SQL Injection Using a Character Field

Posted At : February 22, 2008 5:21 PM | Posted By : Mark Kruger
Related Categories: Coldfusion Security

Ok, I admit it. Most of the examples of SQL injection that I give use a numeric field. Why? Because to inject using a character field requires manipulating single quotes. Since Coldfusion escapes single quotes automatically when using the cfquery tag these attacks are much more difficult to pull off. It may surprise you to know that your character fields can *still be vulnerable* and it is my belief that you should still use CFQueryparam. In fact, one of the attacks below can work even if you *do* use cfqueryparam. Check it out.

Alternate Escapes

This is a vulnerability that does not exist on every platform. MS SQL is not affected for example, but MySQL *is* affected. It has to do with alternate ways of escaping characters. You probably already know that single quotes are escaped by doubling. To use the string *I can't get no satisfaction* to select against a (grammatically poor) database table you would need to escape the apostrophe in *can't* to make it **I can''t get no satisfaction**. This would result in a successful select. In fact Coldfusion does this for you automatically. The following code works:

```
<Cfset song = "I can't get no satisfaction"/>
    <cfquery ...>
        SELECT     *
        FROM    IgnominiousHits
        WHERE    Title = '#song#'
    </cfquery>
```

It works because inside of CFQUERY coldfusion converts **Can't** to **Cant''t**.

But database platforms sometimes provide other ways of escaping characters. For example, MySQL allows you to escape a single quote *either* by doubling it *or* by using the backslash. As an example, here is a syntactically correct MySQL statement:

```
SELECT  *

FROM IgnominiousHits

WHERE Title =  'I can \'t get no satisfaction'
```

Now if you tried to use that in a CF statement as a variable it would error out.

```
<Cfset song = "I can\'t get no satisfaction"/>
    <cfquery ...>
        SELECT     *
        FROM    IgnominiousHits
        WHERE    Title = '#song#'
    </cfquery>
```

This code would produce something like:

```
SELECT  *
```

```
FROM IgnominiousHits

WHERE Title =  'I can\''t get no satisfaction'
```

You see how Coldfusion automatically doubles the quote? Look carefully. If the first quote mark is escaped by the backslash, that second quote mark (the one *added by Coldfusion*) actually terminates the string. That means anything that comes afterward is fair game for the hacker. For example, consider the following:

```
<Cfset song = "I can\' OR 1 = 1 -- t get no satisfaction"/>
    <cfquery ...>
        SELECT     *
        FROM    IgnominiousHits
        WHERE    Title = '#song#'
    </cfquery>
```

This would result in the following SQL statement:

```
SELECT  *

FROM IgnominiousHits

WHERE Title =  'I can\'' OR 1 = 1   -- t get no satisfaction'
```

The two dashes "comment out" the rest of the string literal leaving "or 1 = 1 as the tail end of the WHERE clause. Obviously this benign example would return all the hits in the IgnominiousHits table (it would be a bloodbath). Does Cfqueryparam solf this problem? Yes indeed it does. By binding the whole string to a variable and *type* cfqueryparam insures that the string will always be treated as a string and never read as anything else.

PreserveSingleQuotes() is Madness

In many Coldfusion applications throughout the web the use of Preservesinglequotes() is common. This is probably a legacy of folks coming to CF from ASP or JSP where concatenating a string together and passing it to an SQL execute function is standard practice. It looks something like this:

```
<cfset string = "SELECT * FROM users WHERE username = '#form.username#'"/>
    <cfquery name="getUser" datasource="myDSN">
        #preserveSingleQuotes(string)#
    </cfquery>
```

This aptly named function keeps CF from escaping the quotes and creating a syntax error. In so doing it should be obvious that it also opens up the flood gates to injection. If form.Username contains the string *Mkruger' OR username LIKE 'Admin%'* it is going to be able to drill into the database. Although the use of preserveSingleQuotes is a fairly obvious vulnerability the next one is not so obvious.

Runtime Execution in Stored Procedures

Hang around long enough and you will hear folks talk about how wonderful stored procedures are for encapsulation and security. Stored procedures *can* be the bees

knees to be sure but they are not a panacea. In fact, one particular use of stored procedures can bite you even if you *are using cfstoredprocparam* (the procedure equivilent of Cfqueryaparm) or cfqueryparam. Consider my simple example - this time using MS SQL:

```
<cfquery ...>
CREATE PROCEDURE usp_test
     (    @whereClause nvarchar(2000) )
WITH RECOMPILE
AS
SET NOCOUNT ON


DECLARE @STR nVARCHAR(4000)

SELECT @STR = 'SELECT * FROM stats WHERE ' + @whereClause

print @STR

EXEC sp_executesql @STR
</cfquery>
```

This procedure takes a *where clause as a string* and concatenates the entire statement for execution. Why would anyone want to do this? In my experience some shops are narrowly focused on using SPs for everything. In most cases it works fine but it is exceedingly difficult to do those fancy search forms with lots of different search parameters. The same task is really easy and maintainable using Coldfusion logic directly inside of a cfquery tag. For example:

```
<cfquery ...>
        SELECT     *
        FROM     users
        WHERE    username = '#form.username#'
        <cfif isDefined('form.lastname') AND NOT isEmpty(form.lastname)>
            AND    lastName = '#form.lastname#'
        </cfif>
        <cfif isDefined('form.email' AND isValid('email',form.email)>
            AND email = '#form.email#'
        </cfif>
    </cfquery>
```

Please note that I did NOT use cfqueryparam in the above example only to save a little space. In production, Cfqueryparam would *definitely be used*. Code like that above is simple and visually understandable in CF. Not so much in a stored procedure where you have to account for nulls, empty strings and the like - and use coalesce and other less common SQL techniques. So ironically shops that want to use stored procedures for security and encapsulation end up passing whole or partial query *statements* as strings and converting them into executable SQL after they reache the SQL server. They do something like this:

```
<cfsavecontent variable="whereclause">
        username = '#form.username#'
        <cfif isDefined('form.lastname') AND NOT isEmpty(form.lastname)>
            AND    lastName = '#form.lastname#'
        </cfif>
        <cfif isDefined('form.email' AND isValid('email',form.email)>
            AND email = '#form.email#'
```

```
        </cfif>
    </cfsavecontent>
    <!--- run query --->
    <cfquery ...>
        usp_test (<cfqueryparam cfsqltype="CF_SQL_LONGVARCHAR" value="#whereclause#"/>)
    </cfquery>
```

Of course the string passed to the SQL server bound as a string. But then lo and behold it is treated as part of an executable SQL statement. This rather defeats the purpose of using Cfqueryparam in the first place.

Conclusion

So as you can see there is more to SQL injection than meets the eye. My final thought is a comment on how developers approach applications. Dozens of times a year my staff and I are dumbfounded as we release a product for testing to a client, and they promptly do something with it that we never thought of before (sometimes even unearthing errors or bugs). Developers get tunnel vision pretty easily - and the smart ones are sometimes the worst. In the words of Grenalda Smortensgrammer before he leapt into the chasm - "It's not what I know that troubles me, it's all the stuff I know that I don't know".