, the application server tier, and the database tier. If information can be retrieved from the database through the server by an unauthorized person, it is considered as a malicious attack. The attack can happen due to many reasons. It may occur because the code has a loophole which could be exploited, the input from the user may not be validated, and the output from the database may not be validated, due to poor programming practices and such others. Failing to check the inputs from the user or input to the database will lead to the input validation based attacks, namely the SQL Injection Attack (SQLIA) and the cross site scripting (XSS) attack.

An SQL injection attack occurs when an attacker causes the Web application to generate SQL queries that are functionally different from what the developer intended. Sensitive data can be extracted from a database that is prone to be vulnerable. The attack can tamper the existing data, destroy data or it can execute administrative operations on a database, which could be fatal for the vendor. There are different kinds of SQL injection attack.

- Tautologies:

Tautology means a formula that is true for every possible interpretation. Tautological SQL injection attack is a technique of structuring an SQL statement such that the interpretation of the statement is always true. For example, the SQL statement in the application would be,

\[
\text{Select * from user where userid} = \text{'+text1.userid’ and password} = \text{’+text2.password ’+ ‘};
\]
If the user submits the input as *admin* for user id and *xx or ‘1 = 1’* -- for password, then the statement becomes, and

```sql
Select * from user where userid= ‘admin’ and password = ‘xx’ or ‘1 = 1’ -- ‘;
```

The password is checked and since an *‘or’* statement checks for *1=1* condition, which is always true, the access to admin is granted. In SQL *‘--’* is a comment line and so anything that follows *‘--’* is not validated.

- Union queries

A technique of using the UNION operator in the SQL statement to combine two statements to extract data from the database is called an SQL injection attack using union queries.

In the previous example, if the user submits the input for userid as, *wrong’ UNION select * from user where userid= ‘admin’ --’, and password as *wrong, the statement then becomes,*

```sql
Select * from user where userid= ‘wrong’ UNION select * from user where userid=’admin’ --’ and password=’wrong’;
```

The server now retrieves all the data pertaining to the admin without the right password.

- Piggy-backed queries

Providing multiple queries for execution is called piggy-backed queries. These queries are separated by semicolons. This technique could be used to cause an SQL injection attack. Using the previous example, if the user submitted input is *admin’; Drop table user; --’* for userid and *1234* for password, then the statement becomes,

```sql
Select * from user where userid= ‘admin’; Drop table user; --’ and password =’1234’;
```
In SQL many statements can be given together using ‘;’ after each statement. In this example, the table user is deleted and statement after the comment line – is ignored.

The Cross Site Scripting Attack (XSS) mainly happens when the validation on the input and the output from the user is insufficient. The hacker can write a script, most commonly a JavaScript, and submit it to the server. If the server does not validate the input from the user, then this script gets included as part of the main code. This malicious script could contain code that would trap a user’s session identifier (ID) or a cookie and send it to the hacker. If a genuine user makes a request for this infected page, the server responds to this request by sending the malicious script to the user, which in turn traps the user’s information. A system was designed and developed to identify such SQL injection attacks and XSS attacks.

4.2 SYSTEM ARCHITECTURE

The input from the client is validated before submitting the client data to the server and further to the database. In a client server model, a reverse proxy server is placed, in between the client and the server. The presence of the proxy server is not known to the user. The proposed sanitizing application is placed in the reverse proxy server. The reverse proxy server is called the Intrusion Prevention proxy (IP proxy). A reverse proxy is used for the validation of the input because as the number of requests to the server increases, more reverse proxy’s can be used to handle those requests. This enables the system to maintain a low response time, even at a high load. The architecture of the proposed system is given in Figure 4.1.

When compared with 160 bit digest for Secure Hash Algorithm-1 (SHA-1), the 128 bit digest for Message Digest 5 (MD5) algorithm is faster to compute. In client server environment the response time for a request is very important. Hence, the user login data is hashed using MD5 algorithm.
The workflow is described as follows:

1. The client sends the request to the server.

2. The request is redirected to the reverse proxy.

3. The sanitizing application in the proxy server extracts the Uniform Resource Locator (URL) from the Hypertext Transfer Protocol (HTTP), the user data from the SQL statement and the user data from the script code and following action is done:

   a. The URL is sent to the signature check

   b. The user login data (extracted using prototype query model) is encrypted using the MD5 algorithm.

   c. The user input data (extracted using prototype query model) from script code is sent for expression check.
4. The sanitizing application sends the validated URL, validated user input, hashed user login data and the original request to the Web application in the server.

5. The filter in the server denies the request if the sanitizing application had marked the URL request malicious or the user input data malicious.

6. If the URL is found to be benign, then the hashed value is sent to the database of the Web application.

7. If the URL and the user input data are found benign, then the user input data in the feedback / comment form are included in the original code of the Web page.

8. If the hashed user data matches the stored hash value in the database, then the data are retrieved and the user gains access to the account, orelse the user is denied access.

The flow chart of the system is shown in Figure. 4.2. The sanitizing application uses the Data Cleansing algorithm that is designed to detect and mitigate the SQL injection Attack and Cross site scripting attack.

4.3 DATA CLEANSING ALGORITHM

The Data Cleansing (DC) Algorithm separates the user data from the request. It validates the data and sends the result to the server. The data cleansing algorithm is given in Figure. 4.3.

Extracting user data from SQL statement:

The SQL statement is extracted from the HTTP request and the query is tokenized. The tokenized query is then compared with the prototype document. A prototype document consists of all the SQL queries from the Web application. The query tokens are transformed into XML format. The
Extensible Stylesheet Language's (XSL) pattern matching algorithm is used to find the prototype model corresponding to the received query.

![Flowchart of Input Validation System](image)

**Figure. 4.2: Flowchart of Input Validation System**
**Step 1:**
- Parse the user login data into Tokens-`tok`;
- While (not empty of `tok`)
  - Check if `tok` ≠ reserved SQL Keyword
    - Move `tok` to User data Array-`UDA`;

**Step 2:**
- For (every data in `UDA`)
  - Convert to Corresponding MD5 and store in `MD5-UDA`.

**Step 3:**
- Parse the user input data into Tokens-`token`;
- While (not empty of `token`)
  - Check if `token` ≠ reserved JavaScript keyword
    - Set the `flag1` to continue;
  - Else
    - Set the `flag1` to deny;

**Step 4:**
- Extract the URL from HTTP;
- Parse the URL into Tokens-`toks`;
- While (not empty of `toks`)
  - Check if (URL = Benign using the signature check)
    - Set the `flag` to continue;
  - Else
    - Set the `flag` to deny;

**Step 5:**
- Send the `MD5-UDA`, `flag`, `flag1` and original Web request to Web application Server;

---

**Figure. 4.3: Data Cleansing Algorithm**

**XSL’s Pattern Matching:**

The query is first analyzed and tokenized as elements. The prototype document contains the query pertained to that particular application. For example the input query is,

```
SELECT * FROM members WHERE login='admin' AND
password='XYZ' OR '1=1'
```

When this query is received, this is converted into XML format using a XML schema. The resulting XML would be as shown in Figure. 4.4.
Using pattern matching the elements is searched such that the nested elements are similar to query tokens. The corresponding matching XML mapping is as shown in Figure. 4.5.

Figure. 4.5: Matching XML Mapping
When the match is found, the corresponding prototype query would be,

\[
\text{SELECT identifier FROM identifier WHERE identifier op 'userip' AND identifier op 'userip'}
\]

which will be used to identify the user input data. The extra XML tags other than those in the prototype will be considered as user input. This search is less time consuming because the search is based on text and string comparison. The time complexity is $O(n)$. This helps in increasing the effectiveness of the program and reduces the latency time. Similar to the query pattern, a script pattern contains all the valid patterns of script available in the application, to check for user posted data in the feedback form and the comment form.

Encrypting using MD5 hash and Signature check, Script check using regular expressions:

The user data extracted from the extraction phase is then encrypted using the MD5 hash function. All the possible forms of SQL injection manipulation are stored in the signature check in the form of regular expressions. The URL is extracted from the HTTP request and the URL is tokenized. These tokens are checked using the regular expressions. If they contain any form of the signature that has been defined as SQL injection, then the request is marked as malicious or else it is marked as benign. Similarly, The JavaScript keywords are stored in script check in the regular expressions form. The user input data extracted from webpage are tokenized and these tokens are then checked against the regular JavaScript keywords. If any JavaScript keyword is found in the process, then the input given by the user is marked malicious or else it is marked benign.

4.4 IMPLEMENTATION AND ANALYSIS

This system implements the DC algorithm in the automated sanitizing application using Java. 10 systems was used in the lab set-up connected through LAN. One system is considered as the web application server. Two
systems were set up as the proxy server, which has the automated sanitizing application installed. Seven systems act as the client. On the server, Eclipse Integrated Development Environment (IDE) runs the open source project. On the server gateway a filter program is installed. This filter application redirects the request from the user to the proxy server. For each request the server chooses one of the two proxy servers alternatively. This is done to minimize the loading on a particular proxy server, which might slow down the process.

In each of the proxy server, the sanitizing application is executed in the Eclipse IDE. When the redirected request from the server reaches the sanitizing application, the DC Algorithm is triggered. As a first step, the SQL query and the URL are extracted from the HTTP. The SQL query is processed using the XSL’s pattern matching and the prototype document. The user data are separated from the query. The URL is passed on to the signature check, which uses the regular expression to validate the URL.

The following signature checks are done on the URL’s extracted from the HTTP request.

1. Query delimiter ( --)
2. White Spaces
3. Comment delimiter (/* */) 
4. EXEC keyword
5. Unicode Transformation Format (UTF) coding
6. Scanning for query with signature OR followed same characters before and after ‘='.
7. Dropping meta characters (and their encoding) like ;,(,), >, <, %, +,= and @
8. Use of ‘IN’, ‘BETWEEN’ after ‘OR’.

9. Use of SQL keywords. Just looking into the keywords will bring about a lot of false positives. So the context before and after the keyword is also checked.

The following script checks are done on the user posted data:

1. <script>, </script>

2. Names of the event handlers (Eg. onClick, onLoad, onUnload, onChange, onFocus, onSubmit etc..)

3. Function


5. (,)

6. Javascript:

7. <, >

The user data are converted into their corresponding hash values using the MD5 algorithm. The hash values, the validated URL and validated user posted data are then directed back to the server.

Depending on the validation results the filter on the web application server decides whether to continue with the request or to deny the request. If the URL and the user posted input data are benign, then the URL, the original data and the hash value are forwarded to the web application on the server system. The user posted data are included in the original code. The web application sends the hash values to the database and the values are checked. If the values match, then the user gains access. Or else the request is denied. The database used by the web application is MySQL.
This system was tested on 4 open source projects. The open source projects that were considered for this study were taken from gotocode.com. The applications from this site have been used as a standard test bed in major peer works [20, 25, 30, 32, 33, 105 and 106]. The four projects that were taken for study were Online Bookstore, Online portal, Employee directory, and registration form. Burp suite [107] was used as an attacking tool. The inputs to these applications were the SQL injection commands from the cheat sheet developed by Halfond et al., [3]. This system was able to detect all the intrusions injected by burp suite and was able to achieve 100% detection rate. The total number of SQL injections by the Burp suite and the total number of detections by the proposed system defining the detection rate are stated in Table. 4.1.

<table>
<thead>
<tr>
<th>Web Application</th>
<th>Lines of code</th>
<th>No. of Attacks</th>
<th>No. of Detections</th>
<th>Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portal</td>
<td>10051</td>
<td>276</td>
<td>276</td>
<td>100%</td>
</tr>
<tr>
<td>Employee Directory</td>
<td>3526</td>
<td>238</td>
<td>238</td>
<td>100%</td>
</tr>
<tr>
<td>Book store</td>
<td>11078</td>
<td>197</td>
<td>197</td>
<td>100%</td>
</tr>
<tr>
<td>Registration Form</td>
<td>3702</td>
<td>419</td>
<td>419</td>
<td>100%</td>
</tr>
</tbody>
</table>

### 4.5 EVALUATION AND RESULT

The system was tested under 3 different scenarios such as light load condition, medium load condition, and heavy load condition. Under light and medium load conditions 5 and 50 requests respectively were sent from the client system to the server. For heavy load condition 1000 requests were sent from the client system. The time taken for the response, with and without the Intrusion Prevention proxy, was noted in nanoseconds. The results are shown in Figures. 4.6a, 4.6b, and 4.6c.
Figure. 4.6(a): Performance at Low Load

Figure 4.6(a) shows that the difference in response time for each application when executed for a light load of 5 requests, with and without the IP proxy, is negligible.
Figure 4.6(b) shows that the difference in response time for each application when executed for a medium load of 50 requests, with and without the IP proxy, is not significant.

Figure 4.6(c): Performance at High Load

Figure 4.6(c) shows that for each application, there is a 3.49% increase in response time, when executed for a high load of 1000 requests.

From the results it is observed that there was no significant difference in response time for light load and medium load conditions. For heavy load condition there was a slight increase, only in nanoseconds. The overall accuracy of the system is accounted to 100%. The experimental results have proved this trade-off to be a very minimal of 3.49% average increase in response time. This proposed system was compared with other standard methodologies that are used to curb SQLIA and XSS attacks. The detailed analysis is shown in Table 4.2.
Table. 4.2: Analysis of Methodologies used for Curbing Input Validation Attacks

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Change in source Code</th>
<th>Detection/Prevention of attack</th>
<th>Type of attack addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAVES[12]</td>
<td>Not necessary</td>
<td>Automatized/ report generated</td>
<td>SQLIA, XSS/Client side</td>
</tr>
<tr>
<td>AMNESIA[30]</td>
<td>Not necessary</td>
<td>Fully automatized</td>
<td>SQLIA</td>
</tr>
<tr>
<td>SQLGuard[19]</td>
<td>Necessary</td>
<td>Fully automatized</td>
<td>SQLIA</td>
</tr>
<tr>
<td>SQLCheck[20]</td>
<td>Necessary</td>
<td>Partially automatized</td>
<td>SQLIA</td>
</tr>
<tr>
<td>WebSSARI [14]</td>
<td>Necessary</td>
<td>Partially automatized</td>
<td>SQLIA, XSS</td>
</tr>
<tr>
<td>SQLRand[16]</td>
<td>Necessary</td>
<td>Fully automatized</td>
<td>SQLIA</td>
</tr>
<tr>
<td>SQL-IDS[22]</td>
<td>Not necessary</td>
<td>Fully automatized</td>
<td>SQLIA</td>
</tr>
<tr>
<td>Idea[18]</td>
<td>Not necessary</td>
<td>Only exposes vulnerabilities</td>
<td>SQLIA</td>
</tr>
<tr>
<td>SWAP[35]</td>
<td>Necessary</td>
<td>Fully automated</td>
<td>XSS/ Server side</td>
</tr>
<tr>
<td>Noxes[36]</td>
<td>Not necessary</td>
<td>Can be automatized</td>
<td>XSS/ Client side</td>
</tr>
<tr>
<td>Ulfar Erlingsson et al., [39]</td>
<td>Not necessary</td>
<td>Can be fully automatized</td>
<td>XSS/ Client side</td>
</tr>
<tr>
<td>Noncespaces[37]</td>
<td>Necessary</td>
<td>User intervention needed</td>
<td>XSS/ Client side</td>
</tr>
<tr>
<td>Proposed DC algorithm</td>
<td>Not necessary</td>
<td>Fully automated</td>
<td>SQLIA, XSS/ Server side</td>
</tr>
</tbody>
</table>
From the above table it is clear that this system is capable of providing a better and holistic solution when compared with the other standard models considered. Table 4.3 gives the comparison of results with the standard methodologies using the gotocode.com test suite to detect and mitigate SQL injection attacks.

Table 4.3: Analysis of Results for Standard Methodologies

<table>
<thead>
<tr>
<th>S.no</th>
<th>Methodologies</th>
<th>Detection rate</th>
<th>Performance overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CANDID [25]</td>
<td>100%</td>
<td>22.9%</td>
</tr>
<tr>
<td>2</td>
<td>Removing SQL query attribute values [32]</td>
<td>100%</td>
<td>Not available</td>
</tr>
<tr>
<td>3</td>
<td>SQLCheck [20]</td>
<td>100%</td>
<td>3 milliseconds</td>
</tr>
<tr>
<td>4</td>
<td>AMNESIA [30]</td>
<td>100%</td>
<td>10 to 40 milliseconds</td>
</tr>
<tr>
<td>5</td>
<td>Sania [105]</td>
<td>99%</td>
<td>Not available</td>
</tr>
<tr>
<td>6</td>
<td>SDriver [33]</td>
<td>100%</td>
<td>3.7%</td>
</tr>
<tr>
<td>7</td>
<td>Positive Tainting and Syntax-aware Evaluation [106]</td>
<td>100%</td>
<td>6%</td>
</tr>
<tr>
<td>8</td>
<td>Proposed system</td>
<td>100%</td>
<td>3.49%</td>
</tr>
</tbody>
</table>

Table 4.3 shows that the proposed system has 100% detection rate with a better performance rate compared with the other methodologies.
Figure 4.7: Comparative Performance Overhead

Figure 4.7 shows that the performance overhead for the proposed system is less than that of the other techniques using the same test bed.

Change in source code of the application is not needed for the mitigation and the system is fully automated. As the system provides a server side solution, user intervention is not needed. By increasing the number of proxy servers, the Web application can handle any number of requests without obvious delay in time and protect the application from SQL injection and XSS attack. The queries used in the web application are retrieved and preprocessed, which makes the comparison process simplified besides minimizing the error probability. This system will detect and prevent SQL Injection attacks that are based on tautologies, union queries, and piggy backed queries. Moreover, the complexity of the algorithm is \( O(1) \), in contrast to the other techniques such as AMNESIA, SQLCheck, SQLguard that use parse tree to statically and dynamically detect attacks, and whose complexity is \( O(n^3) \).