ISSN:0975-9646

Intrusion Protection against SQL Injection And Cross Site Scripting Attacks Using a Reverse Proxy

Tanmay S. Mule , Aakash S. Mahajan, Sangharatna Kamble, Omkar Khatavkar

Dept. of Computer Engineering, ISB&M School of Technology,Nande, Pune, India.

Abstract— SQL Injection attacks and Cross-Site Scripting attacks are the two most common attacks on web application. Proposed method is a new policy based Proxy Agent, which classifies the request as a scripted request, or query based request, and then, detects the respective type of attack, if any in the request. This method detects both SQL injection attack as well as the Cross-Site Scripting attacks.

SQL injection vulnerabilities have been described as one of the most serious threats to the database driven applications. Web applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. A SQL Injection Attack usually starts with identifying weaknesses in the applications where unchecked users' input is transformed into database queries.

Reverse Proxy is a technique which is used to sanitize the user's inputs that may transform into a database attack. In this technique a filter program redirects the user's input to the proxy server before it is sent to the application server. At the proxy server, data cleaning algorithm is triggered using a sanitizing application.

***Keywords*— SQL Injection, SQL Attack, Data Sanitization, Database Security, Security Threats, Cross Site Scripting.**

I. INTRODUCTION

In this era where internet has captured the world, level of security that this internet provides has not grown as fast as the internet application. Internet has eased the life of human in numerous ways, but the drawbacks like the intrusions that are attached with the internet applications sustains the growth of these applications. One such intrusion is the SQL Injection Attacks (SQLIA). Since SQLIA contributes 25% of the total internet attacks, much research is being carried out in this area. [2]

Now-a-days application-level vulnerabilities have been exploited with serious consequences: E-commerce sites are tricked by attackers and they lead into shipping goods for no charge, usernames and passwords have been cracked, and confidential and important credentials of users have been leaked. SQL Injection attacks and Cross-Site Scripting attacks are the two most common attacks on web application. [1]

A. Present System in Use:

The glory of internet and its merits are being highly masked by the drawback associated with it. Of them the prime issue is internet vulnerability, leading to data modification and data thefts. Many web applications store the data in the data base and retrieve and update information as needed.

B. Flaws in Current System:

Internet is a widespread information infrastructure and an insecure channel for exchanging information. Web application security relies on the ability to inspect HTTP packets to handle threats at Layer-7 of the OSI model. Attackers are all too familiar with the fact that traditional perimeter security methods do not stop attacks against Web applications that are, by nature, designed to allow visitors to access data that drives the Website. By exploiting simple vulnerabilities in Web applications, an attacker can pass through the perimeter security even when the traditional firewall and IDS systems are in place to protect the application. Web applications contain rich content to be transferred from web application to the server site, which makes the website vulnerable to various types of code injection attacks. Injection attacks are the result of a Web application sending untrusted data to the server. [3]

The most common attack occurs from malicious code being inserted into a string which is sent to the SQL Server for execution. This attack, known as SQL Injection, allows the attacker to access data from the database, which can be stolen or manipulated. Cross-Site Scripting, or XSS, is another prevailing security flaw that Web applications are vulnerable to. In an XSS attack, the attacker is able to insert malicious code into a website. When this code is executed in a visitor's browser it can manipulate the browser to do whatever it wants. Typical attacks include installing malware, hijacking a user's session, or redirecting users to another site. [1]

II. BACKGROUND STUDY

Code Injection is a type of attack in a web application, in which the attackers inject or provide some malicious code in the input data field to gain unauthorized and unlimited access, or to steal credentials from the users account. The injected malicious code executes as a part of the application. This results in either damage to the database, or an undesirable operation on the internet. Attacks can be performed within software, web application etc, which is vulnerable to such type of injection attacks. Vulnerability is a kind of lacuna or weakness in the application which can be easily exploited by attackers to gain unintended access to the data [2]. Some common code injection attacks are HTTP Request Splitting Attacks, SQL Injection Attacks, HTML Injection Attacks, Cross-Site Scripting, Spoofing, DNS Poisoning etc.

III. SYSTEM ARCHITECTURE

The architecture of the system is illustrated in Figure 1. 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 sanitizing application is placed in the Reverse proxy server.

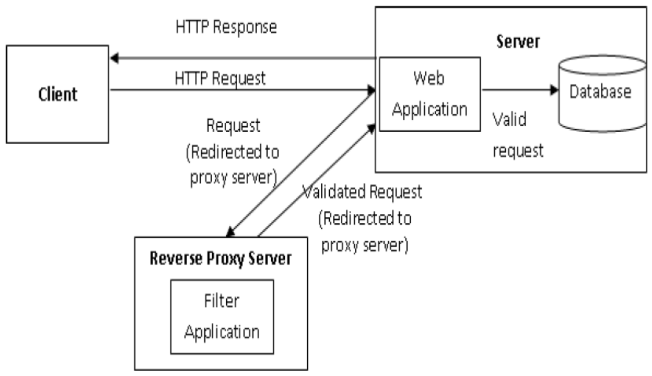


Fig. 1 Conceptual Architecture

A reverse proxy is used to sanitize the request from the user. When the request becomes high, more reverse proxy's can be used to handle the request. This enables the system to maintain a low response time, even at high load.

The general work of the system is 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 URL from the HTTP and the user data from the SQL statement.
   a. The URL is send to the signature check
   b. The user data (Using prototype query model) is encrypted using the MD5 hash.
4. The sanitizing application sends the validated URL and hashed user data 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.
6. If the URL is found to be benign, then the hashed value is send to the database of the web application.
7. If the hashed user data matches the stored hash value in the database, then the data is retrieved and the user gains access to the account.
8. Else the user is denied access. Figure 2 gives the flowchart of the system.

A. *Injection Detector*

A Query Detector is a simple tool which is used to test the precision of SQL Queries, and detecting malicious request from user at the web server. It takes request coming from any user and validates the request before forwarding it to the web server for further execution and processing.

1) *Session Manager*

When HTTP request goes to the web server a Session object for that user is initialized [25], which assign a Session variable or Token for that particular connection. This session remains in its active state until the connection remains active. As soon as the connection is terminated the session terminates accordingly.

2) *Input Valuator*

Input Valuator is a key section of Query detector. It works as a Proxy between Client and the web server and any request going on the web server is first validated at the Input_Valuator. It has an attack vector repository consisting of some special characters (e.g. ' - ;) which are often used in writing malicious code for SQL Injection attack. It does the functionality of matching user supplied data in HTTP request with the text file stored in attack repository. When user supplied text contain any special symbols which are present in the repository, it is treated as invalid request by the Input_Valuator. Execution of that request on the web server is prevented. If no pattern is matched then that request is treated as valid and is forwarded to the next module for filtering the script tags.

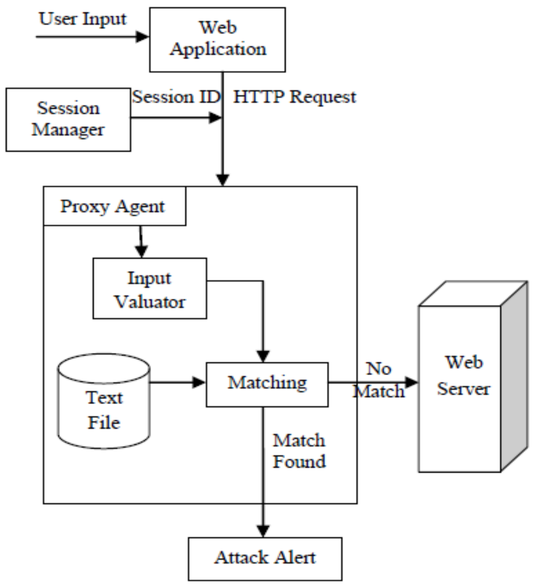


Fig. 2 SQL Injection Detector

B. *Script Detector*

Script detector is used to detect the malicious script embedded in the web application. It sanitizes HTML input before executing on the web server. This sanitization process removes all the invalid and unwanted tags from the user input and then encodes the remaining input into simple text thus preventing the execution of any malicious script. The block diagram of Script Detector shown in figure 4.3 has different blocks which prevent the Cross-Site Scripting attack.

1) *HTML Sanitizer*

HTML Sanitizer removes unsafe tags and attributes from HTML code. . It takes a string with HTML code and strips all the tags that do not make part of a list of safe tags. The list of safe tags is defined according to the whitelist tags list given by Open Web Application Security Project (OWASP) [20]. There are some functions to dis-allow unsafe or forbidden tags like script, style, object, embed, etc. It can also remove unsafe tag attributes, such as those that define JavaScript code to handle events. The links href attributes also gets special treatment to remove URLs that trigger JavaScript code execution and line breaks. The list of all the allowed tags and forbidden tags is given in Table 2. The sanitization process starts with breaking the HTML string in tokens; this functionality is handled by HTML tokenizers.

2) Tokenizer

Tokenizer divides the HTML text within user input into tokens. A token is a single atomic unit of supplied text. In proposed method a token is be one of the following: tag start (), comment (), tag content (“text”), a tag closing (). As a result of this a list of tokens will be created, and then each and every token in this list is matched with the whitelist tags and forbidden tags shown in Table 2. And then the HTML Sanitizer forward’s the user request to HTML Encoder.

3) HTML Encoder

HTML encoder performs the character escaping. It uses the HtmlEncode Method of ASP.NET to encode the user input. The HtmlEncode method applies HTML encoding to a string to prevent a special character to be interpreted as an HTML tag. This method is useful for displaying text that contain "special" HTML characters such as quotes, angular brackets and other characters by the HTML language. Table 1 show a list of some of these special characters and their equivalent encoded value, which is used by the HTML Encoder to encode the input.

4) Script Pattern

This contains all the tags and patterns that are used to match with the tokens which are formed by the tokenizer. It contains list of all the forbidden tags, allowed tags, tag starting pattern, tag closing pattern, comment patterns, style pattern, URLpattern etc. The list of all patterns used by this module is shown in Table 2.

5) Pattern Matcher

The functionality of this module is just to take the input from the list of tokens and match them with the Script Patterns. All the rejected tags are stored in the invalid tags list and all the accepted tags are forwarded to the HTML Encoder for encoding.

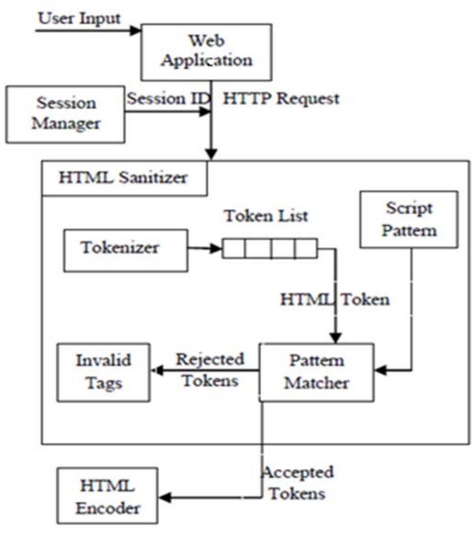


Fig .3 Cross Site Scripting

IV. System Implementation

We implemented the prototype version of CIDT as a Windows .NET application in C#. We choose .NET because in the literature survey we found that all the categories of code injection attacks were not succeeded on the application build using Java. The web applications on which attacks are performed and tested is implemented using simple web technologies like HTML, CSS, and Active Server Pages. Query Detector and Script Detector are implemented separately and then they are combined together to form a Code Injection Detection Tool. Algorithm 4.1 and Algorithm 4.2 are used for implementing the modules.

A web application having login page, a text file containing some special characters and a database to store the user’s login information is required for implementing Algorithm 4.1. It is used for preventing user from SQL Injection attack.

Algorithm 4.1 Query Detector

```
\begin {SQL_Detect}
Step 1: Accept u_name, u_pass in text from users.
Step 2: Start the Session for current u_name.
Step 3: Forward u_name to FileInput.aspx.
Step 4: Set attack ← False;
Step 5: Repeat <for each line of input>
Until { (line equal to Test.txt) and not equal to Null }
\End While
Step 6: Set line ← String Pattern;
Step 7: If { u_name.contains(line)}
Set attack ← true;
\End If
Step 8: If { attack equals to true }
Set Valid ← false;
Else
Set Valid ← true;
\End If
Step 9: If { Valid is equal to false }
Discard U_name from entering into the database.
Else
Allow Connection to database.
\End
```

User’s request through a web application is forwarded to the Query Detector. Algorithm 4.1 then matches the content of user request with the text file for any special character. If any special character gets matched, the request is said to an invalid request and its execution is stopped. Otherwise it is allowed to be executed.

Algorithm 4.2 Script Detector

```
Step 1: Take user input in the form of any HTML text
having scripts, tags, links, or urls.
Step 2: Tokenize the input code.
Step 3: Store all the tokens in a list.
Step 4: Having the list of token, check for every single
token whether it is acceptable or not.
Repeat {for every token check it with a regular
expressions}
```

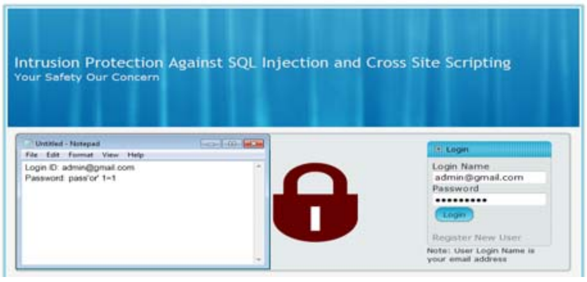
```
a) If token is a comment discard it.
b) If { token is a start tag }
Extract the tags and all its attributes
If { Forbidden Tag }
Remove the tag.
\End if
If { Allowed Tag } then do
Extract every attribute of the tag.
i) Check the "href" and "src" for admitted tags.(a, img, embed )
ii) Check the "style" attribute and discard it.
iii) Remove every "on….." attribute (onclick,onmouseover…)
iv) Encode attribute value for unknown ones.
v) Push the tag on the stack of open tags.

Else
The tag is unknown and will be removed.
\End If
If { token is a end tag } then do
Extract the tag
Check whether the corresponding tag is already open.
Else
It is not a tag encode it.
\End If
\End While
```

Algorithm 4.2 describes the process of sanitizatior Sanitization is a process of filtering html content preser in the input request. The function of sanitizer is t tokenize the user request and collects the list of tokens Each token is matched with the script pattern usin regular expressions. Unwanted or invalid tokens ar removed from the user request and then the systen encodes it before forwarding to the web server.

V. Evaluation

This system was tested on 4 open source projects. The open source projects that was considered for this study, was taken from *gotocode.com.* The four projects that were taken into study were Online Bookstore, Online portal, Employee directory, registration form. We used Burp suite [25] as an attacking tool. Our 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 our system defining the detection rate is stated in Table 1. Figure 3 and Figure 4 shows the response of the system when a malicious input is provided in the input form.


Intrusion Protection Against SQL Injection and Cross Site Scripting
Your Safety Our Concern
Untitled - Notepad
File Edit Format View Help
Login
Login
Register New User

Login ID: admin@gmail.com
Password: pass'or' 1=1

Login Name: admin@gmail.com
Password: •••••••••

Note: User Login Name is your email address

Fig. 4 Malicious input provided to the Application.

TABLE I DETECTION RATE

Web Application	No. of SQL Injection Attacks	No. of Detections	Detection Rate
Portal	276	276	100%
Employee Directory	238	238	100%
Book store	197	197	100%
Registration Form	419	419	100%

VI. Analysis And Result

We have analyzed our system and other methodologies that are used to curb SQLIA. The detailed analysis is shown in Table 2. The system was run under light load condition, medium load condition and heavy load condition. The time taken for the response with our system's Intrusion Prevention proxy (IP proxy) and without the Intrusion Prevention proxy was noted in Nanoseconds. Under Light load condition 5 requests from client system was send to the server. Low load

Under medium load 50 requests was send from client system using threads. For heavy load 1000 requests was send using client system. The time taken did not show much difference for light load and medium load condition. For heavy load condition, there was a slight difference in nanoseconds.

TABLE II ANALYSIS OF METHODOLOGIES CURBING SQLIA

Methodology	Change in source Code	Detection/Mitigation of attack
WAVES[4]	Not necessary	Automatized/ report generated
JDBC-Checker[5]	Needed for automatic prevention of attack.	Can be automatized.
AMNESIA[6]	Not necessary	Fully automatized
SQLGuard[7]	Necessary	Fully automatized
SQLCheck[8]	Necessary	Partially automatized
WebSSARI[9]	Necessary	Partially Automatized
Livshits and Lam[10]	Not necessary	Manual assistance needed
Security Gateway[11]	Not needed	Manual detection / automatized Mitigation
SQLRand[12]	Necessary	Fully automatized
SQL-IDS	Not necessary	Fully Automatized
Idea	Not necessary	Only detection of attacks
COMPVAL	Not necessary	Fully automated
Proposed DC algorithm	Not necessary	Fully automated

The system using the proxy server protection was responding a little slower than the other system, but had full protection against SQL injection attacks. If we increase the number of proxy server to four then the server was able to handle the request with an increased pace. We have not yet worked on optimization of the system. We believe, after optimization of the system, the performance will improve.

VII. CONCLUSIONS

The novel system with intrusion prevention proxy has proved to be effective in detecting the SQL injection attacks and cress site scripting attacks and preventing the attacks from penetrating the web application. This system does not do any changes in the source code of the application. The detection and mitigation of the attack is fully automated. By increasing the number of proxy servers the web application can handle any number of requests without obvious delay in time and still can protect the application from SQL injection attack. In future work, the focus will be on optimization of the system and removing the vulnerable points in the application itself, in addition to detection and studying alternate techniques for detection and mitigation of SQL injection attacks and cross site scripting attacks.

REFERENCES

[1] David Litchfield, (2005) "Data-mining with SQL Injection and Inference", Next Generation Security software Ltd., White Paper.

[2] Allaire Security Bulletin, (1999) "Multiple SQL statements in dynamic queries".

[3] Chip Andrews, "SQL Injection FAQs", http://www.sqlsecurity.com/FAQs/SQLInjectionFAQ/tabid/56/Default.aspx

[4] Y.Huang, F. Huang, T.Lin and C.Tsai, (2003) "Web Application Security Assessment by Fault Injection and Behavior Monitoring", Proc. International World Wide Web Conference '03, pp. 148 -

[5] C.Gould, Z.Su and P.Devanbu, (2004) "JDBC Checker: A Static Analysis Tool for SQL/JDBC Application", Proc. International Conference on Software Engineering '04, pp.697-698.

[6] W. G. Halfond and A. Orso, (2005) "AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks", Proc. ACM International Conference on Automated Software Engineering '05, pp. 174-183.

[7] Gregory Buehrer, Bruce W. Weide and Paolo A. G. Sivilotti, (2005) "Using Parse Tree Validation to Prevent SQL Injection Attacks", Proc. International Workshop on Software Engineering and Middleware, pp. 106-113.

[8] Zhendong Su and Gary Wassermann, (2006) "The Essence of Command Injection Attacks in Web Applications", Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages '06, pp.372-382.

[9] Y.Huang, F.Yu, C. Hang,C.H. Tsai, D.T.Lee and S.Y.Kuo, (2004) "Securing Web Application Code by Static Analysis and Runtime Protection", Proc. International World Wide Web Conference '04, pp. 40-52.

[10] V.B. Livshits and M.S. Lam, (2005) "Finding Security Errors in Java Programs with Static Analysis", Proc. Usenix Security Symposium '05, pp. 271-286.

[11] D.Scott and R.Sharps, (2002) "Abstracting Application-level Web Security", Proc. International Conference on the World Wide Web '02, pp. 396-407.

[12] S.W. Boyd and A.D. Keromytis, (2004) "SQLrand: Preventing SQL Injection Attacks", Proc. 2nd Applied Cryptography and Network Security (ACNS) Conference, pp. 292-302.