Detecting and Defeating SQL Injection Attacks

Sangita Roy, Avinash Kumar Singh and Ashok Singh Sairam, *senior member IACSIT*

***Abstract*—The increasing dependence on web applications have made them a natural target for attackers. Among these attacks SQL Injection Attacks (SQLIA) are the most prevalent. In this paper we propose a SQL injection vulnerability scanner that is light-weight, fast and has a low false positive rate. These scanners prove as a practical tool to discover the vulnerabilities in a web application as well as to test the efficiency of counter attack mechanisms. In the latter part of our work we propose a security mechanism to counter SQL Injection Attacks. Our security methodology is based on the design of a filter for the HTTP request send by clients or users and look for attack signatures. The proposed filter is generic in the sense that it can be used with any web application. Finally we test our proposed security mechanism using the vulnerability scanner developed by us as well as other well known scanners. The proposed security mechanism is able to counter all the vulnerabilities that were previously reported before the deployment of our security framework.**

***Index Terms*—SQL Injection Attacks, URL filter, Web Application Vulnerability Scanner.**

I. Introduction

Web applications such as blogs, social network, webmail, bank etc have become our way of life. The omnipresence of web applications has made them a natural target for malicious minds. Web applications are susceptible to a number of vulnerabilities which can be due to a design flaw or an implementation bug. Among the top ten web application vulnerabilities published by Open Web Application Security Project [10], SQL Injection Attack (SQLIA) is the most vulnerable. According to OWASP, SQL injection vulnerabilities were reported in 2008, making up 25% of all reported vulnerabilities for web applications. An SQLIA occurs when an attacker changes the intended effect of an SQL query by inserting (or injecting) new SQL keywords or operators into the query thereby gaining unauthorized access to a database in order to view or manipulate restricted data. The root cause of SQLIA is insufficient user input validation. Although there is an increasing awareness about security, there are several significant factors that make securing web applications difficult. First web applications are growing at a frantic pace largely fuelled by the simplicity with which one can develop such applications using the numerous tools available. Secondly the developers and administrators do not have the requisite knowledge and experience in the area of security.

A logical approach to tackle the problem of SQLIA is to scan the vulnerabilities present in a webpage and subsequently launch attack counter measure tools. There are a number of open-source as well as commercial tools called Web Application Vulnerability Scanners [7, 8, 9] that perform security testing as well as assessment and finally report the vulnerabilities present. In spite of their continuous evolution,these automated scanners still have some problem with regard to the high number of undetected vulnerabilities and high percentage of false positives [2]. A web vulnerability scanner is not a panacea but an useful tool to access the security of web applications. The methodology proposed in this paper is to first scan a webpage in a controlled environment and discover the vulnerabilities present. Next we provide a framework to prevent SQLIA. To test the performance of our security framework we again run the vulnerability scanner after its implementation. Empirical results on a realistic environment show that our counter security mechanism is able to prevent all the vulnerabilities previously reported by the scanner.

The remainder of the paper is organized as follows. In section II we study existing scanners and review their performance. Section III presents overview of our proposed web vulnerability scanner named CSRScanner and section IV describes the implementation details of CSRScanner and in section V we have done the analysis of our designed tool. In section VI we perform a detailed analysis of the different types of known SQLI attacks and identify a unique pattern or signature for each. Based on these signatures we propose a URL filter to prevent SQLI attacks in section VII. In Section VIII we have checked the effectiveness of our filter approach by using our CSRScanner tool. Finally we conclude the paper in section IX.

II. Web Application Vulnerability Scanners

Web application vulnerability scanners are designed to test security mechanisms applied to web applications [4]. The general methodology of these scanners is based on the discovery of *vulnerable spots* that is positions in the HTTP request where an attacker can inject maliciously crafted SQL codes. In the second step the tool performs a controlled exploit of the vulnerabilities at these vulnerable spots. Finally it verifies the success of the attack and reports the result. The vulnerability scanners are designed such that they perform the same attack as one would do manually and hence they provide a practical environment to test counter measure mechanisms against SQLIA.

A host of vulnerability scanners both commercial as well as open-source are available. A brief review of some of the best known scanners is given below.

- Acunetix([8]): automatically checks web applications for vulnerabilities such as SQL Injections, cross site scripting, arbitrary file creation/deletion and weak password strength

Manuscript received June 16, 2011; revised June 30, 2011.

Sangita Roy is with the Indian Institute of Technology, Patna, Bihar, India (e-mail: r_sangita@ iitp.ac.in).

Avinash Kumar Singh is with the Gwalior Engineering College, Gwalior, MP , India(e-mail: avinashkumarsingh1986@ gmail.com).

Ashok Singh Sairam is with the Indian Institute of Technology, Patna, Bihar, India (e-mail: ashok@iitp.ac.in).

on authentication pages. AcuSensor technology detects vulnerabilities which typical black box scanners miss. Acunetix WVS boasts a comfortable GUI, an ability to create professional security audit and compliance reports, and tools for advanced manual web application testing.

- Netsparker([7]): can find and report security issues such as SQL Injection and Cross-site Scripting (XSS) in all web applications regardless of the platform and the technology they are built on. Netsparker's unique detection and exploitation techniques allow it to be dead accurate in reporting.
- WebCruiser([9]): has a vulnerability scanner as well as a series of security tools. It can support scanning website as well as POC (Proof of Concept) for web vulnerabilities: SQL Injection, Cross Site Scripting, XPath Injection etc. So, WebCruiser is also an automatic SQL injection tool, an XPath injection tool, and a Cross Site Scripting tool!

In order to test the performance of these scanners in a realistic environment we studied web applications available on the Internet and tried to re-create these applications locally. We also directly ran these scanners on some popular web application available online. The results of our study are briefly outlined below, more details regarding the experimental setup and results are given in section V.

- Existing scanners try to discover vulnerable spots by analyzing SQL queries issued in response to HTTP requests. Thus these tools need to analyze the SQL queries for all possible HTTP request hence the overheads are very high.
- Existing automated tools try to discover SQL injection vulnerabilities by indiscriminately applying all possible malicious code to every vulnerable spot. Hence these tools usually take very long time to scan even moderate-sized websites. For example Acunetix took 4 hours to scan a web application that had 100 pages.
- Existing scanners examine the contents returned by a web application to determine whether an SQL injection attack was successful. They conclude an attack as successful if the application returns a response different from a previously known innocent one, although the SQL injection has actually failed. Hence the false positive rate of these scanners is very high.

The observations from our study motivated us to design a light-weight, fast scanner with low false positive rate. The methodology of our proposed scanner is given in the following section.

III. CSRSCANNER

In this section we present our proposed SQL Injection scanner. The tool is designed to be used by a web application developer. In order to check a web application it needs the index or the home page of the web application as input. The tool *crawls* through the entire page of the application just like a web spider, *scans* each page for vulnerable spots, injects SQL code to these vulnerable points and *reports* success or failure of the injection. CRSScanner checks for SQL injection vulnerabilities in three steps.

A. Crawling the whole web application :

For finding the input points we first explore the whole web application. In order to examine the entire web application it is designed in the form of a tree. Figure 1 shows the tree structure of web application where a.php is the home or index page and the other pages are child nodes. After construction of the tree the pages are visited using depth first search. A variable *STATUS* is used to indicate whether a page has been visited or not.

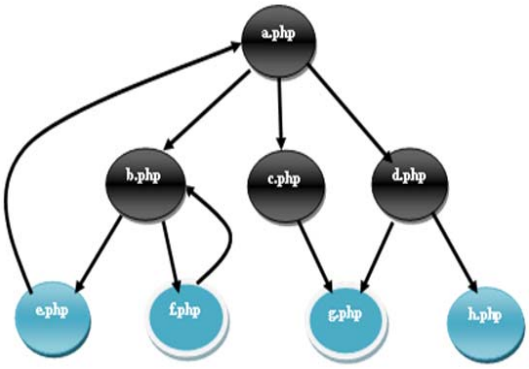

Fig.1. Tree structure of web application

The problem with the existing scanners is that they cannot provide full coverage i.e. they cannot visit all pages. That is why the coverage ratio of the commercial tools is low. Our tool provides full coverage of any web application.

B. Scanning for vulnerable points

The tool examines the URL of each visited and tries to identify the input points. If the page accepts user inputs then it is tagged as a vulnerable point. For example, if we get an URL like http://xyz.ac.in/departments/cse/csecourses.html then we can say that, in this page we do not have any vulnerable points. But if the URL is like http://xyz.site.com/product.php?product_id=10 then we can say product_id takes part in generation of a SQL query which may be of the form:

*SELECT * FROM product WHERE product_id=10.* In this query product_id is the parameter and value is 10. The parameter element is always fixed but an attacker can freely alter the value element. Thus this URL has a vulnerable point.

C. Generate Attack and Report

The final step is to inject the attack codes at the vulnerable spots identified in the last step and report the outcome of the attack. Attack generation consist of two parts: (i) Payload Setup and (ii) Generating Attack.

- Payload Setup: In this phase the attack payload is created based on the prevalent SQLI attacks. For generating the payload we created a list of the common SQL Injections which are used by attackers. The response of an attack will differ depending on the underlying database. In table I we list the response of some of the best known database.

Table II gives a partial list of the attack payload types and their explanation

- Generating Attack : In this phase we generate the attack by concatenating the attack payload with the original

TABLE I: TYPES OF DATABASE ERRORS

Sr. No	Database Errors
1	You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax
2	ADODB.Command error
3	[Microsoft][ODBC SQL Server Driver][SQL Server]Incorrect syntax near
4	Error converting data type nvarchar to bigint
5	Server Error in "/" Application
6	Error in subcategoryUnknown column '10000' in 'order clause'

query URL of the web application, and make request of this specially crafted attack URL e.g., suppose we have web page with the URL :

http://www.xyz.com/level1.php?cate_id=2

here cate_id=2 is a query string of the URL and cate_id is the input for this, which means that this level1.php page will display the records from the database which has the value 2. The query will be represented like this:

*SELECT * FROM user WHERE cate_id=2;*

Now the attacker has fired an SQLI attack and the URL is now http://www.xyz.com/level1.php?cate_id=2 order by 20 --

The corresponding SQL query will be

*SELECT * FROM user WHERE cate_id=2 order by 20;*

If the URL is not sanitized then the entire query will generate a database error i.e. " Error in subcategory Unknown column '20' in 'order clause' " which implies that there is no 20th number of column in the backend database.

TABLE II: ATTACK PAYLOAD DETAILS

Sr. No	Attack	Attack Explanation
1	'	Forcibly termination of query
2	Order by	Will count how many columns are there in that table
3	Select user, password from mysql.user	Checking user has root privileges or not

By sending different specially crafted attack request the proposed scanner checks if SQL injection vulnerabilities lie in a web application or not. For checking vulnerability we have defined a database error table in which we have stored all the database error related to different injection attack. Our tool uses the content analysis technique for finding the database error. We generate the attack request in such a way that, if the web pages are not sanitized the query URL will always provide database syntax error. After putting the attack request our tool automatically checks the response if there exist any database error or not. This database error is already mentioned in our database. If any database error is found in the content of the response page then we can say that vulnerability exists in the input point of this page. We are also testing the single input points with several injection attacks and then we assure that this point is vulnerable.

The steps performed by the CSR scanner can be summarized as follows:

Step1: Create a FIFO queue with two fields URL (Primary Key), STATUS.
Step2: Insert the target URL and set its STATUS=0.
Step3: Update STATUS=1
Send Request for the URL
Analyze the response, Extract its entire links.
Insert these links in FIFO Queue and set STATUS=0.
Step4: Go To Step3 While STATUS=0 ELSE Go To Step 5.
Step5: Finish.

IV. CSRSCANNER IMPLEMENTATION DETAILS

A. Implementation of Crawling and Scanning

For scanning the whole web application we have created two functions, one is *SeedUrl* which provides the URLs to the *Crawling* function which have the STATUS=0, *Crawling* function takes the input URL from *SeedUrl,* updates STATUS to 1 downloads the URL, writes the response to a file, extract all URLs present in the page and inserts it into a database with STATUS set as 0. The process will repeat till there are no URLs with STATUS=0. The JAVA code snippet is been given below.

```
//SeedUrl Starts here
Void SeedUrl(){
Boolea flag=true;
ResultSet rs=stmt.executeQuery("select distinct
status,url from '"+TableName+"'
where status=0");
While(rs.next){
String str=rs.getString("url");//get the url from the
database
Crawling(str);//send str to the main crawler
Flag=false;}
If (flag==false)
SeedUrl()
Else
System.out.print("Crawling Complete");
}//SeedUrl() ends here
Crawling(String str){
Update status=1 where url=str;
URL u = new URL(str);
URLConnection uc = u.openConnection(); //Send
request for the str to
the application server
FileWriter fw=new FileWriter("URL.txt",false);
//Create a file
while ((ct = r.read( )) != -1)
fw.write((char) ct); //Write the response into a file
String regex="href";
Pattern pattern =
Pattern.compile(regex,Pattern.CASE INSENSITIVE
—Pattern.MULTILINE);
//Finding and extracting all the href attributes
qry="insert into url values(0,"+""""+href url+""""+")";
stmt.executeUpdate(qry);
//Insert them into the database and set status=0
}//crawling() ends here
```

B. Attack Implementation

Attacks are implemented by drawing SQL Injection attack codes from a database and then adding those to the URLs

obtained in the previous step. The URL with SQL code injected is send as a request to the web application. The response is written to a file and analyzed for SQLI pattern. To analyze the response we created a database that contains SQLI pattern and error response of different well known databases. Finally if response matches SQLI pattern the tool reports the vulnerability along with which parameter is vulnerable and its cause.

```
// drawn attack from attack library
ResultSet rs=stmt.executeQuery("SELECT * FROM
attack");
//Setting up the SQLI attack
while(rs.next)
{String attack url=url+rs.getString(lib);
URL u = new URL(attack url); //sending request
URLConnection uc = u.openConnection( );//open
conncetion
FileWriter fw=new
FileWriter("Attack.txt",false);//define attack file
while ((ct = r.read( )) != -1)
fw.write((char) ct);//write response in file
//Select Database errors
ResultSet rs1=stmt1.executeQuery("select error
from db error");
while(rs1.next())
{String db error=rs1.getString("error");
Pattern pt = Pattern.compile
(db error,Pattern.CASE INSENSITIVE—
Pattern.MULTILINE);
Matcher mat = pt.matcher(stt);//define pattern for
each attack
while (mat.find())
{flag=true;
error=db error;}
If(flag==true)
{stmt1.executeUpdate("insert into "+table+"
values('"+urlss[0]+"',
'"+prm1.substring(1)+"','"+error+"','yes')");}
else {
stmt1.executeUpdate("insert into "+table+"
values('"+urlss[0]+"',
'"+prm1.substring(1)+"','"+error+"','no')");}
} //db error recordset closed
} //attack recordset closed
```

V. Analysis of CSRScanner

To measure the effectiveness of our penetration testing tool we designed five different types of web applications in the local host. We looked at some popular websites and tried to mimic their operation locally. The tool was also tested online in real on three public web applications. The name of these public sites has not been disclosed for security reasons. All these web applications were manually tested for vulnerability and the SQLI vulnerabilities present in each of them is shown in table III. The sum total of vulnerabilities present in all the web applications is twenty one.

In order to quantify the performance of our tool three other well known SQLI scanners were also tested on these web applications. Out of the total 21 the number of vulnerabilities each tool could detect, the average time taken and number of false positives is shown in table IV.

TABLE III: Vulnerability Details of Different Web Application

Web Application	Domain	Vulnerable page
Karnel Travel Guide	LocalHost	No
On line Real State	LocalHost	SQLI(1)
ICC World Cup	LocalHost	SQLI(1)
On line Tutorial	LocalHost	SQLI(1)
Mail server	LocalHost	SQLI(1)
Travel Site	Public	SQLI(1)
Educational Site	Public	SQLI(4)
Educational Site	Public	SQLI(12)
	Total Vulnerability	21

TABLE IV: Comparative Study of SQLI Scanners

Parameter	Acunetix	Netsparker	WebCruiser	CSRScanner
Vulnera-bility Detected	18/21	16/21	15/21	21/21
Average Time(hr)	2.24	1.2	0.15	0.07
False Positive	12	3	1	0

The result shows that none of the tools provides full coverage ratio and they also take much time to generate the report. Figure 2 shows the results for the execution of penetration testing tools and for the CSRScanner tool. As we can see, different tools reported different numbers of vulnerabilities. An important observation is that the CSRScanner is able to identify a much higher number of vulnerabilities than the remaining tools. In fact, all the penetration-testing tools detected less than 85% of the vulnerabilities, while our tool detected all vulnerabilities. Considering only the penetration testing tools, Acunetix identified the highest number of vulnerabilities (85% of the total vulnerabilities). However, it was also the scanner with the highest number of false positives (it detected 12 vulnerabilities that, in fact, do not exist). The lowest number of vulnerabilities was detected by WebCruiser with false positive is only 1. As different tools detect different sets of vulnerabilities an interesting analysis is how these sets intersect each other.

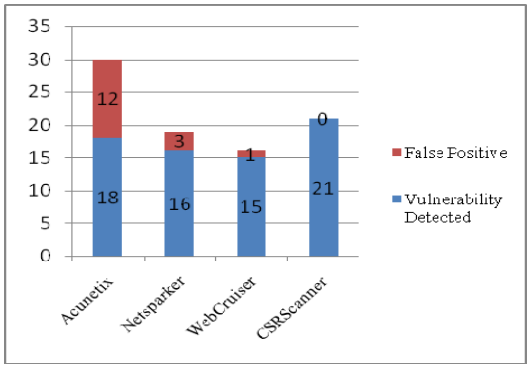

Fig. 2. Vulnerability detection graph

In Figure 3, each circle represents the vulnerabilities detected by a tool and each intersection area represents vulnerabilities found by more than one tool. The circles area is roughly proportional to the represented number, but the same does not happen with the intersection areas, as it would be impossible to represent it graphically. As we can see, there

are 21 vulnerabilities that are detected only by CSRScanner. We observe that Acunetix misses only 3. Although Netsparker had a lower coverage than Acunetux, it could detect 1 of the 3 vulnerabilities that was missed by the latter. WebCruiser could detect 15 vulnerabilities and all of these were detected by the other testing tools. Figure 5 gives a comparison of the time taken by each tool in the form of a bar chart.

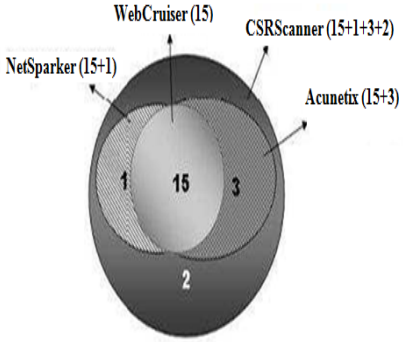

Fig. 3. Vulnerability Covered by each tool

CSRScanner takes very less time around 5.4 minute, which is magnificent among all tools. At the same time CSRScanner provides a good coverage without any false positive. As all the commercial tools are taking much more time for scanning the whole web page so our motivation was also to minimize the time span. To achieve this we have used multithread approach for time span minimization. Time taken by each tool has been depicted in figure 4.

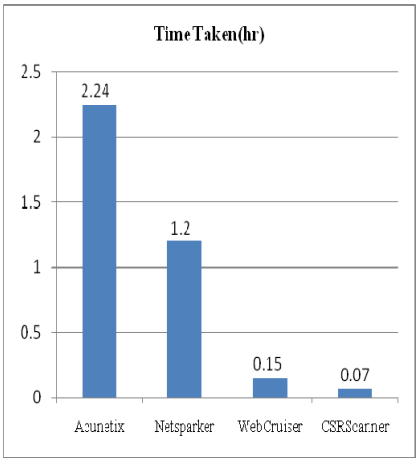

Fig. 4. Time taken by each tool

VI. SQL Injection Attack Types

In figure 5 we show 3-tier web application architecture [5]. Typically a *Client* triggers a request or query which is sent in the form of an *URL* to the *Application Server*. The *Application Server* parses the *URL* and fires a corresponding *SQL query or statement* to the *Database Server*. The *Database Server* processes the *SQL statement* and sends its response back to the *Application Server* which finally gets displayed at the *Client's* browser. An SQL injection attack occurs when an attacker changes the intended effect of an SQL query by inserting (or injecting) new SQL keywords or operators into the *URL* which is sent to the *Application Server* in the first step.

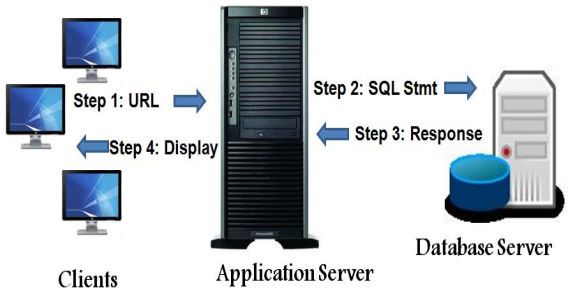

Fig. 5. 3-Tier Web application Architecture

As an example of a SQLIA consider an *Application Server* that executes the Java script code excerpt given in figure 6. The code runs as a server-side script allowing users to read employee information stored in a MySQL database table named *EmpTable*. To use the code in general, a *Client* will fill up an HTML form and the information communicated to the *Application Server* via a *URL* will typically appear as "*http://localhost/?EmpId=10*". This input will cause to display information of the employee with employee id 10 (if the record exists) which is stored in the table. Consider malicious user posses as a *Client* and passes the URL "*http://localhost/? EmpId=10'; DROP TABLE EmpTable--*" to the *Application Server*. This input will lead to an SQLIA because the query the server submits to the database (line 6 of figure 6) will appear as: *SELECT empinfo FROM EmpTable WHERE EmpID = '10'; DROP TABLE EmpTable--*.

This will cause the database to delete the table EmpTable. Such attacks which exploit the lack of validation in the input parameter are known as first-order attack.

In a second-order attack, the attacker injects malicious code into strings that are destined for storage in a table or as metadata. When the stored strings are subsequently concatenated into a dynamic SQL command, the malicious code gets executed.

```
1. // Get input from user
2. var EmpId;
3. EmpId = Request.form("EmpId");
4. // Construct SQL statement and execute
5. var sqlstmt = "SELECT empinfo FROM EmpTable
   WHERE EmpID = ' " +EmpId+" '";
6. $result = mysql_query(sqlstmt);
7. //Display the result
8. while ($row = mysql_fetch_assoc($result)){
9. echo $row['empinfo'];
```

Fig. 6. PHP code excerpt to display employee information.

Sanitizing is a technique used to prevent SQLIA by escaping potentially harmful characters in *Client* request messages. Suppose a web application uses the following code to authenticate a user:

*"SELECT * FROM Users WHERE id = ' " + Request.form(UserId) + " 'AND passwd = ' " + Request.form(Password) + " ' ";*

If an attacker inputs "xyz" and " ' or '1' = '1 " in the UserId and Password field respectively the query becomes

*SELECT * FROM Users WHERE id = ' xyz ' AND passwd = ' ' or '1' = '1'*

As can be seen the WHERE clause of this query always evaluates to TRUE and thus an attacker can bypass the authentication. To prevent this SQLIA the application must

sanitize every single quote by replacing it with double quotes. If the code is sanitized properly the query becomes

*SELECT * FROM Users WHERE id = ' xyz ' AND passwd = ' " or "1" = "1'*

The values supplied by the *Client* are treated as string and thus prevents the SQLIA. Although sanitizing all the vulnerable spots is a sufficient condition to prevent SQLIA, it is a non-trivial task especially for large programs since a programmer needs to manually read through the code line by line and sanitize them. In this section we review the different types of known SQLI attacks [1]. For each such attack we identify a *pattern* of the attack. A pattern is a sequence of characters that will always appear in the URL for that particular attack type. We call such pattern the *signature* of the attack. Our aim is to extract a signature for the known SQLI attacks and then use these signatures to prevent such attacks. The USP of our approach is that it provides a generic framework such that it can be used with any web application to prevent SQLIAs.

A. Tautologies

In a tautology-based attack an attack code is injected using the conditional operator such that the query always evaluates to TRUE. In the example code given in figure 6, an attacker can invoke the code using the URL: *http://localhost/?*EmpId=*1' OR '1'='1*. The conditional *OR* statement will get appended to the SQL statement in line 5 and the query will always evaluate to TRUE. In this example it will render the following statement:

SELECT empinfo FROM EmpTable WHERE EmpID = '1' OR '1'='1'

In this type of attack the injected code will always start with a string terminator (') followed by a conditional operator (OR or AND). The operator will be followed by a statement that always evaluates to TRUE. So the signature for such attacks is the string terminator ('), *OR* or *AND* and a *SQL statement that evaluates to TRUE*.

B. Illegal/Logically Incorrect Queries

If an incorrect query is sent to a DBMS then generally it display the error with a description. There are several ways to perform illegal or incorrect queries like incorrectly terminating the string('), invalid conversions, using AND operator to perform incorrect logics, using order by, having clause to produce error, etc. In illegal/logically incorrect attacks an attacker sends an illegal query and uses the response (DBMS error message) to explore the whole database, like grabbing the banner, extracting columns, table names and records.

Consider the following SQL statement - *"SELECT accounts FROM users WHERE login='' AND pass='' AND pin= convert (int,(select top 1 name from sysobjects where xtype='u'))'*. In this example the injected select query attempts to extract the first user table (xtype='u') from the database's metadata table then tries to convert this table name into an integer. The type conversion is not legal therefore the DBMS will throw an exception say for example "*Microsoft OLE DB Provider for SQL Server (0x80040E07) ...*". The attacker thus gets useful information about the database. The signature for such attacks is type conversion statements, incorrect logic etc.

C. UNION Query

UNION is a SQL command which works for combing two queries. In Union query attacks by using UNION command an attacker merges their specially crafted queries with the original query to extract the records from a table other than the one intended by the developer. Continuing with our running example of figure 6.2, an attacker can invoke an invoke an union-query attack using the URL *http://localhost/?EmpId=' UNION <SQL statement>*.
This url will render the following SQL statement *SELECT empinfo FROM EmpTable WHERE EmpID = '' UNION <SQL statement>*. The table EmpTable will apparently not have a record with EmpId equal to '', therefore it will return null but the second SQL statement will get executed.

The signature of UNION-query attack is the UNION meta character followed by a valid SQL statement. This signature will not occur in a legal request since a *Client* is not supposed to send SQL statements in the URL hence there will be no collateral damage.

D. Piggy-backed Queries

In this type of attack, attacker binds another SQL statement by terminating the first using";". The first query will execute as normal but the subsequent injected queries will also get executed. An example for such an attack is to use the URL *http://localhost/?EmpId=10'; DROP TABLE EmpTable--* for the application given in figure 6.2.

The signature of this attack is (;), the database line terminator followed by a valid SQL statement.

E. Stored Procedure

A stored procedure is a set of SQL commands that has been compiled and stored on the database server. Databases often come with a number of preloaded stored procedures. Client applications can execute the stored procedure over and over again without sending it to the database server every time or compiling it. Since the stored procedure contains SQL statement, it can be exploited by an attacker by piggy-backing the attack code.

The signature of this attack will be the same as that of piggy-backed queries.

F. Blind SQLI

Blind SQL injection is identical to normal SQL Injection except that when an attacker attempts to exploit an application, rather than getting a useful error message, they get a generic page specified by the developer instead. This makes exploiting a potential SQL Injection attack more difficult but not impossible. An attacker can still steal data by asking a series of True and False questions through SQL statements.

To perform Blind SQLI an attacker may verify whether a sent request returned True or False in a few ways. Having a simple page, which displays article with given ID as the parameter, the attacker may perform a couple of simple tests if a page is vulnerable to SQL Injection attack. Suppose our example URL is *http://xyz.com/items.php?id=2* and the SQL query of the query string id=2 is *SELECT * FROM item WHERE id=2*. The attacker may try to inject any (even invalid) query, what should cause the query to return no

results e.g. *http://xyz.com/items.php?id=2 and 1=2*. Now the SQL query should look like this: *SELECT * FROM item WHERE id=2 AND 1=2,* which means that query is not going to return anything. If the web application is vulnerable to SQL Injection, then it probably will not return anything.

The possible guessing start with the AND operator and some time attacker also used the Conditional operators for performing this.

G. Timing Attack

Using some time-taking operation e.g. BENCHMARK(), will delay server responses if the expression is True. *BENCHMARK(5000000,ENCODE('MSG','by 5 seconds'))*- will execute 5000000 times the ENCODE function. Depending on the database server performance and its load, it should take just a moment to finish this operation. The important thing is, from the attacker's point of view, to specify high number of BENCHMARK() function repetitions, which should affect the server response time in a noticeable way.

Other databases than MySQL also have implemented functions which allow them to use timing attacks: *MS SQL 'WAIT FOR DELAY '0:0:10*

In this type of attack the IF ELSE statement is used for injecting queries. So the possible signatures of this attack are WAITFOR, IF,ELSE, BENCHMARK etc.

H. Alternate Encoding

In this attack, the injected text is modified so as to avoid detection by defensive coding practices and also many automated prevention techniques .In this technique; attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII, and Unicode, Because by this way they can escape from developer's filter which scan input queries for special known "bad character". For example attacker use char (44) instead of single quote that is a bad character.

Example:

SELECT * FROM users WHERE login= '' AND pass='' ; exec (char(Ox73687574646j776e))

This example use the char () function and ASCII hexadecimal encoding. The char() function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the shutdown command by database when it is executed.

VII. Defeating SQLIA Using URL Filter

As we have seen in the previous section an SQL code gets injected if an attacker manages to pass SQL Meta Characters (SQL expression) through the user input fields to change the behavior of predefined SQL queries. Thus if we can block the SQL commands in the request send to the *Application Server* we can prevent SQLIA. However, while blocking SQL commands we must ensure that legal queries and statements are not blocked. In this paper we propose to thwart SQLIA by using an URL filter [6]. Every Request coming from the *Client* must pass through the URL filter first before being processed by the Application Server. If the request contains any of the attack signatures mentioned in the previous section it is denied access to the database. Our URL filter is different from a validator([3]) that blindly prohibits SQL meta characters in the input. The proposed filter prohibits a SQL Meta character if it occurs in combination with some other characters such that the database can be abused.

In server-side architecture, a user invokes the services provided by the application server using a browser. The input provided by the user is usually sent to the application server in the form of a parameter string. The application server uses this input to generate a SQL query to retrieve information from the database or update it. As shown in figure 7, our proposed Meta filter is positioned between the user and application server. The filter intercepts the input from the user, parses it into SQL Meta character tokens. If the input from the user contains any attack signature then the injected input is treated as an attack and an error page is displayed [5], otherwise the input is processed by the application server normally.

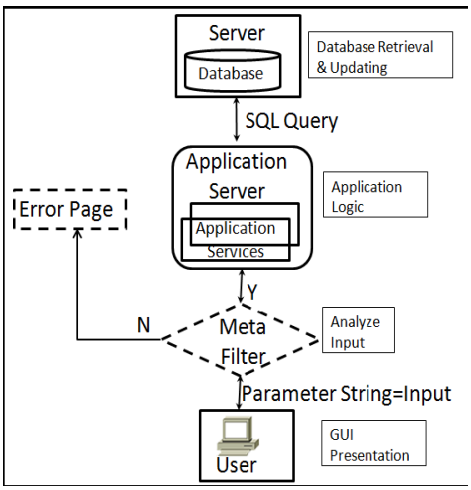

Fig. 7. Server-side Architecture and its interaction with our proposed Meta Filter

As can be seen from figure 7, the SQL query and as such the SQL Meta characters are generated by the application server. Our proposed Meta filter checks for the presence of SQL Meta characters before the input is processed by the application server. Therefore, our proposed solution will not block legal inputs. Moreover, the attack patterns have been so designed so that it is robust to SQL Meta characters that can accidentally occurs in a legal input.

VIII. Result and Analysis of CSRScanner Using Filter

To discuss the effectiveness of our filter approach, we applied our scanner to a web application which has been deployed in our localhost. In our experimental set up we have implemented our web application in two following ways:

Web Application without filter and Web Application with filter. Again in both the cases we have checked the sanitization scenarios. Our experimental result shows that if we deploy filter with the web application then sanitization of every input point is not required. Our filter is able to handle all SQLI even if we do not use sanitization for every input.

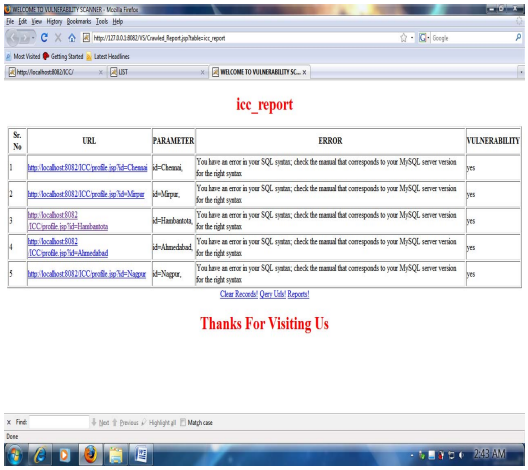

Fig. 8. Scanning report when web application is deployed without sanitization and without filter

Figure 8 shows the scanning report of CSRScanner when our filter is not deployed in our application and the input points are also not sanitized. In this report we have four parameters to consider. In serial no 1 our URL is http://localhost:8082/ICC/profile.jsp?id=Chennai, here id='Chennai' is a query string of the URL and id is the input for this page i.e. profile.jsp. As the input points of our web application is not sanitized and the filter is also not deployed so our scanner can easily find out the vulnerability of different input points and figure 8 shows the result accordingly. The Parameter field shows different vulnerable points through which CSRScanner checks different database syntax error by putting specially crafted malicious code. As CSRScanner performs successful injection in this case so vulnerability parameter in this report shows "YES" where successful injection takes place.

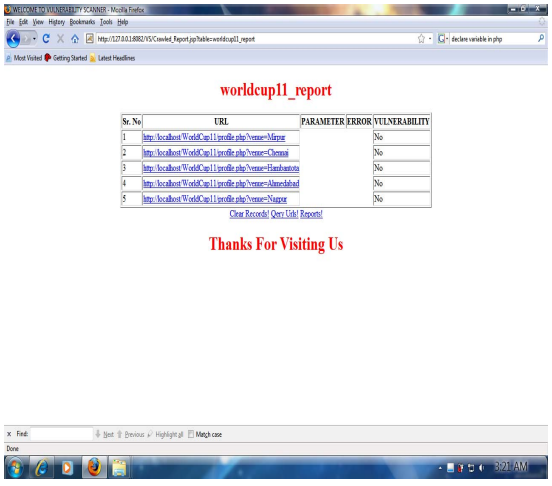

Fig. 9. Scanning report when web application is deployed with sanitization but without filter

Figure 9 shows the scanning report when we have the web application which is sanitized but we have not deployed our filter approach. The first URL of the report is http://localhost/WorldCup11/profile.php?venue=Mirpur where venue is the input point and venue = Mirpur is our query string. As the web application is sanitized, so after crawling all input points, CSRScanner checks the vulnerability for each input point and shows vulnerability as "NO". In this case, CSRScanner is unable to inject any malicious code which can produce any database error.

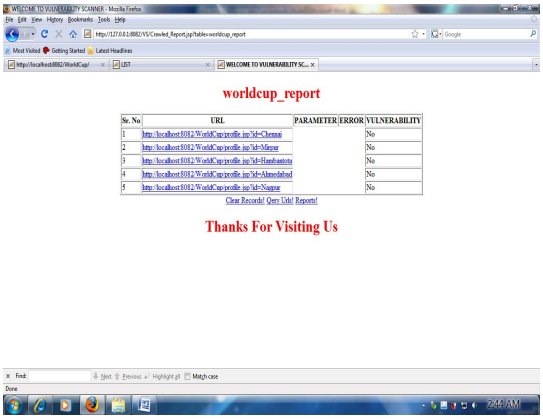

Fig. 10. Scanning report when web application is deployed without sanitization but with filter

Figure 10 shows the scanning report of our web application when our filter is deployed but input points are not sanitized. Figure 10 is just same as figure 9 but here we consider the pages which have been implemented in jsp where the pages, which are shown in figure 9, have been implemented in php. We applied our scanner in both the cases. Figure 9 shows the report of with sanitization but without filter deployed in web application where figure 10 provides the result of without sanitization but with filter approach. In both the cases we fixed same number of input points. CSRScanner finds all the input points and reports "NO" as vulnerability assessment of them. From these three different results we can say that our CSRScanner is able to crawl all input points what we set in our web application and also able to fix all the vulnerability what we defined for each input points.

IX. Conclusions

In this paper we first propose a methodology to scan a webpage in a controlled environment and discover the SQL Injection vulnerabilities present. Next we provide a security framework to prevent SQLIA. The web vulnerability scanner proposed, CSRScanner, is light-weight and fast. The tool crawls through all the web pages of a web application to discover vulnerable spots, performs a controlled exploit of the vulnerabilities at these vulnerable spots and finally verifies success of the attack and reports the result. The performance of the tool was measured by comparing it well-known SQLI scanners. Results show that our proposed scanner is able to cover more vulnerability in lesser time and has fewer false positives. The security framework proposed to defeat SQL Injection attack is based on an URL Meta filter. We analyzed well-known SQL Injection attacks and tried to identify a *signature* for each such attack. The filter works by checking the presence of these *attack signatures* in the user input before it is processed by the application server. The proposed framework is generic and does not depend on the application server as well as the underlying database. The efficiency of the filter was tested by using the CSRScanner.

References

[1] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures," *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Arlington,VA,USA, March 2006.

[2] Jose Fonseca, Marco Vieira, and Henrique Madeira,"Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," *In Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing* (PRDC '07). IEEE Computer Society, Washington, DC, USA, pp. 365-372. DOI=10.1109/PRDC.2007.63 http://dx.doi.org/10.1109/PRDC.2007.63

[3] Atefeh Tajpour, Maslin Masrom, Mohammad Zaman Heydari, Suhaimi Ibrahim,"Evaluation of SQL Injection Detection and Prevention Techniques," *2nd International Conference on Computational Intelligence, Communication Systems and Networks*, Liverpool, United Kingdom pp. 216-221.

[4] Nuno Antunes, Marco Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services." *15th IEEE Pacific Rim International Symposium on Dependable Computing* , 2009, pp. 301-306.

[5] Susanta Nanda, Lap-chung Lam, Tzi-cker Chiueh, " Web Application Attack Prevention for Tiered Internet Services," *In The Proceeding of the Fourth International Conference on Information Assurance and Security (IAS 08),* Sept 2008, pp. 186-192

[6] Sangita Roy, Avinash Kumar Singh and Ashok Singh Sairam, " Analysing SQL Meta Characters and Preventing SQL Injection Attacks Using Meta Filter, " *International Conference on Information and Electronics Engineering, ICIEE 2011, IACSIT Press*, ISBN. 978-981-08-8637-0, vol. 6, pp. 167-170, 28-29 May 2011, Bangkok, Thailand.

[7] Netsparker ofMavituna Security Ltd.: http://www.mavitunasecurity.com/netsparker/visited on January 2011

[8] Acunetix of Acunetix Ltd.:http://www.acunetix.com visited on January 2011.

[9] WebCruiser of Janus Security.:http://sec4app.com visited on January 2011.

[10] OWASP (Open Web Application Security Project) https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project visited on January 2011.

Sangita Roy obtained her B.Tech degree from West Bengal University of Technology, India in the year 2005. She obtained her M.tech degree from Kalinga Institute of Industrial Technology, Bhubaneswar, Orissa in 2008. Currently she is pursuing her Ph.D degree from Indian Institute of Technology, Patna. Prior to this she was working as an Assistant Professor at Kalinga Institute of Industrial Technology, Bhubaneswar, Orissa. Her research interests include steganography, web application security and network security.

Avinash Kumar Singh obtained his B.Sc(IT) and M.Sc(IT) degree from Kumaun University, Nanital in the year of 2007 and 2009 respectively. He has received his M.Tech degree from Kalinga Institute of Industrial Technology in the year of 2011. Currently he is working as an Assistant professor in Gwalior Engineering College, Gwalior. His research interests include web application security and network security. He is also a certified Ethical Hacker.

Dr. Ashok Singh Sairam obtained his B.Tech degree from National Institute of Technology Silchar, India in the year 1993. He obtained his M.Tech and Ph.D degree from Indian Institute Technology Guwahati, India in 2001 and 2009 respectively. Currently he is working as an Assistant Professor at Indian Institute of Technology Patna, India. Prior to this he was working as a senior research officer at Indian Institute Technology Guwahati, India. His research interests include network security, wireless networks and traffic engineering. He is currently working as a chief investigator on a major network security project sponsored by the department of Information Technology, government of India. He has given invited lectures and served as PC member in several international conferences.