# SQL Injection Part II (Make Sure You Are Sitting Down)

Posted At : July 18, 2008 3:52 PM | Posted By : Mark Kruger
Related Categories: Coldfusion Security

Back in February I wrote a blog post on SQL Injection that included an example of how a malicious user might inject into a character field even though ColdFusion escapes single quote marks. The attack involved *other* forms of escaping single quotes - and was effective against MySQL. This week I stumbled upon (more like a train wreck) an attack that is much more sophisticated - and also involves injection into a character field. I am told that others have discovered and written on this attack over the last few weeks - but I was unaware of it until a customer of ours was victimized. Amazingly, the specific real world attack I discovered and fixed allowed the hacker to append a string to *every char column in every table of the database*. It was so pervasive it left me wondering if it was SQL injection at all - until I found a URL entry that looked something like this:

```
someID=129;DECLARE%20@S%20CHAR(4000);SET%20@S=CAST(0x44...
%20AS%20CHAR(4000));EXEC(@S);
```

Note: in the spot above where it says "CAST(0x44..." I have left out a lengthy string of numbers.

First let me say that this code is ineffective against anyone using *cfqueryparam*. It is also ineffective against a simple "VAL( )" function in this case (since the user input was numeric - val() would have taken only the first few characters). But in this case the whole string - everything after *someID*= - was passed into the cfquery. Since the input was not validated the server attempted to execute it as valid SQL. What did it do?

The first part declares a variable as character string of 4000 characters.

```
DECLARE @S CHAR(4000)
```

The second part "sets" this variable with a unique CAST statement.

```
SET @S = CAST(0x44...*long string of numbers* AS CHAR(4000))
```

The final command executes the character string as SQL

```
EXEC(@S)
```

The trick here is the 0x* syntax inside of the CAST() function. It tells SQL that the values contained are actually ASCII codes and not translated characters. CAST then *translates* the numbers into actual characters that are subsequently executed by the EXEC() command. This obscures the attack in the logs, but here's how to unpack it. Find the string in the logs and tease out the CAST statement. Then, using Query analyzer try the following:

```
DECLARE @S CHAR(4000)

SET @S=CAST(40x44***** AS CHAR(4000))

PRINT @S
```

Note that 44***** is that long string of numbers. This will print out the actual SQL being executed. When I did this for the attack I had just fixed I found the following code being executed - and this resulted in a moment of begrudging awe in spite of my distaste for spammers and hackers.

```
DECLARE @T varchar(255),@C varchar(4000)

DECLARE Table_Cursor CURSOR FOR
select a.name,b.name
from sysobjects a,syscolumns b
where a.id=b.id
and a.xtype='u'
and (b.xtype=99
or b.xtype=35
or b.xtype=231
or b.xtype=167)

OPEN Table_Cursor FETCH NEXT FROM Table_Cursor INTO @T,@C
WHILE(@@FETCH_STATUS=0)
BEGIN exec('update ['+@T+'] set ['+@C+']=['+@C+']+''Malicious javascript here'' where '+@C+'
not like ''Same malicous js'')

FETCH NEXT FROM Table_Cursor INTO @T,@C
END CLOSE Table_Cursor

DEALLOCATE Table_Cursor
```

This code creates a cursor of all the user tables in the database and all the character columns within those tables. Then it appends a string to each of the columns. In this case the string was a link back to a web site, but it could have been much worse. As Guru Scott Krebs said to me, "...at least they had the decency to deallocate their cursor". In any case the result was that every single character column in the DB was infected with the malicious string. Since this was a news site it meant that every story, every title, every comment - virtually every piece of useful content on the site - had an embedded link back to the hacker's site.

## The Fix

If you are using MSSQL you are vulnerable to this specific attack or any attack like it. The fix is to *always use CFQUERYPARAM*, validate your user input (meaning anything

passed on the URL or in the Form scope) to be sure it is what it should be. Never rely on Client side validation for anything more than an enhanced user experience. Always always always write validation routines for form inputs. If *any* of these steps had been followed this client would not have been hacked successfully.

**A note on Ajax**

Recently someone asked me about form validation using Ajax. Form elements can be passed back to the server where they are checked and a result sent back to the browser which can display an error or submit the form accordingly. This can be a useful approach. For example, you can make calls to your database or session, check the user's shopping cart or CC etc. The result however is *exactly* like client side validation. You can enhance the user experience but you cannot *secure* user input using Ajax - since the form elements are still within the *users* control. Moreover, you are opening up an additional avenue of attack - particularly if you are touching the database in your Ajax code. Server side validation means that user input is examined on the server *every time* it is submitted. You cannot validate using Ajax and then *not* validate when the submission is made - in that way lies madness.