Caffeine Monkey:
Automated Collection, Detection and Analysis of Malicious JavaScript

Ben Feinstein, CISSP
Daniel Peck
{bfeinstein, dpeck}@secureworks.com
SecureWorks, Inc.

Introduction

In recent years, the web browser has likely become the single most ubiquitous computing application. Web browsers can be found everywhere, from being embedded in video game consoles and handheld devices to being installed on large servers supporting the enterprise. As with any widely deployed type of software, web browsers have become an increasingly popular target for attack. This trend will only continue. Today's browsers represent a complex application stack foisted atop the Operating System's (OS) stack. Putting aside the complexity of achieving robust, cross-browser compatibility, web developers are essentially coding to the browser's stack, with the goal of portability across both browsers and OS platforms. The growth in popularity of "Web 2.0" sites that leverage the most advanced capabilities of the browser stack has contributed to blurring the distinction between data and executable code. This is especially apparent when considering support for dynamically interpreted scripting, in particular JavaScript. Due to the nature of dynamic script interpretation in the browser, it is difficult for many widely adopted security technologies to mitigate the client-side browser attack vector.

In our research, we examined the current state of JavaScript obfuscation and evasion techniques, approaches for collecting JavaScript samples from the wild, and methods for analyzing the collected scripts. We developed a suite of tools for collecting and indexing JavaScript, interpreting the scripting in a sandboxed environment, and performing functional analysis for manual, as well as automated detection mechanisms.

At the outset, we believed that investigating new approaches was warranted. Current methods tend to fall into two large categories: fully automated client honeypot systems or manual human analysis. Client honeypot technology offers a powerful way to actively identify sites attempting to exploit the browser, and has reportedly been used to find a number of zero-day attacks. However there are significant drawbacks to typical, high-interaction client honeypots, as they tend to result in a lot of overhead. In general, a client honeypot requires heavy-weight processing in order to detect exploits and recreate the virtual machine after each completed test. These systems usually produce very good results, but may not be feasible for the independent researcher or small organization to deploy and maintain. Additional processing is still required to move beyond elementary behavioral analysis. For example, if a malicious JavaScript sample only affects a particular version of Internet Explorer which is not being tested against, a previously unknown exploit could go undetected.

On the other side of the spectrum is the manual analysis performed by human researchers. These tedious techniques include walking through each layer of obfuscation, wrapping the sample script in a `<textarea>` HTML tag, or replacing `document.write()` with `alert()`. Some of the more recently crafted malicious scripts are explicitly designed to bypass these manual analysis techniques. Using these techniques can be both labor intensive and dangerous. With the increasing number of browser attacks, analysts run the risk of being exploited themselves. While these methods still have a useful place in the analyst's toolbox, we felt that a hybrid approach might prove to be more valuable.

Thus was born the Caffeine Monkey system. The core JavaScript engine, a safe JavaScript deobfuscator, logger and profiler, is based on extensions to the open source SpiderMonkey JavaScript implementation. The Monkey was hungry and needed to be fed, so we deployed an open source web crawling solution. A MySQL database served to organize the crawls, the retrieved documents, and our analysis results. The other assorted processing tasks were handled by a collection of custom Python, Perl, and shell scripting.

The source code and documentation for the Caffeine Monkey system will be made available during Black Hat USA 2007 at http://www.secureworks.com/research/tools/. All code will be released under an as-yet-to-be-determined OSI approved open source license.

This paper will explore the predominant obfuscation techniques and how Caffeine Monkey can find them, dissect them, and reveal their true functionality. We will also share our thoughts for future research in this area.

JavaScript Obfuscation & Evasion Techniques

To begin, we will examine some of the basic forms of obfuscation, going from the least effective, and most easily detected, to the most effective. While we are looking at these techniques in the context of the JavaScript language, in many case these same concepts are applicable to other scripting languages.

Whitespace randomization is likely the simplest obfuscation technique to implement. Taking advantage of the fact that JavaScript ignores whitespace, an attacker can strategically scatter whitespace characters throughout their code. Without changing the semantics of the JavaScript, this technique will yield large changes in the script's on-the-wire binary representation. It is trivial at runtime to determine the behavior of a script. However, many security technologies rely on content matching for detection and would be blinded by this obfuscation technique. Whitespace randomization is demonstrated by the following scripting:

```
ASCII: var i = “foooo”;

Hex: 7661 7220 6920 3d20 2266 6f6f 6f6f 223b 0a

ASCII: var  i     =     “foooo”;

Hex: 7661 7209 2069 2009 2020 3d20 2020 2020 2022 666f 6f6f 6f22 3b
```

Figure 1

As you can see, the on-the-wire binary representation is significantly different but the semantics of the scripts are identical.

Another basic evasion technique involves the addition of random comments and the manipulation of existing comments in scripting. Just like whitespace, comments are ignored by JavaScript. This is very similar to whitespace randomization, in that the actual code remains unchanged while the on-the-wire binary representation is dramatically altered. Manipulation of comments can also be effective in confusing an analyst. Just as in the case of whitespace randomization, runtime analysis of the script's behavior is straightforward. However, content matching would have difficulty determining the runtime behavior.

Other, more sophisticated obfuscation techniques exist offering even better abilities of evasion. String obfuscation usually involves a custom decoder, anywhere from a simple XOR function to a more complex Caesar cipher or even more elaborate methods. Although this technique is normally not needed to bypass detection mechanisms, it can make analysis much more difficult for the researcher and help maintain the script's effectiveness over a longer period of time. This technique can also be as simple as splitting the string into multiple variables and concatenating them later in the script, perhaps using the `document.write()` method in combination with `String.fromCharCode()`. These strings can also be encoded using various hexadecimal and Unicode representations. The following example shows several ways in which the string “we've got a problem” can be represented.

```
“we%27ve%20got%20a%20problem”

“%77%65%27%76%65%20%67%6F%74%20%61%20%70%72%6F%62%6C%65%6D”

“\x77\x65\x27\x76\x65\x20\x67\x6F” +
“\x74\x20\x61\x20\x70\x72\x6F\x62\x6C\x65\x6D”

“%u0077\u0065\x27%76%65%20\x67%6F%74\u0020%61%20%70%72%u006F%62\x6C%65\x
6D”
```

Figure 2

As you can see, there are many ways to represent this textual phrase, making purely signature based detection impractical due to the large number of different variations. With just the example string and the few encoding forms we've mentioned there are more than 5^{19} possible combinations!

Another obfuscation technique is variable name randomization and function pointer reassignment. A variable or function can be reassigned to another variable or function, potentially misleading analysts trying to decipher the code. This technique has also proved effective at bypassing a variety of security technologies. Short of keeping track of all the variable and function assignments at runtime, a security device would have no assurance that a function named `unescape()` is actually the function defined in by JavaScript specification as `unescape()`. With most security devices suffering from upper-bound requirements on space and time complexity this task becomes increasingly infeasible. For example:

```
randomFunctionName = unescape;

function2 = eval;

var A1 =
randomFunctionName("%61%6c%65%72%74%28%22%77%65%27%72%65%20%67%6f%74%2
0%61%20%70%72%6f%62%6c%65%6d%20%58%65%72%65%22%29");

function2(A1);
```

Figure 3

As shown above, detection is becoming more and more difficult. Integer obfuscation is yet another technique used for evasion. Suppose a certain memory address was needed by a script, but the presence of this memory location in the code could be flagged as suspicious by a variety of detection mechanisms. Using integer obfuscation we can generate the same number with simple mathematical functions. For instance, "0x04000000" could be expressed as 16,777,216 * 42, or any number of other ways.

One of the most sophisticated obfuscation techniques is block randomization. This involves structurally changing a script's statements and code paths to be functionally identical but syntactically different. Typically a script's `if/else` block and `while/for` loops are restructured, however other constructs can also be altered. `while`, `for`, and `do-while` loops can be transformed in a number of ways:

```
for (i = 0; i < 100; i++) { /* for loop */ }

while (i < 100) { i++; /* while loop */ }

do { i++; /* do..while loop */ } while (i < 100)
```

Figure 4

Used alone, each of these techniques can be effective at obfuscating the true functionality of a script. Combined together, they clearly make the job of effective detection very difficult.

Along Came a Spider…

Of course, without evidence of these techniques being actively used in the wild our research would be purely theoretical. With this in mind we set out to crawl selected portions of the web, focusing on sites with a large amount of user created content. Intuitively, we thought that a crawl originating from www.myspace.com would yield an interesting sample of JavaScript.

Not wanting to reinvent the wheel, the open source Heritrix software package was employed. Heritrix is an Internet-scale web crawler developed by the Internet Archive for their own use. Setup was straight forward for those with experience using Java, and before we knew it we were collecting JavaScript from the wild.

Heritrix is designed around the concepts of Profiles and Jobs. A Profile is used to define all aspects of web crawler configuration. A Job is based on a Profile and can override configuration options inherited from the Profile. A Job specifies the seed URIs which are the initial starting points of the web crawl. Jobs are queued for processing and will be picked up by an idle crawler thread. The overall state of a web crawl is maintained internally as a Frontier. Jobs can be paused and resumed by recovering the Job state from a previously saved Frontier.

Heritrix supports custom workflows through the use of the Crawl Operator, Crawl Organization, and Crawl Job Recipient properties. Fine-grained control is given over search strategies (e.g. "Deep", "Broad") and the rules used to select or reject URIs to pursue. Several policies for respecting site `robots.txt` files are configurable. Extensive configuration options to limit resource consumption are available.

Before being able to run a Job using the default Profile we were forced to modify the values of the `User-Agent` and `From` HTTP headers to meet Heritrix's restrictive criteria. The `User-Agent` is required to follow a form like "Mozilla/5.0 (compatible; heritrix/1.12.1 +PROJECT_URL_HERE)" and the `From` header must contain a valid email address. We created a new profile named "Caffeine Monkey" that was based on the default Profile but defined our particular values for the `User-Agent` and `From` headers. Being the good Internet citizens that we are, we configured our crawler to identify itself as belonging to SecureWorks Research.

Using a single seed of www.myspace.com we collected approximately 225,000 web documents over a continuous period of about three and a half days, with a total yield of 7.9 GB. Of these, 364 documents (4.5 MB) were of Content-Type `application/x-javascript` or `text/javascript`, comprising about 0.2% of the total. This comprised our sample.

Pre-processing and Indexing

Heritrix saves the content it collects in an archival file format referred to by its three character file extension, ARC. By default each archive file grows to about 100MB and can contain thousands of spidered documents. We needed a way to efficiently pull the

JavaScript documents out of the archive files in order to subsequently analyze them using the Caffeine Monkey engine.

Fortunately an individual at the University of Michigan had already written a rough collection of Perl scripts for working with Heritrix archive files. One of these scripts was designed to index a collection of archive files into a MySQL database containing a single table. We extended the rudimentary database schema to encompass two new tables: the first representing the Heritrix Jobs that collect the archives, and the second for storing the results from analyzing the JavaScript samples. The existing Perl scripting was modified, adding support for the concept of Heritrix Jobs and the new job database table, as well fixing several bugs.

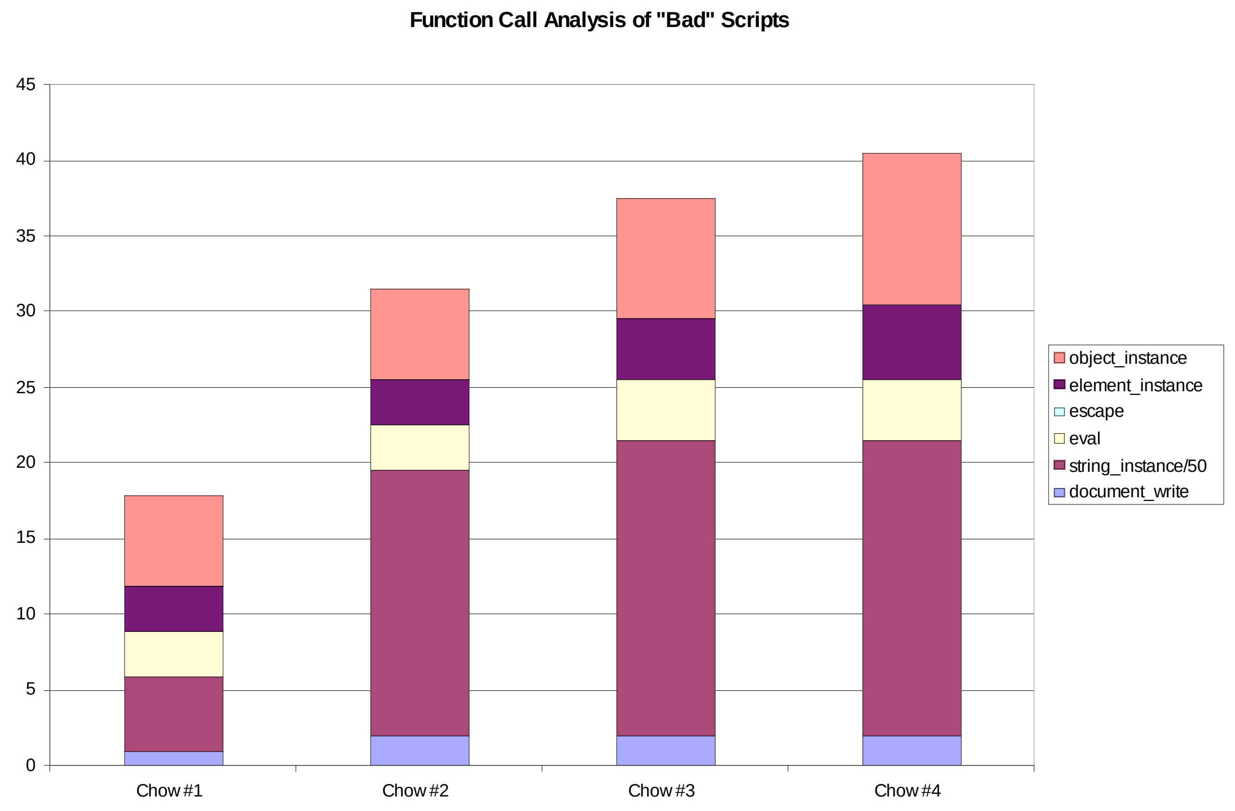
The relevant details of each URI that was retrieved were stored in the database. These included:

- URI
- Heritrix Job the URI was retrieved as a part of
- MIME Content-Type
- 3-digit HTTP response code received from the server providing the URI
- ARC file containing the retrieved document
- Index into the ARC file of the start of the retrieved document
- Content-Length, used along with the file index to extract the retrieved document
- Timestamp

Analysis and Reporting

With all the archive files from our MySpace crawl indexed in our database, it was relatively straight forward to retrieve all the JavaScript documents for analysis. Several Python classes were created to automate the process of analyzing the collected scripts. Each JavaScript sample was run through the Caffeine Monkey engine and its runtime log was analyzed. Statistics on runtime execution were generated from the log and stored in the database. The Caffeine Monkey engine hooks a number of interesting functions. For each script run through the engine, function call statistics were captured.

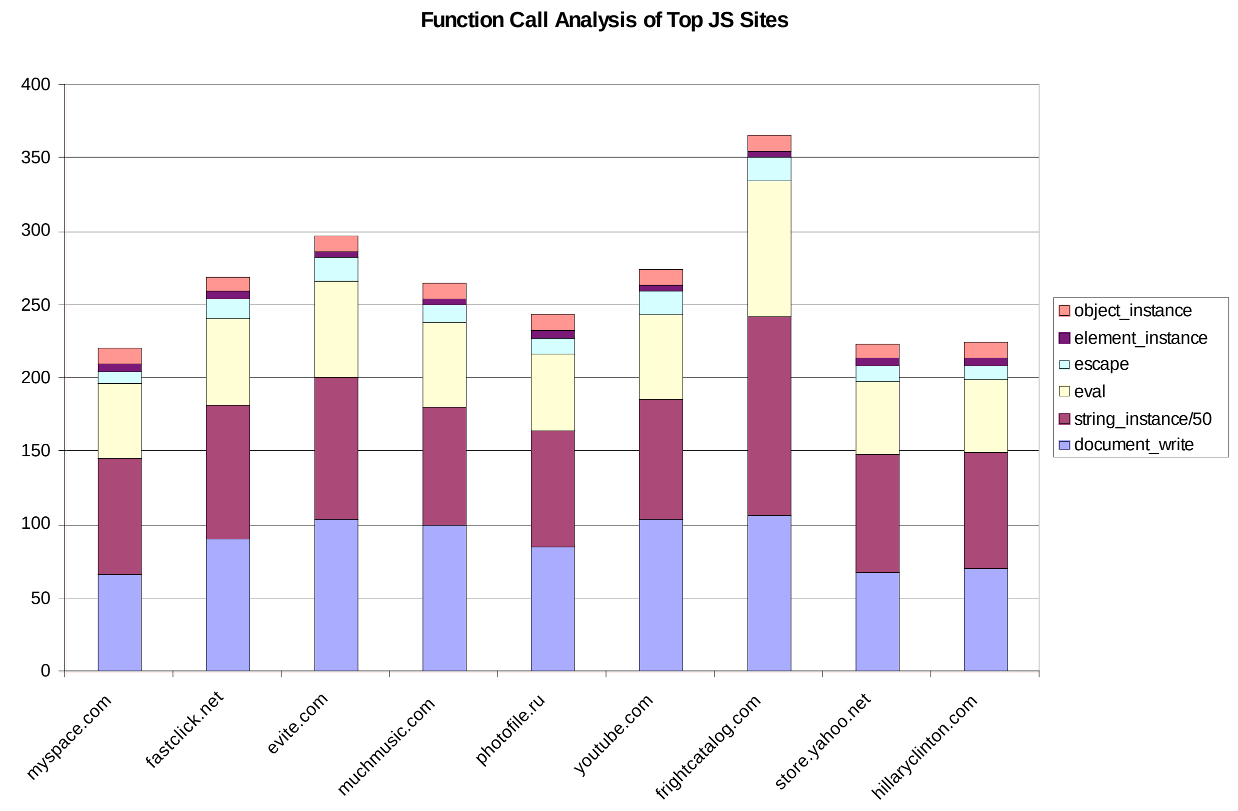
At the time of this writing, we had not identified any malicious JavaScripts among the 364 samples collected through our MySpace web crawl. In order to get samples of known malicious scripts we sent requests to several other security researchers. In all, we received four good samples which we labeled Monkey Chow #1 through #4.



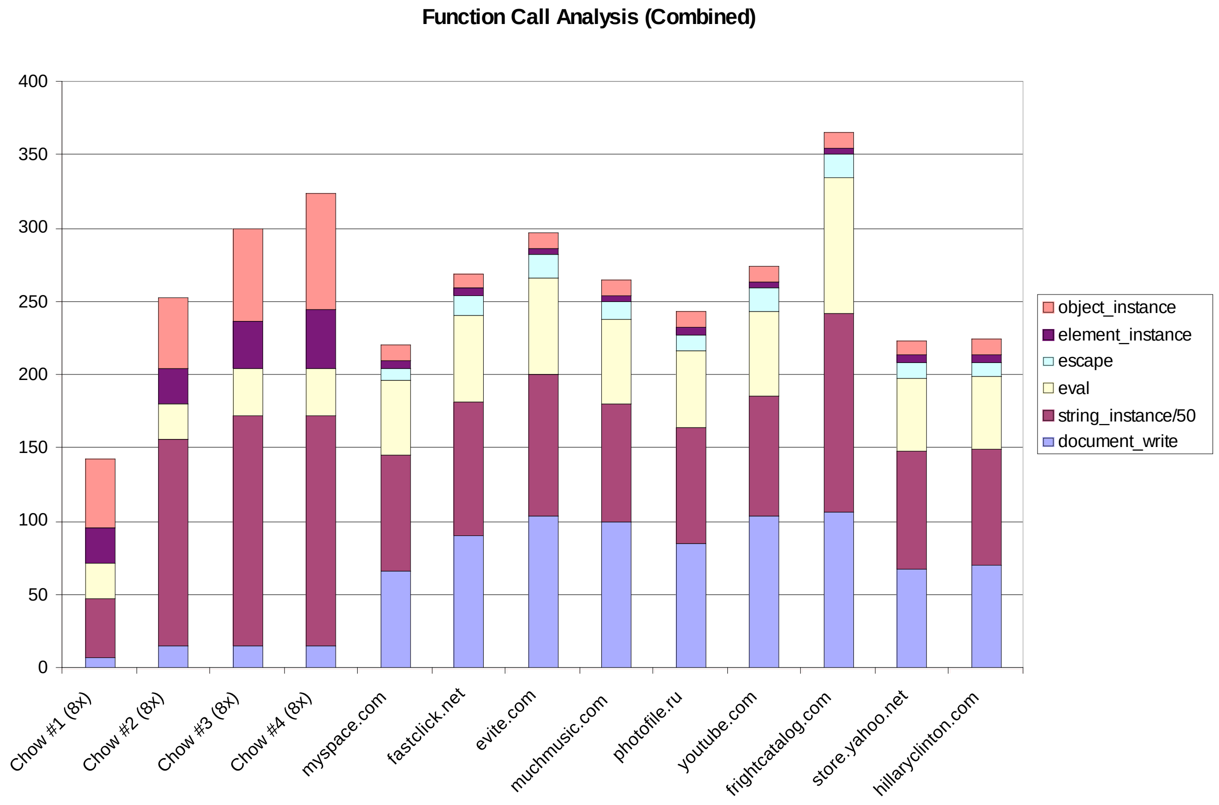
Note that the numbers for string instantiations are scaled by a factor of $^1/_{50}$ in order to yield better chart scaling. There are orders of magnitude more string instantiations than the other categories because string concatenation is a popular operator, and because string concatenation creates three new string instances for each use of the "+" operator.

What is important to look at is not the absolute number of times each function is called, but rather the ratios of the function call counts to one another. For three out of the four samples above, the ratios between the count of object instantiations, element instantiations, calls to `eval()`, and string instantiations are strikingly similar. Future research might involve analyzing larger samples of malicious JavaScript to see if these trends hold.

Based on the results of our MySpace web crawl, we grouped the collected JavaScript URIs by their domain name fragment. We then sorted the domain names by the number of JavaScript documents retrieved from each. Myspace.com was the top domain by the number of JavaScript documents collected, yielding a sample size of twenty-seven. Grouping the domains by the top six counts of JavaScript documents yielded nine domains because of ties. hillaryclinton.com rounded out the top nine, with five JavaScripts collected from the domain. For each of the top domains we generated aggregate statistics and charted the results is the same fashion as our Monkey Chow samples.



Again, the absolute magnitudes of the function call counts are not as important as the ratios between them. There does appear to be similar ratios of function calls across the nine domains sampled. At this point we charted our monkey chow samples against our analysis of the top JavaScript domains from our MySpace crawl.



There are clear differences between the malicious JavaScript samples and the benign JavaScripts collected during the MySpace crawl. The monkey chow statistics were scaled by a factor of 8. We anticipated a difference in the scales of the function call counts between the malicious samples and the benign samples. The four malicious samples were short script fragments; however, the benign samples tended to be large feature-rich scripts. Larger scripts will tend to have higher function call counts simply because they have more lines of code.

It appears that benign scripts make significantly more use of the `document.write()` method while malicious scripts make relatively more use of string instantiation. Our small sample of malicious scripting also instantiates relatively more objects than our samples of benign JavaScript. Malicious scripts tended to programmatically create DOM elements more frequently than the benign scripts. Use of the `eval()` function is also relatively more common in the benign scripts than in our sample of malicious scripts.

Caffeine Monkey at Work

In order to illustrate the utility of the Caffeine Monkey engine, we will now analyze a sample of obfuscated JavaScript found in the wild. This sample of JavaScript isn't truly destructive, but offers a useful overview of the time consuming, error prone, and potentially dangerous process of manual human analysis.

```
function I(mK,G){if(!G){G='Ba,%7(r_)`m?dPSn=3J/@TUc0f:6uMhk;wyHZEs-
^O1N{W#XtKq4F&xV+jbRAi9g';}var R;var TB='';for(var
e=0;e<mK.length;e+=arguments.callee.toString().replace(/\s/g,'').length-
535){R=(G.indexOf(mK.charAt(e))&255)<<18|(G.indexOf(mK.charAt(e+1))&255)
<<12|(G.indexOf(mK.charAt(e+2))&255)<<(arguments.callee.toString().repla
ce(/\s/g,'').length-
533)|G.indexOf(mK.charAt(e+3))&255;TB+=String.fromCharCode((R&16711680)>
>16,(R&65280)>>8,R&255);}eval(TB.substring(0,TB.length-
(arguments.callee.toString().replace(/\s/g,'').length-
537)));}I('friHMU&E6-
=#MV`OMr@^`4K/=&``@(=;/7(S3&Ta3F@i)ZOwMs(40V`Ou_=y)(PJ=4Fy:_3Fu%^X?VMVMq
jOM_Ob6V=#0xdXuV3j6r@XnV`EfHF-mx3X0VTWfUjF?-`EfsTqusTqmquynHtX`q{-
uxPq:caFnyuOSqB;),B;),B;),Bm),B;');
```

Figure 4

Such a script can appear more than a little intimidating at first glance. Considerable manual analysis of this sample would be required to determine the scripts functionality. Function `I()` is defined as taking two arguments, `mk` and `G`. `I()` performs lots of bit-shifting and substitution before assigning the result of a call to `String.fromCharCode()` to the variable `TB`. The whole mess is then passed to the `eval()` function.

The details of how this was accomplished are left as an exercise for the reader, but it all boils down to a single function call.

```
eval("document.write('<SCRIPT LANGUAGE="Javascript"
SRC="http://www.itzzot.cc/style/?ref='+document.referrer+'"></'+'script>
');");
```

Figure 5

Requesting this URI returns an even uglier piece of obfuscated JavaScript.

The gory details of decoding this sample are outside the scope of this document, but clearly it would present a challenge to either a novice analyst or a time conscious veteran. Automating the deobfuscation process allows human analysts to examine the evasion techniques being used by attackers, but also yields some additional wisdom that could be leveraged in a heuristics based detection system. Low-interaction client honeypots could use the Caffeine Monkey engine to rapidly analyze large collections of JavaScript. Using the GNU tool chain's philosophy, the engine can easily be tied together with other utilities to automate the analyst's workflow.

Caffeine Monkey JavaScript Engine

We began developing the Caffeine Monkey engine using the code base of the Mozilla SpiderMonkey (JavaScript-C) Engine, an embeddable open source JavaScript engine implemented in the C language. The initial code base offered a rudimentary interpreter and had an architecture allowing for easy extensibility. We internally hooked the functions we thought most likely to be used in obfuscation and added runtime logging output. Runtime logging allowed us to observe the flow of execution without getting into interactive script debugging. One of the first methods we wrapped and logged was `eval()`. By hooking into the execution path at the interpreter level, obfuscation is completely avoided as we log the actual string passed to `eval()` and where in the script's execution the call is made.

We hooked the string concatenation method, allowing not only a final view of the string, but also creating a record of the order and timing of when string concatenations occurred during execution. This feature is potentially useful for script fingerprinting, as particular obfuscation tools tend to use a similar method to obfuscate the strings each time that are run.

After instrumenting only these two basic functions we were able to produce a very informative log. The log served to reduce the obfuscation example above down to a handful of log lines that are easily readable by both human and machine. For the sake of disclosure, the example above sets a known spyware cookie in the browser and redirect the browser to a niche online dating site.

A full implementation of Document Object Model (DOM) logging is in development at the time of this writing, as the original SpiderMonkey code base only provided the basic JavaScript methods and objects.

A more in depth discussion of the Caffeine Monkey engine's functionality is outside the scope of this paper. Documentation will be made available along with the source code to our tools. The example above makes it clear that an automated analysis tool working at the interpreter level offers the potential to increase efficiency when analyzing scripts using common obfuscation and evasion techniques.

Directions for Future Research

There are many areas for improvements and extensions to these tools. The core engine could be leveraged in an application proxy setting, removing or flagging potentially dangerous JavaScript. With further optimization, Caffeine Monkey and its analytical results could be used within Network Intrusion Detection or Prevention Systems. The tool can also be simply used as a command-line utility by attackers testing their obfuscation schemes or by security analysts working to reverse the efforts of the attackers.

References

Mozilla SpiderMonkey (JavaScript-C) Engine
http://www.mozilla.org/js/spidermonkey/

Seifert, C., Welch, I. and Komisarczuk, P., “Taxonomy of Honeypots”, Victoria University of Wellington, Wellington, 2006.
http://www.mcs.vuw.ac.nz/comp/Publications/index-byyear-06.html

Heritrix web crawler
http://crawler.archive.org/

Baker, N., “Archiving Websites”, HOWTO, University of Michigan, School of Information, February 2005.
http://www.si.umich.edu/mirror/how_to/

Japanese language site apparently discussing malicious JavaScript samples
http://bbs.blueidea.com/thread-2740516-1-5.html

The Strider HoneyMonkey Project
http://research.microsoft.com/honeymonkey/

MITRE Honeyclient Project
http://www.honeyclient.org/trac

Wang, K., “Using Honeyclients to Detect New Attacks”, ToorCon 2005.
http://www.synacklabs.net/honeyclient/Wang-Honeyclient-ToorCon2005.pdf

HoneyC – The Low-Interaction Client Honeypot / Honeyclient
http://honeyc.sourceforge.net/

Caswell, B. and HD Moore, “Thermoptic Camouflage – Total IDS Evasion”, Black Hat USA 2006.
http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Caswell.pdf

ECMAScript Specification (ECMA-262)
http://www.ecma-international.org/publications/standards/Ecma-262.htm

Month of Browser Bugs
http://browserfun.blogspot.com/

eVade O’ Matic Module for Metasploit
http://blog.info-pull.com/2007/01/update-17th-october-2006-aviv-posted.html