Combining SQL Query Strings and CFQUERYPARAM

Posted At : July 21, 2008 12:31 AM | Posted By : Mark Kruger
Related Categories: Coldfusion Security

If you have been following the muse the last few days you will know that I've had my shoulder to the wheel helping customers and fellow developers sort through making changes to their site to protect against a particularly malicious SQL Injection attack (read about the details here). Some of the folks who have contacted me are dealing with extra problems because their code uses string concatenation to build dynamic SQL strings. So the question has been asked a few times, *"How do I go about building an SQL string with CFQUERYPARAMs in it?"* Unfortunately, if you have chosen this approach it's going to be difficult to help you without seriously refactoring your code. Here's a few tips that can help, and one approach that *might* get you most of the way there.

Immediate Steps

First, you should realize that you are in for some late hours. So before you do anything else add some validation code as a stop gap measure. For example, you could use the UDF called isSQLInject found on Cflib.org to check the values of all Form and URL variables (be sure to add Declare, Cast, Char, Varchar, Exec, Execute and sp_sqlExecute to the list in the UDF). Using this UDF you could do something like this:

```
<!--- check the URL scope --->
<cfif isDefined('url')>
    <cfloop collection="#url#" item="uItem">
        <cfif isSQLInject(url[uITem])>
            <Cfabort>
        </cfif>
    </cfloop>
</cfif>
<!--- check the FORM scope --->
<cfif isDefined('form')>
    <cfloop collection="#form#" item="fItem">
        <cfif isSQLInject(form[fITem])>
            <Cfabort>
        </cfif>
    </cfloop>
</cfif>
```

We'll just call this crude arresting approach "Injectus Interuptus". It's not perfect but it will get us some protection while we fiddle with those queries.

Fixing Those Concatenated Query Strings

Before we "fix" these queries let's see an example of what they look like. Note, this is a very simple example.

```
<cfparam name="url.status" default="1"/>
<cfsavecontent variable="strSQL">
    SELECT     *
    FROM    users
    WHERE    status = '#url.status#'
    <cfif isDefined('url.username')>
        AND Username = '#url.username#'
    </cfif>
```

```
    <Cfif isDefined('url.email')>
        AND email = '#url.email#'
    </CFIF>
</cfsavecontent>
<!--- pass the string to the DB --->
<cfquery name="blah" datasource="test">
    #preservesinglequotes(strSQL)#
</cfquery>
```

Why is this approach used? The only time I've ever seen it as necessary is where the table, columns, where clause and joins are *all* unknown until runtime (as in some kind of report generator for example). Other than that I can think of virtually no reason to use this approach. Still, many programmers coming from ASP or JSP where this approach is required will find themselves using this approach - most of the time with a laborious group of CFSET statements rather than the lovely cfsavecontent code above.

You might notice that there is another problem with the code above. It has to do with the use of PRESERVESINGLEQUOTES( ). This function is designed for exactly this purpose - to keep CF from automatically escaping single quotes inside of a query. If you didn't use it then the code above that says "email = '#email#'" would be passed as "email = "#email#"" - and would error out. So preservesinglequotes is the function that allows me to treat my string as a true "sql" string. This has the negative side effect of removing the minimal protection against injection provided by this automatic escaping. A user could, for example, inject SQL in a URL param that looks like this:

```
email=bob@abc.com' or 1 = 1 --
```

Such an approach would allow me to terminate my email variable and then add an OR clause that would always evaluate as true. The double dashes simple comment out whatever is to the right of my malicious code - in this case a single quote mark. Because of PreserveSingleQuotes( ) this would work - since the string would remain "as is". So preservesinglequotes() makes character and numeric columns vulnerable to the simplest injection attacks.

The Fix

Well, you could fix it by simply moving the whole thing into the CFQUERY tag and adding CFQUERYPARAMS to it - like so:

```
<!--- pass the string to the DB --->
<cfquery name="blah" datasource="test">
    SELECT     *
    FROM    users
    WHERE    status=<cfqueryparam cfsqltype="CF_SQL_CHAR" value="#url.status#"/>

    <cfif isDefined('url.username')>
        AND Username = <cfqueryparam cfsqltype="CF_SQL_CHAR" value="#url.username#"/>
    </cfif>

    <Cfif isDefined('url.email')>
        AND email = <cfqueryparam cfsqltype="CF_SQL_CHAR" value="#url.email#"/>
    </CFIF>
</cfquery>
```

Of course in my simple example this is a pretty easy code change. In practice these string concatenations may include a lot of heavy logic. So here's a fix that may help - although in most cases it will be just as easy to move your code into your cfquery tag.

Using Params and Concatenation Together

This approach requires that you replace all your variables with localized SQL variables in your concatenation. If you are in the habit of using column names in your URL or Form variables this can actually be pretty straight forward - just replace #url.blah# with @blah. So the first step in our example would be to alter the cfsavecontent area to look like this:

```
<cfsavecontent variable="strSQL">
    SELECT     *
    FROM    users
    WHERE    status = @status
    <cfif isDefined('url.username')>
        AND Username = @username
    </cfif>
    <Cfif isDefined('url.email')>
        AND email = @email
    </CFIF>
</cfsavecontent>
```

We now have a SQL string that has no direct user content in it. Instead it has placeholders for the variables we will be using. The next step is to build our query in a special way (NOTE: looping through the URL scope is a tricky shortcut. It might very well make more sense to hard code your "declare" and "select" statements).

```
<cfparam name="url.email" default=""/>
<cfparam name="url.username" default=""/>
<cfparam name="url.status" default=""/>
<cfquery name="blah" datasource="test">
<cfloop collection="#url#" index="uItem">
    DECLARE     @#uItem# varchar(200)
    SELECT @#uITem# = <cfqueryparam cfsqltype="CF_SQL_CHAR" value="#url[uITem]#"/>
</cfloop>

#strSQL#

</cfquery>
```

A couple of notes here. If you have very complicated concatenation code you might be able to dress up your code using this approach more easily than completely refactoring it. Only you can judge if this is useful or not. Also note that the uItem now represents the "name" of the url variable instead of the "value". The name is not typically used to attack - but I can imagine that a hacker might use an especially malformed url name and sneak it through some web servers. What's the result of the code above? It's going to be a query that looks like this:

```
<cfquery name="blah" datasource="test">
DECLARE @email varchar(200)
SELECT @email = ***bound variable
DECLARE @username varchar(200)
SELECT @username = ***bound variable
DECLARE @status varchar(200)
SELECT @status = ***bound variable
<!--- The Code below will vary --->
    SELECT     *
```

```
    FROM    users
    WHERE    status = @status
    AND Username = @username
        AND email = @email
</cfquery>
```

Using this approach you may be able to keep your concatenation code largely intact while still leveraging the power and protection CFQUERYPARAM. My take is that you will likely have a better result by simply biting the bullet and moving your concatenation code into CFQUERY tags. Still, this gives you an idea for another option.

Finally, a note to all you muse readers who will be tempted to comment on how this is a bad idea. Please note that I have made that very point in this post. I'm attempting to answer a very specific question with the best solution that I could devise short of completely rewriting the concatenation code altogether. It's not perfect but it is the only thing I could come up with that comes close to answering the question "how do I go about building an SQL string with CFQUERYPARAMS in it?" On the other hand, if you have a brilliant idea that solves that issue let's hear it. The muse is all about learning something new.