Chapter XCIII
Bind but Dynamic Technique:
The Ultimate Protection Against SQL Injections

Ahmad Hammoud
Lebanese American University, Lebanon

Ramzi A. Haraty
Lebanese American University, Lebanon

INTRODUCTION

Most Web developers underestimate the risk and the level of damage that might be caused when Web applications are vulnerable to SQL (structured query language) injections. Unfortunately, Web applications with such vulnerability constitute a large part of today's Web application landscape. This article aims at highlighting the risk of SQL injection attacks and provides an efficient solution.

BACKGROUND

Attackers usually make use of SQL injection attacks in order to compromise both the confidentiality and integrity of RDBMS- (relational database management system) powered Web applications. In some cases, even their availability is compromised (Cerrudo, 2002).

In his "Introduction to SQL Injection Attacks for Oracle Developers," Stephen Kost (2004) says,

application audits have found many web applications vulnerable to SQL injection even though well established coding standards were in place during development of many of these applications. Function-based SQL injection attacks are of most concern since these attacks do not require knowledge of the application and can be easily automated.

Fortunately, developers can use simple and easy-to-implement techniques to defend against SQL injection attacks. There is no need for a special tool or to introduce dedicated hardware; simple coding practices can do the job.

MAIN THRUST: SQL INJECTIONS

To tackle the problem immediately, an example will be given so that the concept of SQL injection is clear before solutions are explored. Consider the following code.

```
SQL = "Select * from UsersTbl WHERE Usr ="
SQL = SQL + SuppliedUsr
SQL = SQL + "And Pwd = "
SQL = SQL + SuppliedPwd
Execute SQL
```

The above code is a typical SQL statement that will be executed whenever a user is trying to log in. This is the code that exists behind the log-in button on the log-in page. Obviously, the SQL statement attempts to find a record in the table called UsersTbl so that the two fields *user* and *password* are equal to the ones supplied by the user. If the *Execute* statement returned rows, then this means the supplied user name and password are correct and the user will be allowed to proceed because he or she is authenticated. The following two scenarios may occur.

Normal User Scenario

Suppose the supplied user name and password is as follows.

- SuppliedUsr : 123
- SuppliedPwd : 456

Then, the SQL statement that will be executed is as follows:

```
Select * from UsersTbl WHERE Usr = 123 And Pwd = 456
```

If there is such a record in the UsersTbl table, then the EXECUTE statement returns a record and, as a result, the user will be able to proceed. If no such record exists, the EXECUTE statement returns zero records. Consequently, the user will be asked to try again.

Hacker Scenario

Suppose the supplied user name and passwords are as follows.

- SuppliedUsr : 123 --
- SuppliedPwd : whatever

Then, the SQL statement that will be executed follows.

```
Select * from UsersTbl WHERE Usr = 123 -- And Pwd = whatever
```

That way, a hacker can deceive the code and bypass the authentication because the above SQL statement will always return the record of the user 123. This is due to the fact that the two consecutive dashes are used in SQL to comment a line so the DBMS will ignore everything after them. Therefore, the password part of the WHERE clause is ignored and the hacker will be able to log in as if he or she is the user 123. Quite trivial yet a catastrophic trick!

Categories of SQL Injection Attacks

SQL Injection attacks against databases can be categorized as follows (Kost, 2004).

SQL Manipulation

This occurs when the SQL statement is modified through the use of SET operations or when the WHERE clause is exploited to let the SQL statement return a different set of rows. Modifying the WHERE clause of the user authentication statement is among the most well-known attacks. Using several tricks and techniques, hackers will deceive the code so that the WHERE clause will always result in true (Overstreet, 2003).

As an example, consider the following piece of code.

```
Declare @SQL nvarchar(100)
Declare @SingleQuote char(1)
```