Another SQLi Attack: Urchin.js

Posted At : April 16, 2010 2:11 PM | Posted By : Mark Kruger
Related Categories: ColdFusion, Coldfusion Security

I spent yesterday cleaning and inoculating another server infected with SQL Injection. Unless you have been living in a cave you know that SQL injection (SQLi) is the most common vulnerability of web based application. This is due to 2 factors - 1) almost all databases use numeric fields and B) web applications by nature pass user input into queries. Of course I could throw in there that web developers are often lax about inoculating their code. There is also the problem of legacy code - code that has been around since the dark ages of the late 90's. Of course SQLi has been around that long as well, but it is surprising how much legacy code chugs along for a decade or more with no problem in spite of the vulnerability.

Anyway, here's the skinny on the latest attack I found. It uses our old friend "Cast" in conjunction with the char() function of MS SQL. Note, this is not a new attack on the web - it's only new to me in that I've never battled this particular attack before.

The Attack

The customer called to tell me that his site had been flagged by Google as containing malware. I went to check it out. Now here's a quick tip. When a customer calls to tell you his site is affected by malware, you might be tempted to simply open his site in a browser to see what's up. It seems like I shouldn't have to say this, but don't do it. If you do, you may infect your own computer and then you are going to say to yourself (and possibly to the entire block) *DOH!* Instead use a browser in safe mode (FF has a version that runs in safe mode) or simply open the infected files in your editor and go searching for the infection.

Step 1: Find the Malware

I examined the page in question in my editor and determined that there was no malicious script in the file itself. There was however, some content output to the page from a table called "content" (names are changed here). My next step was to output that content using "htmleditFormat()".

```
<cfoutput>
        #htmlEditFormat(getContent.pgContent)#
    </cfoutput>
```

Typically an SQLi attack appends or prepends to a long string to a text are lengthy character field in one of your tables. The string is usually a script or tag that is invisible to the end user. Sure enough, at the end of the content I found the following:

```
</title><script src='http://*************/urchin.js'></script>
```

Note, I've obscured the host portion of the string. I don't really want to display the malware string anywhere in its entirety. In fact I discovered several versions of host that go with this attack. One version uses an IP address, one uses *google-serve01.com* and the final one uses *google-ahalytics.com* (note the "h"). The use of Google sounding URLs is an attempt to fool a regular user into thinking the script has something to do with Google, when in fact it does not. I especially thought the misspelling of "ahalytics" was clever. The use of an "h" to visually look like an "n" to a casual glance might indeed fool even a seasoned user - and of course the actual "google-analytics.com" host name will take you to the Google analytic tools. Anyway, a quick look at the content table showed that all of the "pgContent" columns had the same string appended to them, meaning every page served from the DB was also serving this malware.

Step 3: Find The Attack String

SQLi is a simple concept. A malicious user attaches some SQL to a query string or form parameter hoping it will get passed unprotected into your DB. All languages are vulnerable to this in different ways. In ColdFusion the most *common* vector for attack is an "integer" - typically an "ID" number of some sort being passed to the DB to look something up. If you are concerned about SQLi, one tip is to always log the query string in your log files. Most SQLi attacks still come via the URL query string. If you log that data in the log file it is actually quite easy to find because it is usually a very long string in relation to the other entries in the log. I open the log in notepad or text pad, scroll over to the right till I can't see any of the shorter entries, and then scroll down till I find some long ones and I examine them for likely attack code.

Sure enough, I quickly stumbled upon an SQLi string. Using my search tools I teased out some of the key words from the string and searched all the logs for the past 30 days. In this way I was able to determine the source of the *first* attack. Why is this important? If the infection is so bad it cannot be eliminated successfully, then it may be necessary to restore the DB to the backup prior to that date (you do have backups right? :). It's also important to check out the pattern of attacks to determine which tables are being targeted. Finally the logs will tell you *which page is being attacked*. If the string attacks multiple pages as if crawling your site it is likely probing for vulnerability. If, however, it attacks the same page over and over it has likely *found* a vulnerability and it is exploiting it.

Hmmm.... How Did They Get So Specific?

In this case I found that the attacker was able to specifically identify a vulnerable page and use it over and over again. He also attempted to attack several tables in the DB. These tables were not random attempts. *They all existed in the DB*. Wait a minute! He's attacking known table names in the customer DB. How does he know what table names to attack? The answer is surprisingly simple and not usually associated with SQLi. He throws errors. Let me explain. SQLi attaches known good SQL to the end of a numeric value in the hopes that the code will pass throug undetected to the database. So the URL (decoded) string might look like *"ID=44 update content set pgContent = pgContent + **encoded malicious string**"*. But what happens if the user passes in something to intentionally throw a DB error? What happens if the user does the following: *"ID=44 SELECT"*.

The database will throw an error right? It doesn't know what to do with that keyword "SELECT". Why does that matter? If the site has "robust exception information" enabled (and many many CF servers do) and if the site has not implemented CFERROR or a site wide error handler, ColdFusion will return a surprising treasure trove of useful and helpful information to the screen - including things like table names and datasource names. So all a clever hacker has to do to figure out table names is attempt to throw DB errors by passing wacky bits of SQL along on the query string. Sure enough, I found just that sort of wackiness by going back just a bit further.

Ok, let's examine this string shall we. Here is what the query string looked like. I've added line breaks and formatting for readability. I've also changed the actual string that's being "cast" (the part in the middle with all the %2Bchar(n) stuff in it):

```
2010-04-03 12:00:32 94.102.52.27 - 10.0.0.173 80 GET /somepage.cfm

LID=12501+update+content+set+pgContent=cast(pgContent+as+varchar(8000))%2Bcast(

char(84)%2Bchar(104)%2Bchar(101)%2Bchar(32)%2Bchar(101)%2Bchar(121)%2Bchar(101)
%2Bchar(115)%2Bchar(32)%2Bchar(97)%2Bchar(114)%2Bchar(101)%2Bchar(32)%2Bchar(116)
%2Bchar(104)%2Bchar(101)%2Bchar(32)%2Bchar(103)%2Bchar(114)%2Bchar(111)
%2Bchar(105)%2Bchar(110)%2Bchar(32)%2Bchar(111)%2Bchar(102)%2Bchar(32)
%2Bchar(116)%2Bchar(104)%2Bchar(101)%2Bchar(32)%2Bchar(102)
%2Bchar(97)%2Bchar(99)%2Bchar(101)

+as+varchar(8000))
```

Note the actual truly malicious string included a lot of spaces at the end and was much longer. You might be asking, why all the char() functions? Simple, this get's around the single quote escaping. Look carefully - not a single single quote in the entire string. The hacker doesn't have to worry that ColdFusion will escape his single quotes and ruin his well-formed SQL. The char( ) function in SQL is the exact equivalent to the chr( ) function in ColdFusion. It's another way to express a character. So his code is concatenating a long string of single characters together without using the single quote to group them. Consider this simpler SQL code:

```
SET name = char(80)+char(111)+char(111)+char(107)+char(121)
```

If your deconstructed each of the ascii numbers you would see that this is really the equivalent of:

```
SET NAME = 'P'+'o'+'o'+'k'+'y'
```

So it's concatenating individual characters together to do the same thing as "SET name = 'Pooky'". In the case of our attack the end result was the following SQL:

```
UPDATE  content
SET pgContent =
 CAST(pgContent AS varchar(8000) +
 CAST( '</title><script src='http://*************/urchin.js'></script>'
 AS varchar(8000))
```

How to Unpack It

To figure out what one of these long query strings contains requires a little modification in Query Analyzer or SQL studio. Tease out the "middle" of the string - all the char()%2B code and use urlDecode() to turn it into it's SQL version (as in char(10)+char(15)). Then try some code like this:

```
declare @badString varchar(8000)


select @badString=

cast(

char(84)+char(104)+char(101)+char(32)+char(101)+char(121)+char(101)+char(115)+char(32)+
char(97)+char(114)+char(101)+char(32)+char(116)+char(104)+char(101)+char(32)+char(103)+
char(114)+char(111)+char(105)+char(110)+char(32)+char(111)+char(102)+char(32)+char(116)+
char(104)+char(101)+char(32)+char(102)+char(97)+char(99)+char(101)

as varchar(8000))

print @badString
```

It should spell out what is contained in the string. 10 points to any muse reader who can tell me who made the quote spelled out in my sample above :)

Step 4: Repair and inoculation

Once you know what happened you have the tools to fix the data and to prevent a reoccurrence. There is a host ("yes Hefe' ... a plethora") of information on this blog and others on the topic of SQLi. All SQLi attacks can be prevented by the conventional use of CFQUERYPARAM. I have yet to hear anyone give me a viable reason to NOT use it. In conjunction with some snippets and hot keys its as easy as punch. So the first, best thing to do is to fix the attack vector. If you clean the data first it may reinfect before you fix the attack locus and you will end up having to do it again.

Let me know if you have experienced this attack recently. As always I look forward to comments and concerns, but please be polite.