A Software Engine to prevent SQL Injection in Databases

Final Report

Lakshya Tandon

ABSTRACT

SQL Injection is one of the most feared attack in the online world. In this a malicious code is inserted as an input to the text field of an online application. As per the statistics 27 percent of the total web application attacks are SQL Injection attacks. Hence it proves to be the most prominent attack on the databases today as web applications rely on a backend database to store data. Now there is a need for preventing these type of attacks. What most of the developers do is that they write some piece of code within their application to check for malicious code in the input fields. Although, this is a nice thought to start with but most of the times they cannot take care of all types of SQL injection attacks as their main priority is to develop the functionality of application rather than implementing its security. On the other hand, we might have some applications which are non-scalable i.e. it becomes difficult to modify them with changing policies. In order to overcome these issues this project demonstrate the successful prevention of SQL Injection attacks by developing an API for prevention which is also termed as a software engine in this report. It is then shown in the later stages that how this API can be integrated in the application and various SQL injection attacks are performed to validate the API.

1. **Introduction**

   Most of the modern web and desktop based applications use databases to maintain their data. These applications send SQL queries to the database composed of the user fed inputs. The database returns the correct data only if the input values are validated. This a basically how authentication works in a web application. But sometimes a malicious user provides SQL code to the application's input field which gives him the access to the database. This technique of hijacking a database is known as SQL injection. [2] It is one of the most serious threat to database driven applications. Once an attacker finds that a particular application is vulnerable to SQL injection he can do a variety of things ranging from alteration of data to a complete deletion of the database. This action by an attacker causes the application to crash or malfunction. In some of the cases when some data or database is deleted, a large economic loss could also occur. Also, if the database administrator has not taken proper backups the loss could be irreversible. The need of the hour is to stop these kind of attacks. Sometimes application developers use some kind of validation techniques but they aren't enough to stop SQL Injection. Some of the research has been done on SQL injection where different methodologies used for SQL injection were studied and measures to prevent it were suggested. [1] There also exist some applications which try to prevent SQL injection but none of them has proved to be accurate enough. This project aims to develop an API which would work against all SQL Injection attacks.

2. **Motivation**

   The primary motivation to work on this type of a project came from the fact that there still exists some old web applications which are vulnerable to SQL Injection. These lack proper

documentation and have less support of the developers who developed them so it becomes very difficult for a new developer to upgrade its security as not much code can be altered with. Considering all these facts I thought an API could be a life saver for all these applications. The developer just needs to pass all the input data through the API and the API would take care of all the security breaches (SQL Injection issues in my case). Also another problem which further amplified my motivation was that there were different types of SQL injection attacks which can be performed on a web application. Developers while developing the application focus more on the functionality of web application. They might also cover the security aspect but most of them do not tend to research about each of the Injection attacks and develop an antidote to them. This still leaves the application vulnerable to SQL Injection. As this would cover all SQL Injection attack techniques the developer just needs to integrate it into his project without worrying much about the security breaches. This way he can concentrate more on the core functionalities of the application.

3. Types of SQL Injection Attacks

As mentioned previously in this report there are different types of SQL Injection attacks and can be done using different ways. These attacks are mentioned as follows:

Tautology

The term tautology is used for something which is always true. While an SQL query executes in the database, it checks the database for the parameters which go along with the query. If they match with the values in the database, the query becomes true. Now the attacker takes advantage by specifying an already true statement in the input field. This true statement can be two same data values with an '=' sign between them. Let's consider an example for this:
We have a login page which wants the user to specify his username and password. This query which authenticates user is:

SELECT count(*) FROM tablename WHERE username = 'textbox1value' AND password = 'textbox2value'

If the attacker gives the textbox1value as *' or 1 = 1* -- , the query becomes:

SELECT count(*) FROM tablename WHERE username = '' or 1 = 1 -- AND password = 'textbox2value'

Here the query gets commented after '--' and only the query prior to this gets executed which would always return the total count. This is how attacker can breach the security.

Logically Incorrect Queries

These queries are the queries which are syntactically true but have some logical error when they are executed in the code. The attacker does this intentionally to extract some information from the database. Let's explain this on the previous example:

The attacker gives the following information in the password field of the query and the query becomes:

SELECT count(*) FROM tablename WHERE username = 'textbox1value' AND password = '' AND pin= convert (int,(SELECT top 1 name FROM sysobjects WHERE xtype='u'))

In this case the later statement is supposed to give a type conversion error and when this error occurs it displays some specific information related to the database. This information can be useful for the attacker to develop more SQL Injection attacks.

Union Query

Using this kind of an attack, the attacker can trick the application return data from a different table. Attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application. Let's suppose the query executed from server side is

SELECT Name, Phone FROM Users WHERE Id=$id

The following can be injected in the field Id: *$id=1 UNION ALL SELECT creditCardNumber,1 FROM CreditCardTable*

Final query will be:

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

This will join the result of the original query with all the credit card users.

Piggy Backed Queries

In this the attacker appends another query or a SQL keyword after the query which is to be executed by the application. This results in the execution of both the queries. Even if the first query fails the attacker is able to execute second query or the command to the database. The two queries are separated by ';'. This type attack can prove to be very lethal as it can even shut down the database. An example for this would be:

SELECT info FROM users WHERE login='doe' AND pin=0; shutdown

This command will shut down the database service, the database connection will close and it would no longer be available to the user. The database administrator would have to restart the service in order for the user to access it.

Stored Procedures

Stored Procedures are written on server side of the application. Some developers think that using stored procedures is the best way to prevent SQL Injection attacks but the truth is something else. Even stored procedures can be injected with SQL Commands. These commands can be passed onto the stored procedure input along with the input from the user to the application. This works almost the same as piggy backed query. Consider the following stored procedure.

CREATE PROCEDURE DBO.isAuthenticate @userName varchar2,
@pass varchar2, @pin int
AS
EXEC("SELECT accounts FROM users WHERE login='" +@userName+ "' and
pass='" +@password+"' and pin=" +@pin);
GO

The query in this procedure can again be injected as:

SELECT accounts FROM users WHERE login='sid' and
pass=''; SHUTDOWN; -- and pin=" +@pin

This will shut down the database service.

Inference attacks

In these type of attacks, the attacker usually tries to infer information from the database by proving some queries. There are two types of inference attacks:

- *Blind Injection*
  Sometimes the developers hide error details by using exception handling techniques so it becomes difficult for the attacker to find out if the application is prone to injection or not as logically incorrect queries don't work as required. Now the attacker checks for SQL Injection vulnerability by asking a series of true or false questions. For example, consider the two queries:
  Q1: SELECT accounts FROM users WHERE login='doe' and 1=0 -- AND pass= AND pin=0
  Q2: SELECT accounts FROM users WHERE login='doe' and 1=1 -- AND pass= AND pin=0

  If the first query returns false, the attacker isn't sure if it is because of incorrect login or fallacy but if the second query returns true the attacker becomes sure that the application is prone to SQL injection.
- *Timing Attacks*
  The attacker gathers information from the database by observing the timing delays in the database responses. If-then statements are used within the queries. Suppose an attacker gives a time delay of 5 seconds in the query and when he gives an input he observes that the database responds 5 seconds later than the expected time, then he infers that his injection is working and there is a further scope of injection within the application.

Alternate Encoding

The attacker may modify the injection query by using an alternate encoding technique such as hexadecimal. This way he can escape any developer filter scans. For example, he can use *char(0x73687574646f776e)* instead of *shutdown* command. The following query would work to completely shut down the database.

SELECT accounts FROM users WHERE login=" AND pin=0; exec (char(0x73687574646f776e)).

These are all different SQL Injection attacks and my tool would protect against all of them in the best way possible.

4. Methodology

The developed API basically works on the principle of string and pattern matching to successfully identify malicious code in the input fields. Now, it needs to match inputs against something so an XML file is maintained which contains all the keywords against which the input string is matched. So basically there are two components of this engine: XML File and Filtration API.

XML File

As stated above, the XML file contains all the keywords which are to be matched against. There keywords are put separately in the XML file under different nodes. The structure of XML file is such that there is a root node, then the child of root node is the attack type which has the child Sub category of attack type. The child of each sub type contains the keywords associated with it.

Only the first child node of root has this type of structure. The first node is called as the *GeneralSQL* Node which has sub-nodes or its child nodes as the type of SQL queries which can be select, insert, update, delete etc.

All the other nodes apart from the first child node of root have a different sub structure. They have attack type to as a child node to root and keywords as child node to attack type.

Currently in the XML file we have only two sub nodes to root namely *GeneralSQL* and *SQLReservedKeywords*. The following screen shots will give a better picture of how the nodes look like.

```
<Root>
  <AttackType Name="GeneralSQL">
    <SubType Name="SelectSt">
      <Keyword>select</Keyword>
      <Keyword>from</Keyword>
    </SubType>
    <SubType Name="update">
      <Keyword>update</Keyword>
      <Keyword>where</Keyword>
    </SubType>
    <SubType Name="delete">
      <Keyword>delete</Keyword>
      <Keyword>from</Keyword>
    </SubType>
    <SubType Name="drop">
      <Keyword>drop</Keyword>
      <Keyword>table</Keyword>
    </SubType>
```

Fig1: For GeneralSQL

```
<AttackType Name="SQLReservedKeywords">
  <Keyword>add</Keyword>
  <Keyword>all</Keyword>
  <Keyword>alter</Keyword>
  <Keyword>and</Keyword>
  <Keyword>any</Keyword>
  <Keyword>fetch</Keyword>
  <Keyword>for</Keyword>
  <Keyword>shutdown</Keyword>
  <Keyword>exit</Keyword>
  <Keyword>null</Keyword>
  <Keyword>proc</Keyword>
  <Keyword>escape</Keyword>
  <Keyword>delete</Keyword>
```

Fig2: For SQLReservedKeywords

This XML file is created using tool called XML Creator which was developed specifically for making such XML files and editing them.

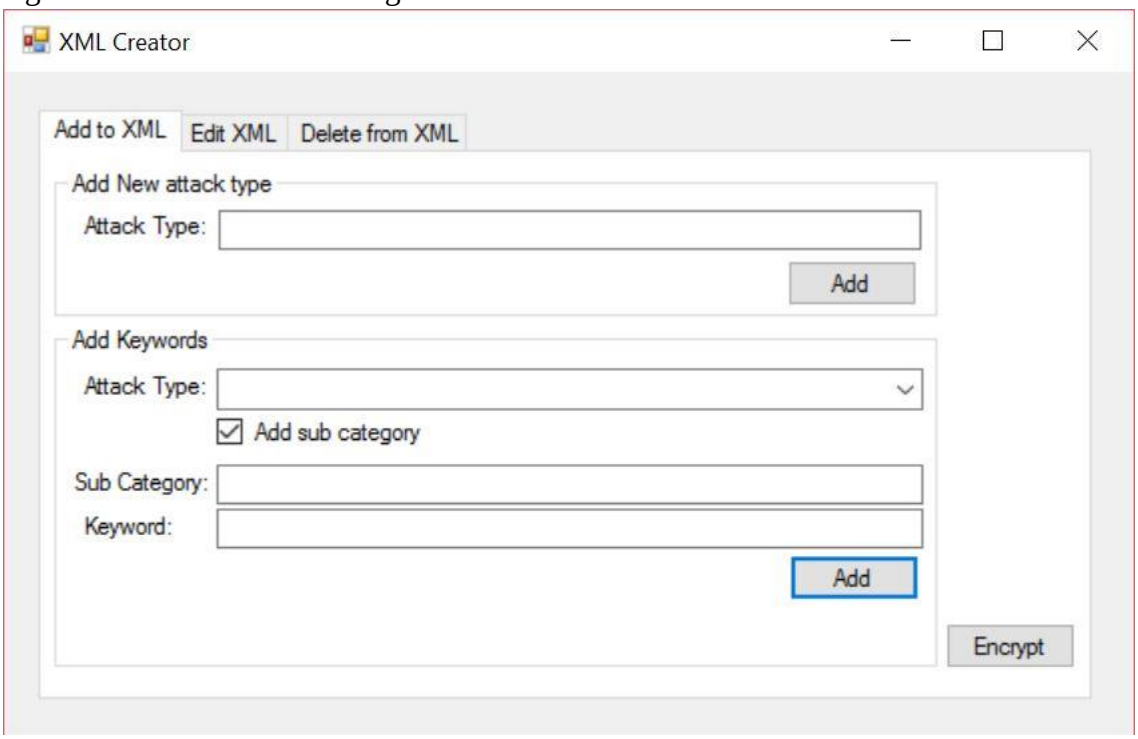

Fig3: Show a screenshot of XML Creator

The file is kept encrypted on the machine so as to make the file contents more secure. The *Encrypt* button in XML Creator is used to encrypt the file.

Filtration API

The filtration API uses the XML file to validate the input. As the file is encrypted it first decrypts the file and loads file contents in the systems memory. Once it has done that, it starts with the validation process. The API has a method which can be used to perform what type of check has to be performed. The method is defined as:

```
public static bool checkInjection(string value, string path, char mode);
```

checkInjection can be called by the developer when the API is invoked. The arguments of this method can be interpreted as:

- value: The string which is to be checked for injection goes here.
- path: The path of the xml file goes here.
- mode: It is used to specify what kind of SQL Injection check the developer wants to perform. He may perform all of the different types of SQL Injection checks mentioned above or may choose just one. The following annotations represent the different letters for different Injection checks.
    - a – All
    - t – Tautology and Inference attacks
    - l – Logically Incorrect Queries
    - u – Union Queries
    - p – Piggy Backed Queries and Stored Procedures Injection
    - e – Alternate Encoding

The techniques which I've used for string and pattern matching are mentioned below.

Detection of General SQL Query

There are many cases of Injection where an SQL query is fed as an input to the textbox. Hence it becomes very important to identify if the input contains a pattern of an SQL Query. For this purpose, I have made *GeneralSQL* attack type in the XML file. As the child nodes to this attack we have different SQL statement nodes each of which contain keywords. The API checks for each SQL statement one by one. It starts with *select*, reads the keywords and sees if they occur in a continuous pattern in the input. It looks for the first keyword of *select* (sub attack), if it finds it then it splits the input string from the point where *select*(keyword) occurs and looks for the next keyword which is *from* in the second half of the split string. If *select*(keyword) doesn't occur, it moves on to the next sub attack which is update. It continues this until it traverses all children of *GeneralSQL*.

If it finds continuous pattern of keywords in the sub attack node. It specifies the input as an attack and stops it going to the database.

Example: Suppose the input is

';convert (int,(SELECT top 1 name FROM sysobjects WHERE xtype='u' --

Here the algorithm will break the string where it finds *select.* Now it will start searching for *from* in the second half of the string.

top 1 name FROM sysobjects WHERE xtype='u' --

Now, as soon as it finds from in the above string the API will specify it as malicious.

Detection of Tautology

The algorithm which checks for tautology basically looks for the statements which are always true. It first aims to find out '=' sign in the input field. Once it finds it then I compare both left and the right side of '=' sign. If it finds it equal, it specifies it as tautology attack.

Please note that it considers the fact that input value 'hello1=1world' is not a tautology while 'hello 1=1 world' is. This means that it considers spaces while doing so.
This also identifies any interference attacks taking place.

Detection of Logically Incorrect Queries
Logically Incorrect Queries tend to introduce some logical errors in the input. Usually these tend to cast a query to a datatype which isn't valid. So for this the algorithm first tends to find convert keyword in the input string. Once it finds it, it splits the input into two string dissecting it from *convert* and calls the function for detection of general SQL Query and passes the second string as a parameter to it. If the called function returns true that means input has malicious code.

Detection of Union Queries
It works almost the same way as the logically incorrect queries. The difference being that *union* is used instead of *convert* and the latter part of query detection remains the same.

Detection of piggy backed queries
This algorithm first searches for ';'. If this is found, then it checks the latter part for a query. If a query is found after ';' the algorithm returns false and specifies input as malicious. If a query is not found it checks for the *SQLReseervedKeywords* after ';'. If one of the keywords in XML file matches, then it returns false else it specifies input as genuine.
It works in a similar manner for stored procedures and also check for an extra word 'exec'.

Detection of Alternate Encoding
The algorithm checks for '*char*' in input string. If it finds char it proceeds further by checking for a string of pattern *0x73687574646f776e* and separates the text after *0x.* It then puts the text through a hex to string converter method. The output which this method gives is compared against the *SQLReservedKeywords.* If the output gets matched the input is termed as malicious.

5. Technologies Used

Three different projects were developed which were filtration engine (API), XML creator and Demo web application. The primary language used to develop all three of them was C# and was used alongside .Net framework. For developing the demo web application ASP.Net was used along with C# as backend language. SQL Server 2014 was installed on the local machine and the database was created to cater the web application. XML libraries included with the .Net framework were used to work with XML files.

6. Results

To check the validity of the API a login page was developed and connected to the database. Then the API was integrated in the login form and was tested by performing different types of SQL Injection attacks. The following images show the results.

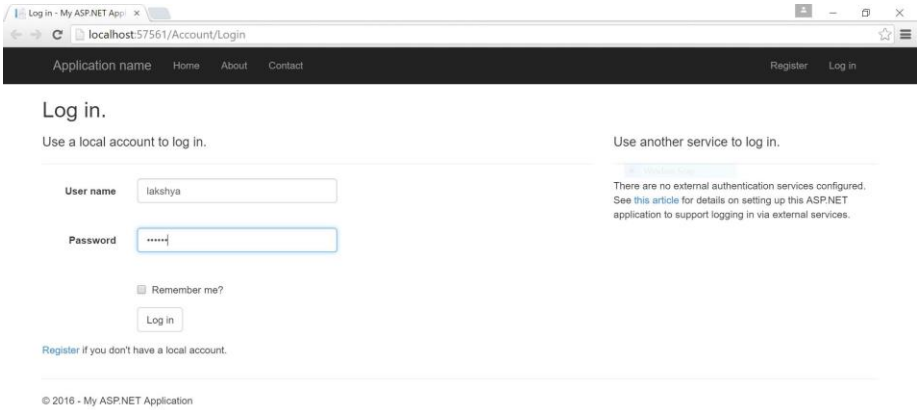

Fig 4: Normal Login

Fig 5: Successful Login

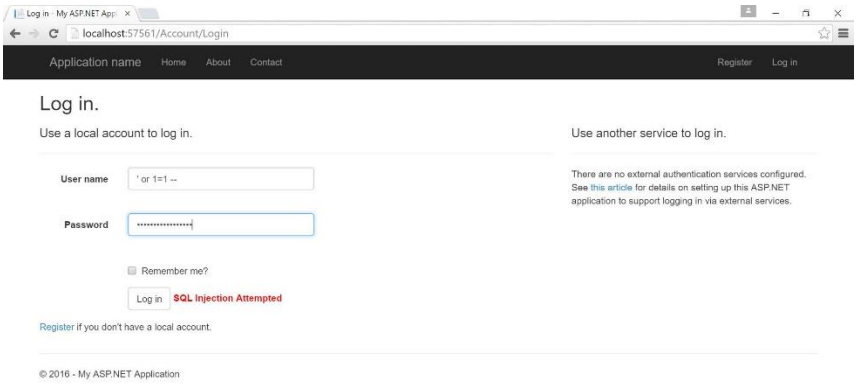

Fig 6: Tautology; SQL Injection attempted

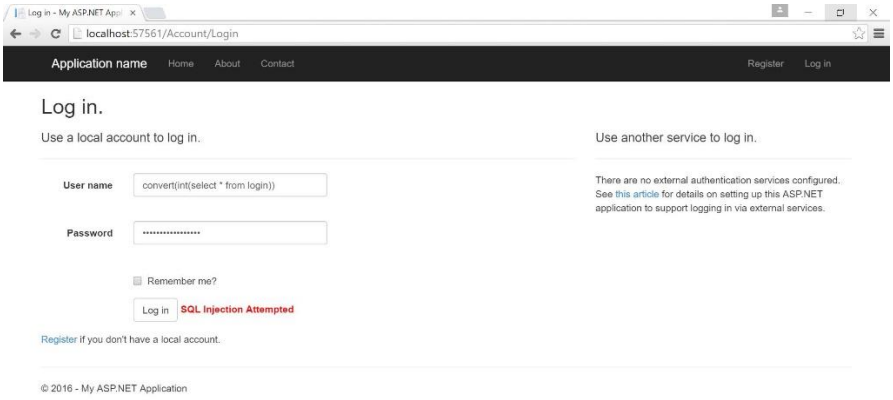

Fig 7: Logically Incorrect Query; SQL Injection attempted

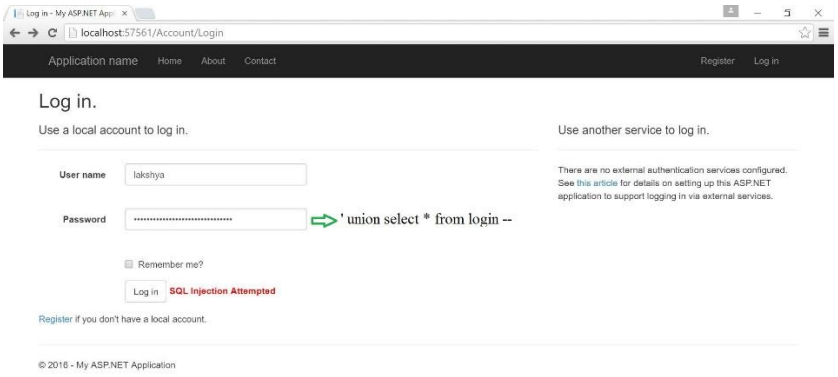

Fig 8: Union query; SQL Injection attempted

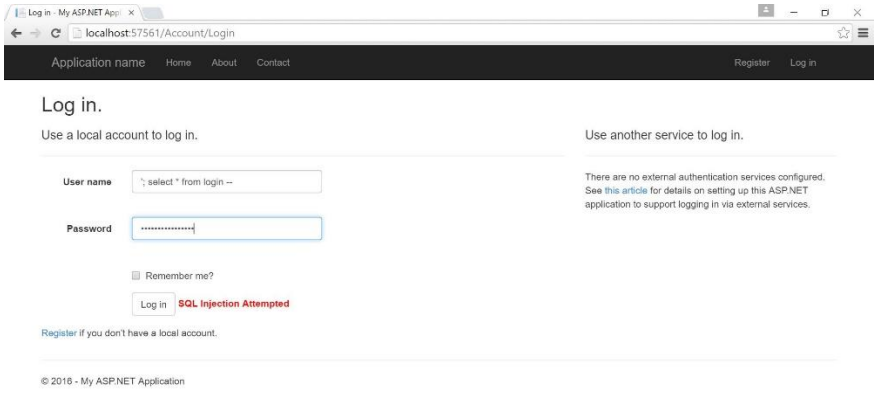

Fig 8: Piggy backed query; SQL Injection attempted

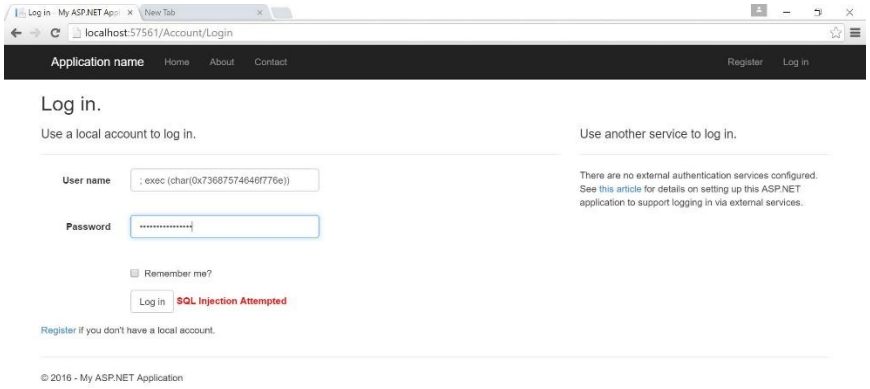

Fig 8: Alternate Encoding; SQL Injection attempted

7. Conclusion and Future work

As proposed in the project, earlier the aim was to cover all of the different SQL Injection techniques and provide a collective solution to it. The project developed helps to detect different types of SQL injection attacks and successfully stops malicious code to enter the database.

Although it covers a lot of things but the future scope of the project could be the study of sub areas in each of the SQL Injection techniques and its implementation in the current project. Right now it covers all SQL injection techniques but doesn't explore the extent to which these techniques could be modified to breach the current security standards.

8. References

[1] Sharifi, M., Tajpour, A., & Ibrahim, S. (2012).Web application security by SQL injection detection
tools. In *Int J Comp Sci Issues*, (Volume: 9) (pp. 332–339).

[2] Tajpour, A., Heydari, M., Masrom, M., & Ibrahim, S. (2010). SQL injection detection and prevention
tools assessment. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on (Volume:9)* (pp. 518 - 522). Chengdu: IEEE.

[3] Halfond, W. G., Viegas, J., & Orso, A.(2006). A Classification of SQL-Injection Attacks and Countermeasures. In *SSSE*.